OpenVOS STREAMS TCP/IP Programmer's Guide

Stratus Technologies
R420-12

Notice

The information contained in this document is subject to change without notice.

UNLESS EXPRESSLY SET FORTH IN A WRITTEN AGREEMENT SIGNED BY AN AUTHORIZED REPRESENTATIVE OF STRATUS TECHNOLOGIES, STRATUS MAKES NO WARRANTY OR REPRESENTATION OF ANY KIND WITH RESPECT TO THE INFORMATION CONTAINED HEREIN, INCLUDING WARRANTY OF MERCHANTABILITY AND FITNESS FOR A PURPOSE. Stratus Technologies assumes no responsibility or obligation of any kind for any errors contained herein or in connection with the furnishing, performance, or use of this document.

Software described in Stratus documents (a) is the property of Stratus Technologies Ireland, Ltd. or the third party, (b) is furnished only under license, and (c) may be copied or used only as expressly permitted under the terms of the license.

Stratus documentation describes all supported features of the user interfaces and the application programming interfaces (API) developed by Stratus. Any undocumented features of these interfaces are intended solely for use by Stratus personnel and are subject to change without warning.

TCP Wrappers copyright information:

Manual Name: *OpenVOS STREAMS TCP/IP Programmer’s Guide*

Part Number: R420
Revision Number: 12
OpenVOS Release Number: 19.3.0
Publication Date: March 2021

Stratus Technologies, Inc.
5 Mill and Main Place, Suite 500
Maynard, Massachusetts 01754-2660

Contents

Figures

Tables

Preface

The *OpenVOS STREAMS TCP/IP Programmer's Guide* (R420) documents the STREAMS TCP/IP (STCP) application programming interface (API) on a Stratus module running Open Virtual Operating System (OpenVOS) Release 19.3.0 or later.

This manual is intended for programmers who are designing STCP applications to run on an OpenVOS computer. It assumes familiarity with the Transmission Control Protocol/Internet Protocol (TCP/IP) family of communications protocols. For information on the basic concepts underlying the TCP/IP architecture, see one of the many trade books that describe TCP/IP.

Manual Version

This manual is a revision. Change bars, which appear in the margin, note the specific changes to text since the previous publication of this manual.

This revision incorporates changes in the following sections:

- "Specifying How the TCP Maximum Segment Size Is Determined" on page 4-18
- "Window Size" on page 4-15
- The following function descriptions:
  - `send`
  - `sendmsg`
  - `sendto`

Manual Organization

This manual contains the following chapters and appendixes.

Chapter 1 introduces STCP.

Chapter 2 describes how an STCP application uses the socket interface functions.

Chapter 3 explains how to use the functions included in the STCP socket library that perform tasks supporting data transmission.

Chapter 4 describes programming considerations that are of concern when developing and testing applications. It also explains how to compile and bind an application.

Chapter 5 describes each STCP function in detail.

Appendix A provides sample programs that use the STCP API.

Appendix B lists deprecated socket options.

Related Manuals

For information on other aspects of the STCP product, including hardware and software requirements, configuration and administration, command descriptions, and differences between STCP and OpenVOS OS TCP/IP, see the other manuals in the STCP manual set.

- *OpenVOS STREAMS TCP/IP Migration Guide* (R418)
- *OpenVOS STREAMS TCP/IP Administrator's Guide* (R419)
- *VOS STREAMS TCP/IP User's Guide* (R421)

For information about OpenVOS STREAMS, see the *OpenVOS Communications Software: STREAMS Programmer's Guide* (R306).

For information about the OpenVOS Standard C (that is, the ANSI C-compliant) implementation of the C language, see the *OpenVOS Standard C User's Guide* (R364) and the *OpenVOS Standard C Reference Manual* (R363).

For information on POSIX.1, see the following documentation:

- *OpenVOS POSIX.1: Conformance Guide* (R217M)
- *OpenVOS POSIX.1 Reference Guide* (R502)

Notation Conventions

This manual uses the following notation conventions.

Warnings, Cautions, Notices, and Notes

Warnings, cautions, notices, and notes provide special information and have the following meanings:

WARNING

A warning indicates a hazardous situation that, if not avoided, could result in death or serious injury.

AVERTISSEMENT

Un avertissement indique une situation dangereuse qui, si pas évitée, pourrait entraîner la mort ou des blessures graves.

CAUTION

A caution indicates a hazardous situation that, if not avoided, could result in minor or moderate injury.

MISE EN GARDE

Une mise en garde indique une situation dangereuse qui, si pas évitée, pourrait entraîner des blessures mineures ou modérées.

NOTICE

A notice indicates information that, if not acted on, could result in damage to a system, hardware device, program, or data, but does not present a health or safety hazard.

NOTE

A note provides important information about the operation of an ftServer system or related equipment or software.

Typographical Conventions

The following typographical conventions are used in this manual:

- Italics introduces or defines new terms. For example:

  The *master disk* is the name of the member disk from which the module was booted.

- Boldface emphasizes words in text. For example:

  Every module **must** have a copy of the `module_start_up.cm` file.

- Monospace represents text that would appear on your terminal's screen (such as commands, subroutines, code fragments, and names of files and directories). For example:

  ```
  change_current_dir (master_disk)>system>doc
  ```

- Monospace italic represents terms that are to be replaced by literal values. In the following example, the user must replace the monospace-italic term with a literal value.

  ```
  list_users -module module_name
  ```

- Monospace bold represents user input in examples and figures that contain both user input and system output (which appears in monospace). For example:

  ```
  display_access_list system_default

  %dev#m1>system>acl>system_default

  w  *.*
  ```

Syntax Notation

A *language format* shows the syntax of an OpenVOS Standard C statement, portion of a statement, declaration, or definition. When OpenVOS Standard C allows more than one format for a language construct, the documentation presents each format consecutively. For complex language constructs, the text may supply additional information about the syntax.

The following table explains the notation used in language formats.

The Notation Used in Language Formats

Notation	Meaning
element	Required element.
element...	Required element that can be repeated.
{*element_1 element_2*}	List of required elements.
{*element_1 element_2*}...	List of required elements that can be repeated.
{*element_1* *element_2*}	Set of elements that are mutually exclusive; you must specify one of these elements.
[*element*]	Optional element.
[*element*]...	Optional element that can be repeated.
[*element_1 element_2*]	List of optional elements.
[*element_1 element_2*]...	List of optional elements that can be repeated.
[*element_1* *element_2*]	Set of optional elements that are mutually exclusive; you can specify only one of these elements.
Note: Dots, brackets, and braces are not literal characters; you should **not** type them. Any list or set of elements can contain more than two elements. Brackets and braces are sometimes nested.	

In the preceding table, *element* represents one of the following OpenVOS Standard C language constructs.

- reserved words (which appear in monospace)
- generic terms (which appear in monospace italic) that are to be replaced by items such as expressions, identifiers, literals, constants, or statements
- statements or portions of statements

A reserved word has special meaning for the compiler; you cannot define a reserved word as an identifier. A keyword is a reserved word that is underlined in a language format. The compiler uses keywords to generate the code. Reserved words that are not underlined enhance readability but have no effect on compilation.

The elements in a list of elements must be entered in the order shown, unless the text specifies otherwise. An element or a list of elements followed by a set of three dots indicates that the element(s) can be repeated.

The following example shows a sample language format.

In examples, a set of three vertically aligned dots indicates that a portion of a language construct or program has been omitted. For example:

Getting Help

If you have a technical question about ftServer system hardware or software, try these online resources first:

- **Online documentation at the StrataDOC Web site**. Stratus provides complimentary access to StrataDOC, an online-documentation service that enables you to view, search, download, and print customer documentation. You can access StrataDOC at the following Web site:

  http://stratadoc.stratus.com

- **Online support from Stratus Customer Service**. You can find the latest technical information about an ftServer system in the Stratus Customer Service Portal at the following Web site:

  http://www.stratus.com/go/support

  The Service Portal provides access to Knowledge Base articles for all Stratus product lines. You can locate articles by performing a simple or advanced keyword search, viewing recent articles or top FAQs, or browsing a product and category.

  To log in to the Service Portal, enter your employee user name and password or, if you have not been provided with a login account, click **Register Account**. When registering a new account, ensure that you specify an email address from a company that has a service agreement with Stratus.

If you cannot resolve your questions with these online self-help resources, and the ftServer system is covered by a service agreement, contact the Stratus Customer Assistance Center (CAC) or your authorized Stratus service representative. To contact the CAC, use the Service Portal to log a support request. Click **Customer Support** and **Add Issue**, and then complete the **Create Issue** form. A member of our Customer Service team will be glad to assist you.

Commenting on This Manual

You can comment on this manual using one of the following methods. When you submit a comment, be sure to provide the manual's name and part number, a description of the problem, and the location in the manual where the affected text appears.

- From StrataDOC, click the **site feedback** link at the bottom of any page. In the pop-up window, answer the questions and click **Submit**.
- From any email client, send email to `comments@stratus.com`.
- From the Stratus Customer Service Portal, log on to your account and create a new issue.

Stratus welcomes any corrections and suggestions for improving this manual.

Chapter 1
Introduction to the STREAMS TCP/IP Application Programming Interface

STREAMS TCP/IP (STCP) is a STREAMS-based implementation of the Transmission Control Protocol/Internet Protocol (TCP/IP) family of communications protocols for OpenVOS modules. Unlike the older VOS OS TCP/IP product set, the STCP product functions within a standard STREAMS environment.

STCP supports a number of standard TCP/IP applications and relies on OpenVOS device drivers and Stratus communications hardware to support standard local area network (LAN) architectures such as Ethernet. It also provides a standard socket programming interface.

This manual describes how to write STCP applications. This chapter introduces the STCP application programming interface (see ‘‘The STCP Application Programming Interface’’ on page 1-2). The manual continues with the following additional chapters and appendixes:

- Chapter 2, ‘‘Using the Socket Interface Functions”
- Chapter 3, ‘‘Using the Supporting STCP Functions”
- Chapter 4, ‘‘Programming Considerations”
- Chapter 5, ‘‘Socket-Library Functions”
- Appendix A, ‘‘STCP Sample Programs”
- Appendix B, ‘‘Deprecated Socket Options

This manual describes the legacy functionality of the STCP application programming interface. OpenVOS also supports POSIX.1 functionality. POSIX.1 refers to Part 1 of the IEEE POSIX standard, which is a system application program interface (API). OpenVOS support of POSIX.1 enables OpenVOS programmers to port applications that conform to the POSIX.1 standard. To create POSIX.1-compliant source code for OpenVOS, use the following documentation:

- *OpenVOS POSIX.1: Conformance Guide* (R217M), which describes how the OpenVOS POSIX.1 implementation adheres to or deviates from the POSIX

standard. This document is available only on the OpenVOS StrataDOC Web site: http://stratadoc.stratus.com.

- *OpenVOS POSIX.1 Reference Guide* (R502), which documents the OpenVOS POSIX features.

The STCP Application Programming Interface

The application programming interface (API) of STCP consists of a socket library that enables you to write applications that use the standard TCP/IP protocols to communicate with other peer processes in a TCP/IP network. The socket library comprises a special group of C functions, called the *socket interface* or *networking functions*, which are based on the Berkeley Software Distribution (BSD™) UNIX® Version 4.3 socket interface.

You can develop applications and bind them with object modules located in the `(master_disk)>system>object_library` directory (see ‘‘Binding an Application’’ on page 4-3). The header files that STCP uses reside in the `(master_disk)>system>include_library` directory and its subdirectories.

The STCP socket library is ANSI C- and POSIX.1-compliant. STCP supports only applications written in the C language. Stratus **strongly recommends** that you compile your applications with the OpenVOS Standard C compiler (invoked with the `cc` command), which is ANSI C-compliant. For more information about the OpenVOS Standard C compiler, see the following documentation:

- The `cc` command description in the *OpenVOS Commands Reference Manual* (R098)
- *OpenVOS Standard C Reference Manual* (R363)
- *OpenVOS Standard C User’s Guide* (R364)

Chapter 2
Using the Socket Interface Functions

This chapter describes how an STCP application uses the socket interface functions. It contains the following sections.

Chapter 5 describes in detail the functions discussed in this chapter.

Creating a Socket

This section describes how to create sockets.

A *socket* is a logical communications endpoint at which a process accesses, or interfaces with, a TCP/IP protocol that will handle communications across the network. A process that wants to communicate with a peer process must use a socket.

A process creates a socket by calling the `socket` function. The `socket` function has the following syntax.

```
int socket(int af, int type, int prot);
```

The `socket` function has three arguments.

- The `af` argument identifies an address family. Specify one of the following values:
  - `AF_INET` for the Internet address family with IPv4 protocol support.
  - `AF_INET6` for the Internet address family with IPv6 protocol support.

NOTES

1. You cannot mix usage of `AF_INET` and `AF_INET6` on the same socket.

2. You can operate an IPv4 connection on an AF_INET6 socket using IPv4-mapped IPv6 addresses. In this case, both the source and destination must be IPv4-mapped addresses. The IPV6_V6ONLY option of the socket function disables the use of IPv4 on an AF_INET6 socket, which is needed to control whether IPv4 packets are received when AF_INET6 sockets are bound and listened on with a wildcard IP address. The IPV6_V6ONLY option also determines whether IPv4 packets are received with UDP and RAW sockets.
3. The IPv6 protocol (AF_INET6 and IPV6_V6ONLY) as well as AF_UNIX and AF_LOCAL are supported only by POSIX. For information, see the *OpenVOS POSIX.1 Reference Guide* (R502).

- The type argument indicates the protocol type associated with the socket. Specify one of the following values.
  - The SOCK_STREAM value associates the socket with TCP, which is a virtual-circuit protocol. This type of socket is also known as a connection-mode socket.
  - The SOCK_DGRAM value associates the socket with the User Datagram Protocol (UDP), which is a transport-level datagram protocol. This type of socket is also known as a connectionless-mode socket.
  - The SOCK_RAW value associates the socket with one of various network-level protocols (for example, Internet Control Message Protocol (ICMP), Internet Gateway Routing Protocol (IGRP), or Open Shortest Path First (OSPF)). Note that to use one of these protocols, you must build your own header.
- The prot argument indicates the protocol(s) that the socket uses. In general, IP uses this value to determine which socket to send incoming messages to. The protocol for a message is kept in a field in the ip header. Specify values as follows:
  - For sockets of the type SOCK_STREAM or SOCK_DGRAM, specify the value 0.
  - For sockets of the type SOCK_RAW, specify a value for the protocol used, as defined in the netinet/in.h header file. For example, specify the value IPPROTO_ICMP for ICMP. If you specify the value 0 or the value IPPROTO_RAW, no incoming messages are delivered to the socket. You can set a protocol value after the socket is created by using the bind function. For more information, see "Using Sockets of the Type SOCK_RAW" on page 2-19.

The socket function returns either a value called a socket descriptor or -1. A *socket descriptor* serves as a reference mechanism for a process. In subsequent operations, such as calls to the bind, listen, and send functions, the process can use the

descriptor to identify the socket that the operation pertains to. The socket descriptor is available only to the process that created the socket. If the returned value is `-1`, the `socket` function call was unsuccessful.

The following `socket` call creates a virtual-circuit or streams-type socket with the TCP protocol providing the underlying communications support.

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

The following `socket` call creates a datagram socket with the UDP protocol providing the underlying communications support.

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

Specifying IP Addresses and Port Numbers

A client process and a server process can communicate only if they can identify each other. They do this through the use of an IP address and port number.

Many STCP functions require an IP address and port number as input or produce them as output. These functions require that a process specify the address in a `sockaddr` data structure, which is defined by the header file `sys/socket.h`. The `sockaddr` structure follows.

```
struct sockaddr {
    unsigned short sa_family; /* Address family; only AF_INET */
    char     sa_data[14]; /* Address-family specific    */
};
```

The fields in the `sockaddr` structure are as follows:

- The `sa_family` field specifies the type of address family. STCP supports only the `AF_INET` family.
- The `sa_data` field contains the address in a protocol-dependent format.

An `AF_INET` address is specified by a 16-byte `sockaddr_in` structure containing four fields. This is a redefinition of the `sockaddr` structure for an `AF_INET` address. When you call a function requiring a pointer to a `sockaddr` structure and you have set up the required address in a `sockaddr_in` structure, you must cast the pointer to the `sockaddr_in` structure to a pointer to a `sockaddr` structure.

The sockaddr_in structure, which is defined by the header file netinet/in.h, is as follows.

```
struct sockaddr_in {
    short          sin_family; /* STCP supports only AF_INET */
    unsigned short sin_port;   /* 16-bit port number         */
    struct in_addr sin_addr;   /* 32-bit network ID/host ID  */
    char           sin_zero[8]; /* Not used                 */
};
```

The fields in the sockaddr_in structure are as follows:

- The first field, sin_family (bytes 1 and 2), indicates the address family, which must be AF_INET. TCP/IP supports **only** the AF_INET address family. For information about support for the AF_INET6 address family, see the *OpenVOS POSIX.1 Reference Guide* (R502).
- The second field, sin_port (bytes 3 and 4), indicates the port number, if one exists, for sockets of the type SOCK_STREAM and SOCK_DGRAM. For sockets of the type SOCK_RAW, the sin_port field must be 0.
- The third field, sin_addr (bytes 5 through 8), is an in_addr structure in which the network number and host number are stored as a bit string, using the format of one of the four address classes of IP addresses. IP addresses use a standard dot-notation form of 32 bits divided into four 8-bit fields, or octets, and each octet can range from 0 to 255. For additional information about and examples of address classes, see the *OpenVOS STREAMS TCP/IP Administrator's Guide* (R419).
- The fourth field, sin_zero (bytes 9 through 16), is unused.

The in_addr structure that stores the network number/host number is defined in the header file netinet/in.h. The in_addr structure follows.

```
struct in_addr {
  union {
            struct{unsigned char s_b1,s_b2,s_b3,s_b4;} S_un_b;
            struct{unsigned short s_w1,s_w2;} S_un_w;
            unsigned long S_addr;
  } S_un;
#define s_addr  S_un.S_addr
#define s_host  S_un.S_un_b.s_b2
#define s_net   S_un.S_un_b.s_b1
#define s_imp   S_un.S_un_w.s_w2
#define s_impno S_un.S_un_b.s_b4
#define s_lh    S_un.S_un.b.s_b3
};
```

Instead of using the preceding in_addr structure, you can use the following in_addr structure.

```
struct in_addr {
     unsigned long s_addr;
};
```

Binding an IP Address and Port Number to the Socket

A new socket has an IP address of 0.0.0.0 and a port number of 0. You use the bind function to assign a new IP address and port number.

The bind function has the following syntax.

```
int bind(int s, struct sockaddr *a, int al);
```

The bind function has three arguments.

- The s argument identifies the socket descriptor of the socket to which a socket address will be bound.
- The a argument is a pointer to a sockaddr structure defined previously by the process; it contains the address to be bound to the socket. (See "Specifying IP Addresses and Port Numbers" on page 2-3 for a detailed description of socket addresses and the sockaddr structure.)
- The al argument contains the length, in bytes, of the address. Specify a value of sizeof(struct sockaddr) for this argument.

For example, if your process created a socket with the socket descriptor sd, and defined a sockaddr structure named new_addr that contains the address and port number to be bound to the socket, you might issue the following call to bind the address and port number stored in new_addr to the socket.

```
struct sockaddr_in new_addr;
/* assign a value to sockaddr_in */
bind(sd, &new_addr, sizeof(new_addr));
```

A process can create and bind **one** unicast address per socket. A process can create and bind **one** multicast address per socket of the type SOCK_DGRAM (you cannot use IP multicast with TCP). To obtain the address bound to a socket, a process can call the getsockname function. See Chapter 5 for a description of the getsockname function.

You can only bind to an IP address that has been configured on one of the interfaces on the current module, or to the special address INADDR_ANY.

An application that calls the bind function to associate a socket address with a socket can simplify binding by specifying special values for the network number, host address, and port number components of a socket address.

- Specifying a network number and host address of INADDR_ANY associates the socket with any of the AF_INET host addresses available on the current module. For example, suppose that the addresses 192.52.249.1 and 192.52.50.200 are available on the module, and that the application binds the address INADDR_ANY,2000 to a socket (note that 2000 represents the port number). The socket could then receive incoming connection requests for either of the following addresses: 192.52.249.1,2000 or 192.52.50.200,2000.

  Note that an application can specify a network number and host address of INADDR_ANY even if the module has only one address available.

- Specifying a port number of 0 indicates that the application wants STCP to select an appropriate port number. While an application may specify a unique port number during binding, user applications occasionally do not know the values to use for the local port number. Consequently, when your client process sets sin_port to 0, STCP will assign an unused port number whose value is greater than 49,152. (Port numbers in the range 1 through 1023 are reserved for certain services such as FTP and TELNET. Note that a server process must use a known port number, while a client process can use any port number.)

If an application does not call the bind function, STCP automatically assigns an unused port number for SOCK_STREAM sockets when they issue a connect.

Communications between Processes

Data communications between two processes (that is, sending data to a peer or receiving data from a peer) proceed differently depending on the protocol type used. This section discusses the following topics related to data communications.

- ‘‘Using TCP and UDP Protocols’’ on page 2-6
- ‘‘Using Sockets of the Type SOCK_RAW’’ on page 2-19

Using TCP and UDP Protocols

Figure 2-1 presents the TCP/IP data communications model, which illustrates how the TCP and UDP protocols interact.

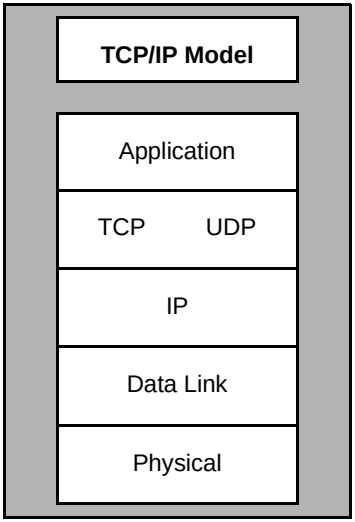

Figure 2-1. The TCP/IP Model

The components of Figure 2-1 interact as follows.

- The *physical layer* deals with physical media, connectors, and the signals that represent 0's and 1's.
- When sending data, the *data link layer* splits the data into frames to be sent to the physical layer. When receiving data, the data link layer accepts frames from the physical layer and decodes them into the indicated upper-layer packet.
- When sending data, the *IP layer* accepts the data from the TCP or UDP layer, fragments the data into multiple packets if it is too long, adds IP addressing, performs some error checking, and passes the data to the data link layer. When receiving data, the IP layer accepts packets from the data link layer and formats them back into a TCP or UDP segment, while buffering fragmented packets (if necessary) and performing some error checking.
- When sending data, the *TCP/UDP layer* accepts data from the application and formats it into segments by adding application addressing, then it passes the data to the IP layer. The TCP layer also resends any segments that were not acknowledged. When receiving data, the TCP/UDP layer accepts a segment from the IP layer, performs some error checking, and passes the data to the application. The TCP layer also sends acknowledgements for the data it receives, and it buffers segments that are out of order until the data can be sent to the application in the correct order. Note, however, that the UDP protocol does not provide sequence numbering (though it does provide error checking), so it is not as reliable as the TCP protocol. See ''TCP Communications'' on page 2-8 and ''UDP

Communications'' on page 2-17 for more information about the TCP and UDP protocols, respectively.

- The *application layer* performs the tasks of your application, which may include the following.
  - The application coordinates a connection between two computers.
  - The application converts files from one format to another if the server and client use different formats.

TCP Communications

The TCP protocol provides reliable, flow-controlled, duplexed (two-way) end-to-end data transmission. TCP uses `AF_INET` addresses with the number of a port on the host appended to each address (see ''Creating a Socket'' on page 2-1 for restrictions on using `AF_INET` addresses). TCP communications require port numbers.

TCP communications require that two processes be connected using a socket. The process that requests the connection is known as the *client*. The process that accepts the connection is known as the *server*.

Once a connection is established, a process can call the `getpeername` function to obtain the address of the peer process to which it is connected. The `getpeername` function is very similar to the `getsockname` function. Both functions are described in Chapter 5.

TCP does not preserve the message boundaries of the data sent between client and server applications. For example, if a client sends two messages, each with a length of 1 KB, the server might read them as a single 2 KB message. The client and server must agree upon how they will determine the beginning and end of transmitted messages.

You **cannot** use IP multicast with TCP or raw sockets. You can use IP multicast only with UDP sockets of the type `SOCK_DGRAM`. For more information about IP multicast, see the *OpenVOS STREAMS TCP/IP Administrator's Guide* (R419).

STCP supports a maximum of 32,767 TCP sockets per module, depending on system usage. The maximum number of open sockets per process is slightly less than 4096: the number of open ports per process is 4096, but the maximum number of open sockets is 4095 minus the number of ports used by the system.

The following sections describe the typical sequence of function calls that a process uses to establish and implement each side of a TCP connection.

- ''The Server Side of a TCP Connection'' on page 2-9
- ''The Client Side of a TCP Connection'' on page 2-14

The Server Side of a TCP Connection

Before accepting a TCP connection, a server process performs the following tasks.

- It uses the `socket` function to create a `SOCK_STREAM` socket, specifying the `AF_INET` address family.
- It uses the `bind` function to bind an IP address and port to a socket.
- It optionally sets the I/O mode as well as any desired socket options (Chapter 3 describes how to set I/O mode and socket options)

By using wildcard addressing, a listener can listen for connection requests from clients on all networks. In *wildcard addressing*, instead of binding its socket to an actual `AF_INET` address, the server binds it to the symbolic address `INADDR_ANY`, which represents an unspecified valid address. This address, which is a wildcard, matches any addresses specified in connection requests from multiple networks and multiple network interfaces.

The `listen` function has the following syntax.

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

The `listen` function has two arguments.

- The `s` argument identifies the socket descriptor of the socket whose status is to be changed from active to passive. (An *active* socket is one that is ready for use by a client; a *passive* socket is one that is ready for use by a server.) Note that a socket is neither active nor passive until the application takes further action. For a TCP connection, the server calls the `listen` function to place a socket in passive mode and prepare it to accept incoming connections.
- Requests for a server connection wait in a queue until the TCP connection is established and the server application issues an `accept` call. The `backlog` argument specifies the number of requests that can wait in the queue. (The maximum value is 1024.) If the queue is full when a connection request arrives, STCP refuses the connection and the client receives the error code `ECONNREFUSED`.

Suppose that the process that created the socket whose descriptor is `sd` wants to become a server on that socket, and that the queue of connection requests for the server will contain a maximum of three requests. In this case, the client issues the following call.

```
listen(sd, 3);
```

The socket `sd` is now defined as a server.

The server application must issue an `accept` function call. If no connection is established, the function either waits if the socket is in blocking mode or returns the

error EWOULDBLOCK if the socket is in nonblocking mode. You may want to change a socket that is in blocking mode to nonblocking mode and use the select or select_with_events function to check the socket's status. See the descriptions of these functions in Chapter 5 for more information.

If a connection is requested, the accept function creates a new socket on the server to complete the connection. The new socket has the same characteristics as the original socket. The server uses this new socket to transmit data to or from the client, using the same functions as a client would to read or write data.

In STCP, every socket created by an application corresponds to an OpenVOS port that attaches automatically to the STREAMS protocol device (for example, the STCP device #stcp.m1). The protocol device is a *clonable STREAMS device*, which means that each socket is associated with an OpenVOS port attached to a cloned device (for example, #stcp.m1_234, where 234 is a cloned device number).

To transfer a socket from one process to another, use the receive_socket() and transfer_socket() functions. The transfer_socket() function returns the device name associated with the socket. The transferring process must then communicate the device name to the receiving process. The receiving process then uses the receive_socket function to open the device and acquire a new socket. Because transfer_socket() does not close the socket as part of the operation, the caller must explicitly close the socket. In order to do so, the process that calls receive_socket must notify the process that calls transfer_socket after it has received the socket path name and opened the socket using receive_socket. The process that calls transfer_socket must wait for that notification before it explicitly closes its socket. Note that in this case, multiple socket descriptors refer to one socket. The socket descriptors can be in the same process or different processes.

You can transfer a socket from a parent process to a child process using the fork function. When the parent process forks a child process, the two processes are attached to the same device and share the same socket. Each process has its own socket descriptor for the socket that is shared. After the fork completes, the parent usually explicitly closes any sockets that will be used by the child, and the child explicitly closes any sockets that will be used by the parent. For information on the fork function, see the *OpenVOS POSIX.1: Conformance Guide* (R217M) and the *OpenVOS POSIX.1 Reference Guide* (R502).

Figure 2-2 illustrates how to use the fork function.

```
#define _POSIX_C_SOURCE 200112L

#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <ctype.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define bzero(s, len)             memset((char *)(s), 0, len)

int errno;

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

{
    short port_no;

    struct sockaddr_in serv_addr;
    struct sockaddr_in cli_addr;
    int    socklen_t clilen;

    int socks0, socks1;
    pid_t iPID;

    int recvBytes, sendBytes;
    char buffer [200];
    short i;

    if (argc == 2)
       port_no = atoi (argv [1]);
    else

    {
       printf ("\nUsage: stcp_fork <port_number>\n");
       exit (-1);
    }
    printf ("stcp_fork %d\d\d", port_no);

    if ((socks0 = socket (AF_INET, SOCK_STREAM, 0)) < 0)
    {
       perror ("stcp_fork: can't open stream socket");
       exit (errno);
    }
```

(Continued on next page)

```
    bzero ( (char *) &serv_addr, sizeof (serv_addr));
    serv_addr.sin_family        = AF_INET;
    serv_addr.sin_addr.s_addr   = htonl (INADDR_ANY);
    serv_addr.sin_port          = htons (port_no);

    if (bind (socks0, (struct sockaddr *) &serv_addr, sizeof (serv_addr)) < 0)
    {
       perror ("stcp_fork: can't bind local address");
       exit (errno);
    }

    listen (socks0, 5);

    accept_again:
    socks1 = accept (socks0, (struct sockaddr *) &cli_addr, &clilen);

    iPID = fork ();

    if (iPID == -1)
    {
       printf ("Error %d during fork - exiting");
       exit (errno);
    }

    if (iPID > 0) /* This is the parent process */
    {
       close (socks1); /* close the accepted socket */
       goto accept_again;
    }

    /* must be the child process - do all the real work */

    close (socks0); /* no need to keep the listen socket open */

    /* do all of the real work here */

}
```

Figure 2-2. Sample Program Using the `fork` Function

The Client Side of a TCP Connection

Before opening a TCP connection, a client process performs the following tasks.

- use the `socket` function to create a `SOCK_STREAM` socket, specifying the `AF_INET` address family
- optionally set the input/output (I/O) mode as well as any desired socket options (Chapter 3 describes how to set I/O mode and socket options)

A client process is not required to call the bind function to bind an address to the socket. Instead, the STCP software automatically selects and binds an address to the socket when the client requests a connection with a server.

The client calls the connect function to request an active connection with a server. The connect function has the following syntax.

```
int connect(int s, struct sockaddr *a, int al);
```

The connect function has three arguments.

- The s argument identifies the socket descriptor of an active socket to be used for the connection.
- The a argument specifies the address (in a sockaddr structure) of the peer.
- The al argument specifies the length of the address of the a argument (specified as sizeof(struct sockaddr_in)).

Suppose that the process has created the socket whose descriptor is sd and wants to connect with a peer whose address is contained in a sockaddr structure named peer_addr. The process calls the connect function as follows:

```
struct sockaddr_in peer_addr;

connect(sd, &peer_addr, sizeof(peer_addr));
```

Once the server accepts the connection, the client can begin data transmission. To send data, the client typically calls the send function. You can also use the write function. To determine which function is appropriate for your application, see the description of the send function in Chapter 5 and the description of the write function in the*OpenVOS Standard C Reference Manual* (R363).

To receive data, the client typically calls the recv function. You can also use the read function. To determine which function is appropriate for your application, see the description of the recv function in Chapter 5 and the description of the read function in the*OpenVOS Standard C Reference Manual* (R363).

The response from a function can be delayed if the socket is in blocking mode (the default). If a socket is in blocking mode and no data is available to be read, a data-reading function does not return until data is available and has been read. If a socket is in blocking mode, a send (or write) call will block if the local buffer does not have sufficient space to hold all the data. If the write buffer has sufficient space, the write call copies the data to the buffer and returns. For a socket in nonblocking mode, a partial send of data occurs, and the sending function returns. In this case, the client would need to send the remaining data by calling the sending function again. See ‘‘Setting and Checking I/O Mode’’ in Chapter 3 for more information about blocking mode.

When data transmission is complete, the client should terminate the connection with the `close` function.

Figure 2-3 illustrates a sample client-connection setup in blocking mode (the default mode). In the example, three vertical dots indicate that lines of code have been omitted.

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

main (int argc, char *argv[])   /* Args: IP address, port */
{
     int   sd;                  /* Socket descriptor */
     struct sockaddr_in  sin;   /* Structure of IP address & port number */

     /* Get remote IP address and remote port from command line.
        Fill in sockaddr_in structure of peer address. */

     sin.sin_family = AF_INET;
     sin.sin_addr.s_addr = inet_addr (argv[1]);
     sin.sin_port = htons (atoi (argv[2]));

     /* Open a TCP socket and exit program on failure. */

     if ((sd = socket (AF_INET, SOCK_STREAM, 0)) < 0)
     {
        perror ("opening socket");
        exit (errno);
     }

     /* Socket is in BLOCKING mode by default. */

     if (connect (sd, (struct sockaddr *) &sin, sizeof (sin)) < 0)
     {
        perror ("Error return from connect");
        exit (errno);
     }
     /* socket sd is successfully connected and can be used */
     .
     .
     .
}
```

Figure 2-3. Sample Client-Connection Setup in Blocking Mode

Figure 2-4 illustrates a sample client-connection setup in nonblocking mode.

```
#include <stdlib.h>
#include <sys/socket.h>
#include <prototypes/inet_proto.h>
#include <errno.h>
#include <poll.h>
#include <fcntl.h>
#include <streams_utilities.h>              /* For ioctl(), fcntl() */

main (int argc, char *argv[])      /* Args: IP address, port */
{
     int       sd;                 /* Socket descriptor */
     int       non_blocking = 1;   /* Enable nonblocking mode */
     struct    pollfd client[1];   /* For polling */
     int       flags;

     struct sockaddr_in  sin;      /* Structure of IP address & port number */

     /* Get remote IP address and remote port from command line.
        Fill in sockaddr_in structure of peer address. */

     sin.sin_family = AF_INET;
     sin.sin_addr.s_addr = inet_addr (argv[1]);
     sin.sin_port = htons (atoi (argv[2]));

     /* Open a TCP socket and exit program on failure. */

     if ((sd = socket (AF_INET, SOCK_STREAM, 0)) < 0)
     {
        perror ("opening socket");
        exit (errno);
     }

     /* Place the socket in NONBLOCKING mode. */

     if ( (flags = fcntl(sd, F_GETFL, 0)) < 0)
     {
        perror ("Error return from fcntl F_GETFL");
        exit (errno);
     }

     if (fcntl(sd, F_SETFL, (flags | O_NDELAY)) < 0)
     {
        perror ("Error return from fcntl F_SETFL");
        exit (errno);
     }
```

(Continued on next page)

```
      while (connect (sd, (struct sockaddr *) &sin, sizeof (sin)) < 0)
      {
         /* errno set to EISCONN after asynchronous connect has completed */

         if (errno == EISCONN)
            break;

         /* errno set to EINPROGRESS when initiating asynchronous connect
            and to EALREADY when asynchronous connect already active */

         if (errno != EINPROGRESS && errno != EALREADY)
         {
            perror ("Error return from connect");
            exit (errno);
         }

         client[0].fd = sd;
         client[0].events = POLLOUT;
         client[0].revents = 0;
         if (poll(client, 1, -1) <= 0)           /* wait forever */
         {
            perror ("Error from poll");
            exit (errno);
         }

         if (client[0].revents & (POLLHUP | POLLERR))
         {
            perror ("Socket error from poll");
            exit (errno);
         }

         /* POLLOUT return means connect MAY have completed; loop back
            and reattempt connect to make sure. */
      }

      /* socket sd is successfully connected and can be used
         (after one or more connect attempts) */
      .
      .
      .
}
```

Figure 2-4. Sample Client-Connection Setup in Nonblocking Mode

UDP Communications

This section describes the typical sequence of function calls used in UDP communications.

The UDP protocol for transport-level datagram communications provides simple but unreliable data transmission. The UDP protocol does not report lost or out-of-sequence packets. The UDP protocol uses AF_INET addresses with the number of a port on the host appended to each address. Port numbers are required in UDP communications (see ‘‘Binding an IP Address and Port Number to the Socket’’ on page 2-5 for more information about assigning port numbers).

Use the following functions for tasks that a process must perform before beginning UDP communications.

- You must use the socket function to create a SOCK_DGRAM socket, specifying the AF_INET address family (see ‘‘Creating a Socket’’ on page 2-1 for restrictions on using AF_INET addresses).
- You can optionally use the bind function to bind an IP address and port number to the socket if you are also using the sendto function—See ‘‘Binding UDP Sockets’’ on page 2-18 for more information about binding UDP sockets.
- You can optionally set the I/O mode as well as any desired socket options (Chapter 3 describes how to set I/O mode and socket options).

The process can use the connect function to establish a destination for future data packets. If it does, the process can use the write and read or send and recv functions. Note that, in this case, establishing a connection with a peer process does not provide the reliability of TCP communications; instead, it simply means that the application does not need to supply the address with each send call.

You can use the write function with connected SOCK_DGRAM sockets. You can use the read function with connected or unconnected SOCK_DGRAM sockets.

Instead of using the connect function to establish a destination for future data packets, the process can begin to send or receive data immediately. In this case, a process must specify the address of the peer process whenever it sends data. When a process receives data, the sender’s address is passed back. Typically, the process uses the sendto and recvfrom functions. However, it can also use the sendmsg and recvmsg functions. To determine which function is appropriate for your application, see their descriptions in Chapter 5.

After your program issues the send request, the program is notified of the occurrence of asynchronous ICMP errors via return values only when the socket is connected. Note that a socket is considered connected after the connect function is called, unless the specified address is INADDR_ANY or the address family is AF_UNSPEC. You can

disconnect a previously connected SOCK_DGRAM socket by using one of these two values.

You can use UDP communications to broadcast data packets (that is, to transmit them to all hosts on a network). The UDP protocol is particularly suitable for broadcast communications because transmission reliability is not a major concern.

An application sends a broadcast message by calling a write function and specifying a broadcast address for the recipient. The *broadcast address* is a special address that is automatically bound by all hosts on the network. Broadcast messages must be sent using the UDP protocol (a socket of type SOCK_DGRAM). Before sending a broadcast message, an application must call the setsockopt function to specify the value SO_BROADCAST for the optname argument.

In the AF_INET address family, an address is interpreted as a broadcast address if the host-address portion is a bit string composed entirely of ones.

See ‘‘Specifying IP Addresses and Port Numbers’’ on page 2-3 for more information about address classes.

NOTE

Broadcast messages can adversely affect network performance; use them only when necessary.

Binding UDP Sockets

UDP sockets can bind to any of the following:

- the symbolic address INADDR_ANY, which allows you to send multicast packets, receive all unicast packets addressed to the bound port, and receive any multicast packets that were explicitly registered
- a multicast address, which allows you to receive only multicast packets for the specified multicast group (no unicast packets), if the specified multicast group has been joined
- a local IP address, which allows you to send multicast packets and receive unicast packets

If the current module is *multihomed* (that is, the module has multiple physical interfaces or has multiple addresses assigned to one physical interface), you must specify the local IP unicast address as the *default system interface* (that is, the interface over which multicast messages are sent if no interface was specified in the setsockopt function’s IP_MULTICAST_IF option). To set the default system interface, issue the route add command, specifying the multicast address as the destination and the interface as the gateway.

OpenVOS supports IP multicast level 2, which allows you to send and receive IP multicast messages. The maximum number of IP multicast addresses to which a socket can bind is 256.

NOTE

When you use IP multicast, you should build reliability at the application level, as UDP communications may not be reliable.

For more information about multihoming, the `route` command, and IP multicast, see the *OpenVOS STREAMS TCP/IP Administrator's Guide* (R419). For more information about the `setsockopt` function, see its description in Chapter 5.

Using Sockets of the Type `SOCK_RAW`

When you create a socket of the type `SOCK_RAW`, you must associate the socket with a protocol. The protocol for a message is kept in a field in the IP header. In general, the protocol value determines which socket should be sent the incoming messages. To associate the socket with a protocol, use the `prot` argument of the `socket` function when you create the socket, or use the `bind` function after the socket is created.

- For the `prot` argument of the `socket` function, specify a value as defined in the `netinet/in.h` header file. For example, specify the value `IPPROTO_ICMP` for ICMP. If you specify the value `0` or the value `IPPROTO_RAW`, no messages are delivered to the socket.
- You can set a protocol value after the socket is created by using the `bind` function. To do so, specify the protocol value in the `sin_port` field of the `sockaddr_in` structure that is passed with the `bind` function. Specifying `IPPROTO_IP` (the value `0`) for `sin_port` enables the socket to receive messages for all protocols.

NOTE

The value `0` for the `prot` argument of the `socket` function and the value `0` for the `sin_port` field of the `sockaddr_in` structure that is passed with the `bind` function do **not** have the same meaning. The value `0` for the `prot` argument of the `socket` function is identical to using the value `IPPROTO_RAW` (`255`). Thus, you cannot request the behavior of `IPPROTO_IP` by using only the `socket` function. You must also use the `bind` function.

You can use only the `sendto` and `recvfrom` functions to communicate with `SOCK_RAW` sockets; you cannot use the `connect` function.

You cannot use IP multicast with raw sockets.

A process must be privileged to call sockets with the type SOCK_RAW. In this case, *privileged user* refers to a user who has write access to the streams_user access control list (ACL). With this definition, a privileged user is unrelated to an OpenVOS privileged user. (For information on the streams_user ACL, see the *OpenVOS STREAMS TCP/IP Administrator's Guide* (R419).)

If you want to issue ICMP requests with SOCK_RAW sockets, you must specify a unique ID (typically, the process ID), which is used to direct the reply back to the requesting socket.

Shutting Down and Closing a Socket

The following sections describe how to shut down and close a socket.

- "The shutdown Function" on page 2-20
- "The Partial Close Procedure" on page 2-21
- "The close Function" on page 2-21

The shutdown Function

The shutdown function shuts down a TCP socket for subsequent read operations, write operations, or both read and write operations. It does not release the IP address and port number bound to the socket or any resources used by the socket. The shutdown function has the following syntax.

```
int shutdown(int s, int rw);
```

The shutdown function has two arguments.

- The s argument identifies the socket descriptor of the socket to be shut down. The rw argument prevents subsequent read or write operations, as follows:
  - The value SHUT_RD prevents subsequent read operations.
  - The value SHUT_WR prevents subsequent write operations.
  - The value SHUT_RDWR prevents subsequent read **and** write operations.
- Subsequent attempts to read and/or write to the socket (as indicated by the value of rw) return an error code. Once a socket is shut down, it cannot be reactivated.

For example, the following shutdown call shuts down write operations for the socket whose descriptor is sd.

```
shutdown(sd, SHUT_WR);
```

Note that you do **not** need to call shutdown before calling close.

See Chapter 5 for more information about the `shutdown` function.

The Partial Close Procedure

When an application finishes using a connection, it can call the `close` function to end the connection and deallocate the socket. However, closing a connection often is not simple because TCP allows two-way communication. Thus, coordination between the client and server is necessary when closing a connection.

For example, consider a client that repeatedly issues requests to which a server responds. Since the server does not know whether the client will issue more requests, the server cannot close the connection. However, the client may not know whether all data has arrived from the server, even though it knows that it has no more requests to send.

TCP addresses this problem by providing the ability for one end of a connection to terminate its output, while still receiving data from the other end. This capability is called a *partial close*.

To perform a partial close, your application must call the `shutdown` function with a second argument of `SHUT_WR`, rather than calling the `close` function. Your application must contain `shutdown(sd, SHUT_WR)`, which sends an end-of-file signal to the other end of the connection when the application has finished sending data. The application should also continue to receive data from the other end of the connection until the other end sends the application an end-of-file signal. End-of-file is indicated when a `read` or `recv` function returns zero.

For example, when a client finishes sending requests, it can use the `shutdown` function to indicate that it has no more data to send, while not deallocating the socket. The server application then receives an end-of-file signal. At that point, the server knows that no more requests will arrive, and it can close the connection after sending its last response.

Note that many applications terminate both directions of the connection by calling the `close` function, which is described in ‘‘The close Function’’ on page 2-21. These applications must have provisions in the protocol they use to ensure that no data is lost. The partial close procedure is the only way that the TCP protocol can ensure no data is lost.

The `close` Function

The `close` function causes a file or socket to be closed, which destroys the socket. For sockets of the type `SOCK_STREAM`, `close` initiates or completes a termination protocol between the connected peers, which results in an orderly termination of communication. For sockets of the type `SOCK_DGRAM`, `close` immediately makes the socket available for reuse.

The close function has the following syntax.

```
int close(int s);
```

The s argument identifies the socket descriptor of the socket that is to be closed.

If successful, close returns the value 0. If unsuccessful, it returns the value -1. The function sets errno to EBADF if the specified socket descriptor is invalid or has already been closed. It sets errno to EWOULDBLOCK when SO_LINGER has been specified with a non-zero time value for the socket and the time has expired before all data was sent and acknowledged by the peer.

After close closes the socket, no further operations can be performed using socket descriptor s. If the process issuing close is the last process referencing the socket, and if the socket is of the type SOCK_STREAM, the normal termination protocol is initiated (that is, a FIN segment is sent to the peer following all existing data in the send buffer). If unread data exists, the connection does not close normally; instead, it is reset.

If some data is untransmitted, the close function can block for a linger interval until all data is transmitted, under the following conditions:

- The socket is of the type SOCK_STREAM.
- Nonblocking mode has not been set.

  NOTE

  The SO_LINGER option has no effect on a socket that is set to nonblocking mode; a delay never occurs for such a socket.

- The SO_LINGER option has been set with a nonzero value for the time interval.

Under these same conditions, the close function can also block for a linger interval if the remote system is not responding and does not acknowledge (ACK) the caller's close request (with the FIN segment). This can occur even if all data has been transmitted.

Control typically returns immediately to the caller of close. For sockets of the type SOCK_STREAM, the connection to the peer remains active until the termination protocol is complete. The system attempts to deliver any remaining data to the peer, and the socket (that is, the connection) cannot be reused until after the system receives final acknowledgement of the close from the peer.

NOTE

The successful completion of close does not guarantee that resources have been freed. For example, if the

socket is associated with a TCP connection, the address of this socket is considered in use until the TCP connection is completely closed. Under normal conditions, the connection is closed soon after the `close` call is issued. However, under certain conditions, this may take longer (possibly several minutes after the return from `close`). For more information about reusing addresses, see the description of the `SO_REUSEADDR` option in "The SO_REUSEADDR Option" on page 3-8 and in the `setsockopt` function description in Chapter 5.

Note that the socket descriptor passed as the `s` argument of `close` is **different from** the socket that represents the connection.

- The socket descriptor is a per-process entity. A process can have up to 4096 sockets and/or file descriptors opened simultaneously. When `close` is complete and control returns to the caller, the socket descriptor can no longer be used and is eventually reassigned.
- A socket that represents the connection is a system entity that contains a unique pairing of two address/port combinations. When `close` is complete and control returns to the caller, a socket remains in use throughout the termination while it transitions through various states (you can display these stages using the `netstat` command).

For TCP `(SOCK_STREAM)` connections, the `close` function initiates or completes the default termination protocol between the connected peers, which results in an orderly termination of communication. The initiator of the termination sends a close request (with the `FIN` segment) to its peer, and then it enters the `FIN_WAIT_1` state until it receives an acknowledgement (`ACK`). The peer sends the `ACK` when it recognizes the `close` request. The initiator of the termination then enters the `FIN_WAIT_2` state. At some point, the application running on the peer recognizes that the socket has been closed (for example, when it receives `0` in response to `recv`), and initiates a close, which sends `FIN` back to the initiator of the termination. The initiator enters the `TIME_WAIT` state and remains in that state for a duration of twice the maximum segment life (MSL), which is two minutes, by default. This additional waiting time is intended to prevent this connection from receiving stray messages erroneously directed to it by a new connection of the same address/port pair. Figure 2-5 shows the default connection termination and associated transition states of the `close` function.

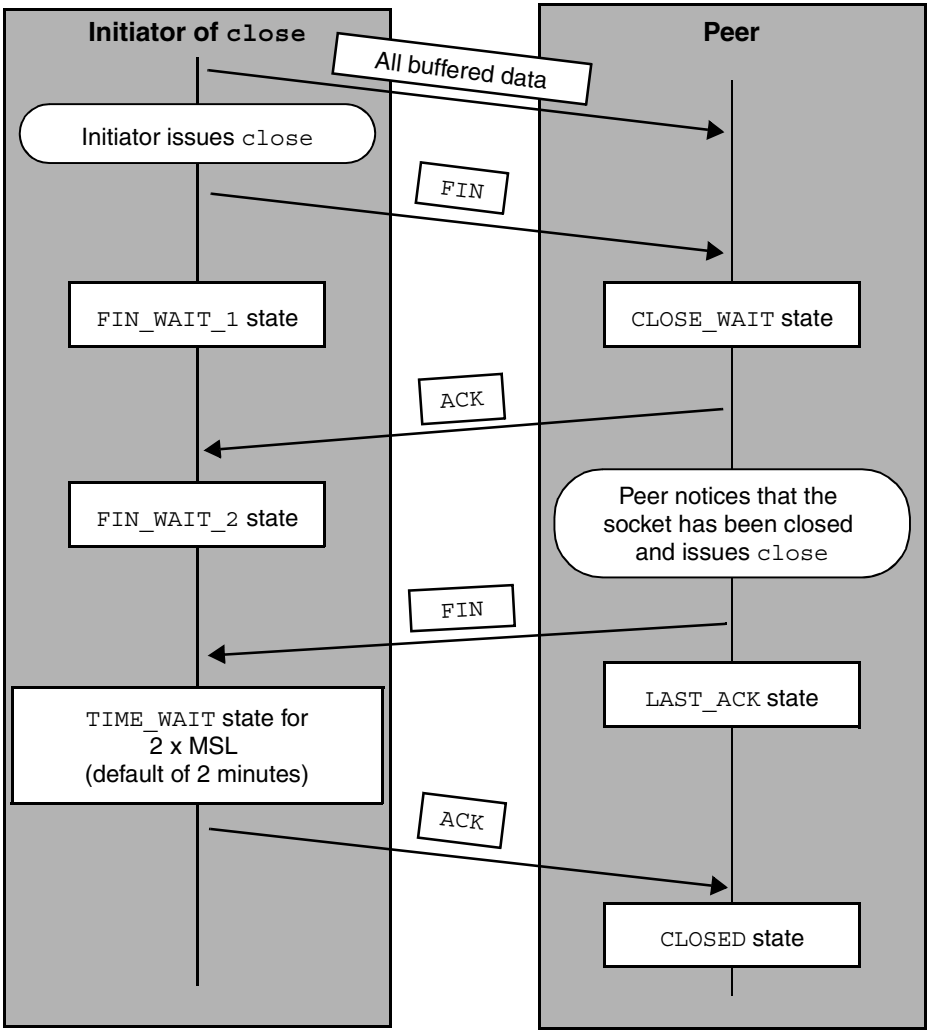

Figure 2-5. The Default Connection Termination of the `close` Function

This default behavior of close can result in problems. The application calling close cannot know if the peer TCP has acknowledged the receipt of all data sent unless the application has provisions in its own protocols to ensure the receipt of data sent.

The best way to avoid data loss is to use the shutdown function with SHUT_WR (see "The shutdown Function" on page 2-20); however, you can make some of the problems with close less likely to occur by using the SO_LINGER option or by using other wait-related values, as the following sections describe.

- "Using the SO_LINGER Option to Solve close Problems" on page 2-25
- "Using Other Wait-Related Values to Solve close Problems" on page 2-26

For more information about the close function, see the *OpenVOS Standard C Reference Manual* (R363).

Using the SO_LINGER Option to Solve close Problems

The SO_LINGER option enables you to set a time-out period called a *linger interval*, which solves problems with close, at least partially.

The SO_LINGER option uses the linger structure, which allows you to specify a value for the l_onoff variable to indicate on-off and a value for the l_linger variable to indicate a time value. To set values for the l_onoff and l_linger variables, you use the setsockopt function. The linger structure, which follows, is defined in the header file sys/socket.h.

```
struct linger {
int l_onoff;  /* 0=off; nonzero=on */
int l_linger; /* Linger time */
};
```

When l_onoff equals zero, the SO_LINGER option is turned off. When l_onoff has a nonzero value, the value of l_linger causes the following behavior:

- When l_linger has a nonzero value, control is not returned to the caller of close either until the time limit expires or until the peer has acknowledged all sent data and the FIN segment (that is, until the transition to the FIN_WAIT_2 state occurs). If the time limit expires, close returns -1 with errno set to EWOULDBLOCK (or the equivalent value EAGAIN); otherwise, it returns 0.
- When l_linger equals zero, the peer is sent a reset request (RST), which aborts the connection immediately. In this case, the send buffer and receive buffer are discarded, which avoids the normal close state transition. The connection can be reused immediately because the TIME_WAIT delay does not occur. However, a new connection could receive a message intended for the old one.

Table 2-1 lists and describes values of the l_onoff and l_linger variables.

Table 2-1. Values for Variables in the `linger` Structure

`l_onoff` Value	`l_linger` Value	Description
`0`	(any value is ignored)	The `SO_LINGER` option is turned off. The `close` function returns immediately after initiating the TCP termination protocol by sending a `FIN` segment, which indicates that the process has finished sending data.
nonzero	`0`	TCP aborts the connection when it is closed. Any remaining data is deleted and a reset/restart (`RST`) control bit is sent immediately to the peer process, which instructs the receiver to delete the connection without further interaction. This avoids the normal TCP termination protocol and thus eliminates the `TIME_WAIT` state.
nonzero	nonzero	The process lingers when the socket is closed. If any data remains to be sent, the process waits until all the data is sent or the linger time has expired. The application should check the return value from `close` because if the linger time expires before the remaining data is sent, `close` returns `-1` and sets `errno` to `EWOULDBLOCK`. TCP continues the normal termination protocol.

Using Other Wait-Related Values to Solve `close` Problems

You can use other wait-related values to solve problems with `close`. For example, you can adjust the value of the `FIN_WAIT2` state by using the `set_stcp_param` request of the `analyze_system` subsystem to set the value of the `finwait2` parameter. By default, the `FIN_WAIT2` state has no time limit. TCP waits until a peer process responds to an active close request by issuing a corresponding close, regardless of the time required. Note, however, that when an OpenVOS process terminates, OpenVOS automatically closes any of the process' sockets that are still open. Also, note that setting `FIN_WAIT2` to low values can cause connections to be reset. You would typically either not set it or leave it at a very large number (for example, 15 minutes). You can temporarily reduce this value to clean up sockets if a large number have been orphaned by another machine crashing or by network disconnections.

Chapter 3
Using the Supporting STCP Functions

Many functions perform tasks supporting data transmission. This chapter describes the supporting functions in the STCP socket library. It contains the following sections.

Chapter 2 discusses the `socket`, `bind`, `connect`, `listen`, `accept`, `write`, `read`, `shutdown`, and `close` functions, which are the basis for all programs using the STCP API.

Setting and Checking I/O Mode

A process performs I/O in one of two modes: blocking or nonblocking. In *blocking mode*, a call to a TCP/IP function does not return until the call completes or an error occurs. If the call completes, it returns the appropriate value. If an error occurs, the function generally returns a value of `-1` and sets the `errno` global variable to indicate the type of error that occurred. (Exceptions are noted in the "Return Values" section of each function description in Chapter 5.) A socket is created in blocking mode by default.

In *nonblocking mode*, a call to a TCP/IP function returns immediately. If successful, the function returns the appropriate value. If the call could not be completed on the specified socket, the function returns a value of `-1` and sets the `errno` global variable to the error code for `EWOULDBLOCK`. The process then waits until it is able to retry the call. In the meantime, it can perform other processing, send or receive data on other

sockets, check the progress of function calls, and so forth. If the call is unsuccessful for other reasons, the function generally returns a value of `-1` and sets the `errno` global variable to the appropriate error code.

For example, suppose that a process tries to receive data at a socket, but no data is available. If the socket is in blocking mode, the function waits until data is available. If the socket is in nonblocking mode, the function returns an error. Similarly, suppose that a process tries to send new data when the peer process has not yet finished reading the previous data sent. If the socket is in blocking mode, it may block if the TCP window is closed. If the socket is in nonblocking mode, the function returns an error.

To specify a socket's I/O mode, call the `fcntl` function. (You can also set the I/O mode using the `ioctl` function; for information, see the `ioctl` function description.) The `fcntl` function has the following syntax.

```
fcntl(fd, cmd, arg);
```

The `fcntl` function has three arguments.

- The `fd` argument identifies a file descriptor. In the case of STCP, `fd` identifies the socket descriptor.
- The `cmd` argument specifies the command to be executed by `fcntl`, either `F_SETFL` to set the file-descriptor flag or `F_GETFL` to return the current flag settings.
- The `arg` argument specifies the value of the file-descriptor flag.

Suppose that a process will be using a socket argument whose descriptor is `sd` for I/O. To designate `sd` as a **blocking** socket, the process calls the `fcntl` function as follows:

```
int flags;

/* Set Socket blocking */

if ( (flags = fcntl(sd, F_GETFL, 0)) < 0)
     perror("F_GETFL error");
flags &= ~O_NDELAY;
if (fcntl(sd, F_SETFL, flags) < 0)
     perror("F_SETFL error");
```

To designate sd as a **nonblocking** socket, you need to turn on the O_NDELAY bit, as follows:

```
int flags;

/* Set Socket nonblocking */

if ( (flags = fcntl(sd, F_GETFL, 0)) < 0)
     perror("F_GETFL error");
flags |= O_NDELAY;
if (fcntl(sd, F_SETFL, flags) < 0)
     perror("F_SETFL error");
```

For more information about the fcntl function, see the *OpenVOS Communications Software: STREAMS Programmer's Guide* (R306).

You can use the select or poll function to determine if a socket is available for reading or writing before calling a function as well as to determine whether a call can be retried when a socket is operating in nonblocking mode. For more information about the select function, see the *OpenVOS Standard C Reference Manual* (R363) and the *OpenVOS POSIX.1 Reference Guide* (R502). For more information about the poll function, see the OpenVOS Subroutines manuals.

Setting and Checking Socket Functions

STCP provides two functions, setsockopt and getsockopt, that allow a process to set and check the I/O characteristics and other options that control socket behavior. (You can also use the ioctl function to set I/O characteristics that control socket behavior; for information, see the description of the ioctl function.)

The setsockopt function sets values for the options that exist at socket level (application level) or the IP level. The setsockopt function has the following syntax.

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

The setsockopt function has the following arguments.

- The s argument identifies a socket descriptor.
- The level argument indicates the protocol level at which the option exists. This value can be SOL_SOCKET, to indicate a "socket-level" (application-level) option, or IPPROTO_IP, to indicate an IP-level option.
- The optname argument indicates the option to be set. Table 5-9 and Table 5-10 list the values allowed for each option.
- The optval argument contains the value to be set for the specified option.
- The optlen argument contains the length, in bytes, of optval.

To determine the current value of an option set with `setsockopt`, call the `getsockopt` function. The `getsockopt` function has the following syntax.

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

The `getsockopt` function has the following arguments.

- The `s` argument identifies a socket descriptor.
- The `level` argument indicates the protocol level at which the option exists. This value can be `SOL_SOCKET`, to indicate a "socket-level" (application-level) option, or `IPPROTO_IP`, to indicate an IP-level option.
- The `optname` value indicates the option whose value is to be checked. Table 5-3 and Table 5-4 list the values allowed for each option.
- The `optval` argument contains the value to be set for the specified option.
- The `optlen` argument contains the length, in bytes, of `optval`.

The following sections describe frequently used socket-level options.

- "The SO_KEEPALIVE Option" on page 3-4
- "The SO_LINGER Option" on page 3-8
- "The SO_REUSEADDR Option" on page 3-8

The `SO_KEEPALIVE` Option

You can use the `SO_KEEPALIVE` option to test whether the connection is still functioning; however, keepalive functionality does not confirm that a remote program is running correctly. You can use keepalive functionality to allow a program to clear connections that have failed (for example, when a peer crashes), and to free the socket for another use.

NOTES

1. Keepalive functionality is not intended to continually monitor a connection. It is not a program heartbeat test for connectivity because the time interval for sending the first probe packet (the default value is two hours) is too long, and reducing the time interval to something more useful (for example, five minutes) may cause connections to break during transient internet failures, or result in other problems with network load or performance.
2. Use the _VOS_KEEPALIVE option in POSIX.1-compliant applications. See the *OpenVOS POSIX.1 Reference Guide* (R502) for details.

3. RFC 1122 recommends that the time interval for sending the first keepalive probe packet be two hours.

The following sections provide information about using keepalive functionality.

Keepalive Functionality and an STCP interface

Keepalive functionality is enabled on the STCP interface by default when an administrator adds an interface using the `ifconfig` command. If an administrator has added an interface using the `ifconfig` command with the `-no_kalive` argument, the `SO_KEEPALIVE` option has no effect.

To determine if keepalive is enabled, issue the `ifconfig` command. If it is enabled, `KEEPALIVE` appears in the command output, as in the following example.

```
ifconfig #sdlmux2

%s#sdlmux2: <UP, BROADCAST, RUNNING, NOFORWARDBROADCAST,
KEEPALIVE> 164.152.77.6 netmask 0xfffffe00 broadcast
164.152.77.255
```

If `KEEPALIVE` does not appear, you cannot use keepalive functionality on connections using this STCP interface.

Parameters of the `SO_KEEPALIVE` Option

Various parameters control the behavior of the `SO_KEEPALIVE` option. Before running a program that creates sockets, an administrator can set values for these parameters. STCP assigns these values when a program creates a socket; it does not assign new values to sockets that are already allocated.

An administrator can list and set values of these parameters by using requests of the `analyze_system` command: the `list_stcp_params` request lists values and the `set_stcp_param` request sets a value. Table 3-1 lists and describes the parameters.

Table 3-1. `keepalive` Parameters

Parameter	Description
`keepalive_time`	Sets the time interval, in minutes, when STCP sends the first probe packet after the last packet is received from the remote peer. If, for example, `keepalive_time` is set to the value `150`, STCP sends the first probe packet 2 and 1/2 hours (150 minutes) after receiving the last packet from the remote peer. The range of values is `1` to `480`; the default value is `120` minutes.
`keepalive_tries`	Sets the number of probe packets that `SO_KEEPALIVE` sends. The range of values is `1` to `25`; the default value is `9`.
`check_if_dead`	Sets the time interval, in seconds, between probe packets. The range of values is `30` to `360`; the default value is `75` seconds.

For example, the following `analyze_system` requests enable STCP to send a total of five probe packets, where the first probe packet is sent 60 minutes after the last packet is received from the remote peer and subsequent probe packets are sent at two-minute intervals.

```
as: set_stcp_param keepalive_time 60
Changing keepalive time interval (keepalive_time)
      from 120 min to 60 min

as: set_stcp_param keepalive_tries 5
Changing keepalive tries (keepalive_tries)
      from 9 to 5

as: set_stcp_param check_if_dead 120
Changing keepalive check dead time (check_if_dead)
      from 75 sec to 120 sec
```

Writing a Program That Contains the `SO_KEEPALIVE` Option

An individual program can set the `SO_KEEPALIVE` option with the `optname` argument of the `setsockopt` function, as follows:

```
struct linger keepalive;
keepalive.1_onoff = 1;
keepalive.1_linger = 0; /* use default of 2 hours */
int setsockopt(s, SOL_SOCKET, SO_KEEPALIVE, (char *)
               &keepalive, sizeof(keepalive));
```

The `SO_KEEPALIVE` and `SO_LINGER` options both use the `linger` structure, which allows you to specify a value for the `l_onoff` variable to indicate on-off and a value for the `l_linger` variable to indicate a time value. If the value of the `l_onoff` variable is non-zero, then the value of the `l_linger` variable defines the number of seconds

that STCP waits before it sends the first probe for this socket (after the last packet is received from its peer). If you want the socket to use a non-default value for the timing of the first probe (for example, 1 hour), the program must pass a `linger` structure and set the `keepalive.l_linger` field to `3600`.

Figure 3-1 illustrates how to set the `KEEPALIVE` option on a socket connection. This sample code displays `Keepalive set ON to system default 2 hr 0 min` when it runs. In the example, three vertical dots indicate that lines of code have been omitted.

```
#include <stdlib.h>
#include <sys/socket.h>
#include <prototypes/inet_proto.h>
#include <errno.h>
#include <streams_utilities.h>     /* For ioctl(), fcntl() */

main ()
{
     int    sd;                    /* Socket descriptor */
     struct linger keepalive;
     int    keepalive_min;
     int    on_off;
     int    len;

     /* Open a TCP socket and exit program on failure. */

     if ((sd = socket (AF_INET, SOCK_STREAM, 0)) < 0)
     {
        perror ("opening socket");
        exit (errno);
     }

     /* Verify keepalive is off, as we'd expect for a new socket. We
        don't care about the value, so we will use the int argument
        rather than the struct; we could have used either. */

     len = sizeof(int);
     if (getsockopt(sd, SOL_SOCKET, SO_KEEPALIVE, (char *) &on_off, &len) < 0)
     {
        perror ("getting on-off keepalive value");
        exit (errno);
     }
     if (on_off)
        printf("Keepalive unexpectedly ON\n");

     /* Set keepalive on, using system default keepalive time */
```

(Continued on next page)

```
      on_off = 1;
      if (setsockopt(sd, SOL_SOCKET, SO_KEEPALIVE, (char *) &on_off, len) < 0)
      {
         perror ("setting keepalive value");
         exit (errno);
      }

      /* Report the time interval which was set, i.e., the system default */

      len = sizeof(struct linger);
      if (getsockopt(sd, SOL_SOCKET, SO_KEEPALIVE, (char *) &keepalive,
                                                                  &len) < 0)
      {
         perror ("getting full keepalive value");
         exit (errno);
      }

      if (! keepalive.l_onoff)
         printf("Keepalive unexpectedly OFF\n");
      else
      {
         keepalive_min = keepalive.l_linger / 60;  /* convert to minutes */
         printf("Keepalive set ON to system default %d hr %d min\n",
                 keepalive_min / 60, keepalive_min % 60);
      }

      /* socket now can be used to connect or to bind and accept.  Any
         new sockets created as the result of accept will inherit this
         keepalive value. */
      .
      .
      .
}
```

Figure 3-1. Sample Code That Uses `SO_KEEPALIVE`

The `SO_LINGER` Option

You can use the `SO_LINGER` option to enable lingering and to set a time-out period called a *linger interval*. When a process calls `close` for a socket, data that was previously written to the socket may not have been delivered to the connected peer process. The `SO_LINGER` option provides you with some control over that data. For information on the `SO_LINGER` option, see ‘‘The close Function’’ on page 2-21.

The `SO_REUSEADDR` Option

Generally, STCP rejects attempts to bind a socket to a local address and port number if the socket that previously used that address and port still exists. However, some applications, such as FTP, require that connections always use the same local port. To

override the default port selection algorithm, you can call the setsockopt function, specifying the SO_REUSEADDR option, prior to address binding, as follows:

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

In the preceding example, the address associated with socket s may be assigned to other sockets. This does not violate the TCP/IP address uniqueness requirement since the software still checks prior to accepting a connection request that any other sockets with the same local address and port do not have the same foreign address and port. If an association already exists, errno is set to EADDRINUSE.

The SO_REUSEADDR option may cause the protocol (TCP or UDP) to ignore existing sockets when binding; otherwise, the bind call returns EADDRINUSE.

You can control the functionality of the SO_REUSEADDR option by using the tcp_reuseaddr_action and udp_reuseaddr_action parameters of the set_stcp_param request of the analyze_system subsystem. The values of the parameters are as follows:

- unsafe—Multiple servers are allowed to bind the same address-port combination as long as the first bound socket has set the SO_REUSEADDR option. This is the legacy behavior.
- safe—Multiple servers are allowed to bind the same address-port combination as long as all sockets have set the SO_REUSEADDR option and all processes have the same session ID. In this case, the assumption is that the bind callers are cooperating to share the port.

  The *session ID* is a unique number that identifies a group of processes that were started by a single user. Typically, a session is a login session, but some servers also start sessions. A session ID cannot be set directly, but the setsid() function turns a process into the first process in a new session, and then sessions forked from that first process are in the same session, with the assumption that the group of processes is cooperating.

  Duplicate bind operations to UDP multicast addresses are allowed as long as all of the sockets set the SO_REUSEADDR option.

NOTE

Connections are always sent to the first socket that was bound to the target address/port. STCP will not load balance among multiple listening sockets. However, STCP distributes incoming connections to multiple listening processes if they are listening on the same

socket that has been shared using `fork` or `transfer_socket`.

Obtaining Information about Hosts

A network must maintain information about hosts on the system—their names, aliases (alternative names), and addresses—and make that information available to users and applications so that communication endpoints can be specified precisely.

This section discusses the following topics related to hosts.

- ‘‘The hosts File’’ on page 3-10
- ‘‘The Host Information-Retrieval Functions’’ on page 3-11

The `hosts` File

A network administrator can establish a `hosts` database file containing information about host addresses, symbolic names, and aliases in the directory `(master_disk)>system>stcp`. A sample `hosts` file follows.

```
127.0.0.1     loopback     lo    localhost #required
172.16.2.27   accnt         m1
172.16.2.3    sales         m2
172.16.9.44   xt
```

In the preceding example, each line describes a different host. On each line, the first item is the host address, in dot-notation form (for example, `172.16.2.27`). (Dot notation is described in ‘‘Translating Addresses’’ on page 3-16.) The second item is the host’s symbolic name (for example, `sales`). Any remaining items **not** preceded by a number sign (`#`) are aliases (for example, `lo` and `localhost`). An item preceded by a number sign is a comment (for example, `#required`).

If the `resolv.conf` file is absent from the `(master_disk)>system>stcp` directory, STCP queries a DNS name server located on the local module via the loopback address, `127.0.0.1`, before it uses the `hosts` file.

NOTE

The `nsswitch.conf` file can change how STCP performs various Internet lookup functions such as name resolution. For information on the `nsswitch.conf` file, see *OpenVOS STREAMS TCP/IP Administrator’s Guide* (R419).

The Host Information-Retrieval Functions

The *host information-retrieval functions* allow an application to obtain host information from either a name server or a `hosts` database file. The application does not need to know which method the network administrator used to store host information. The host information-retrieval functions will either query a name server or search the `hosts` database file.

The host information-retrieval functions are as follows:

- `sethostent`
- `gethostbyaddr`
- `gethostbyname`
- `gethostent`
- `endhostent`

The application uses the `sethostent` function first. This function either specifies the protocol type that will be used if communicating with the name server, or opens the `hosts` database file if accessing database files.

The application then uses the `gethostbyaddr`, `gethostbyname`, or `gethostent` function. Each of these functions returns a pointer to a `hostent` structure containing information about a host. In `gethostbyaddr`, the host is specified by its address; in `gethostbyname`, the host is specified by its name. The `gethostent` function returns a `hostent` structure for the next entry in the `hosts` file. The `hostent` structure is defined (in the header file `netdb.h`) as follows:

```
struct hostent {
     char     *h_name;
     char     **h_aliases;
     int      h_addrtype;
     int      h_length;
     char     **h_addr_list;
#define h_addr h_addr_list[0]
};
```

The `hostent` structure has the following fields. (Note that the `hostent` structure is static data that will be overwritten by subsequent calls.)

- The `h_name` field contains the host's name.
- The `h_aliases` field contains an array of aliases, each terminated by a zero.
- The `h_addrtype` field contains the value `2` to indicate the `AF_INET` address family.
- The `h_length` field contains the length, in bytes, of the host address.

- The `h_addr_list` field contains a list of IP addresses for the host, in network byte order.

Finally, the application uses the `endhostent` function to close any connections to the name server that were opened, if communicating with the name server, or to close the `hosts` file, if accessing database files.

See the *OpenVOS STREAMS TCP/IP Administrator's Guide* (R419) for information about how an administrator sets up and manages either a name server or a `hosts` database file. See Chapter 5 for detailed descriptions of the host information-retrieval functions.

Obtaining Information about Networks

A network must maintain information about the networks known to it and make that information available to users and applications. An application obtains the network number used to construct an IP address from the `networks` database file.

STCP provides this information in the `networks` database file (located in the directory `(master_disk)>system>stcp`). The file contains a one-line entry for each known network. Each entry consists of the network's name, its network number, and any aliases.

A sample `networks` file follows.

```
loopback-net        127       software-loopback-net
agency-net          120       agnet
sales-net           150.1
company-net         192.20.1  cent  compnet
```

In the preceding example, each line describes a different network. The first item on each line is the network's symbolic name (for example, `loopback-net`). The second item is the network number, in dot-notation form (for example, `127`). (Dot notation is described in ''Translating Addresses'' on page 3-16.) Any remaining items are aliases (for example, `software-loopback-net`). See the *OpenVOS STREAMS TCP/IP Administrator's Guide* (R419) for detailed information about the `networks` database file.

Network Information-Retrieval Functions

The *network information-retrieval functions* allow an application to access the information in the `networks` database file. The network information-retrieval functions are as follows:

- `setnetent`
- `getnetbyaddr`

- getnetbyname
- getnetent
- endnetent

The application uses the setnetent function first. This function opens the networks database file and resets the file marker to the beginning of the file. The function can also be used to specify that the file remain open after calls to getnetbyname and getnetbyaddr.

The application then uses the getnetbyaddr, getnetbyname, or getnetent function. Each of these functions returns a pointer to a netent structure containing information about a specified network. In getnetbyaddr, the network is specified by its network number; in getnetbyname, the network is specified by its name. The getnetent function returns a netent structure for the next entry in the networks database file. By calling getnetent repeatedly, an application can search the file.

The netent structure is defined (in the header file netdb.h) as follows:

```
struct netent {
     char               *n_name;
     char               **n_aliases;
     int                n_addrtype;
     unsigned long      n_net;
};
```

The netent structure has the following fields.

- The n_name field contains the network's name.
- The n_aliases field contains an array of aliases, each terminated by a zero.
- The n_addrtype field contains the value 2 to indicate the AF_INET address family.
- The n_net field contains the network number, in host byte order. (Note that the netent structure is static data that will be overwritten by subsequent calls.)

Finally, the application uses the endnetent function to close the networks database file.

See the *OpenVOS STREAMS TCP/IP Administrator's Guide* (R419) for information about how an administrator sets up and manages the networks database file. See Chapter 5 for detailed descriptions of the network information-retrieval functions.

Obtaining Information about Protocols

A network must maintain information about the protocols known to it and make that information available to users and applications.

STCP provides this information in the `protocols` database file (located in the directory `(master_disk)>system>stcp`). A sample `protocols` file follows.

```
ip        0    IP        # internet protocol, pseudo protocol
                           number
icmp       1     ICMP      # internet control message protocol
tcp        6     TCP       # transmission control protocol
udp        17    UDP       # user datagram protocol
```

The file contains a one-line entry for each known protocol. Each entry consists of the protocol's name, its protocol number, and any aliases. See the *OpenVOS STREAMS TCP/IP Administrator's Guide* (R419) for detailed information about the `protocols` database file.

Protocol Information-Retrieval Functions

The *protocol information-retrieval functions* allow an application to access the information in the `protocols` database file. The protocol information-retrieval functions are as follows:

- `setprotoent`
- `getprotobyname`
- `getprotobynumber`
- `getprotoent`
- `endprotoent`

The application uses the `setprotoent` function to open the `protocols` database file and reset the file marker to the beginning of the file. The application then uses the `getprotobyname`, `getprotobynumber`, or `getprotoent` function, each of which reads the `protocols` database file and returns a pointer to a `protoent` structure containing information about a protocol. (Note that the `protoent` structure is static data that will be overwritten by subsequent calls.) The application finally uses the `endprotoent` function to close the `protocols` database file.

See Chapter 5 for detailed descriptions of the protocol information-retrieval functions.

Obtaining Information about Network Services

A network must maintain information about the user services known to it (that is, the application-layer utilities such as FTP and TELNET) and make that information available to users and applications.

STCP provides this information in the `services` database file (located in the directory `(master_disk)>system>stcp`). A sample `services` file follows.

```
ftpdata         20/tcp
ftp             21/tcp          ftpd
telnet          23/tcp          telnetd
smtp            25/tcp
bootps          67/udp          bootpd
bootpc          68/udp          bootp
```

The file contains a one-line entry for each known service. Each entry consists of the service's name, its port number, the name of the associated protocol, and any aliases. See the *OpenVOS STREAMS TCP/IP Administrator's Guide* (R419) for detailed information about the `services` database file.

Service Information-Retrieval Functions

The *service information-retrieval functions* allow an application to access the information in the `services` database file. The service information-retrieval functions are as follows:

- `setservent`
- `getservbyname`
- `getservbyport`
- `getservent`
- `endservent`

The application uses the `setservent` function to open the `services` database file and reset the file marker to the beginning of the file. The application then uses the `getservbyname`, `getservbyport`, or `getservent` function, each of which reads the `services` database file and returns a pointer to a `servent` structure containing information about a service. (Note that the `servent` structure is static data that will be overwritten by subsequent calls.) Finally, the application uses the `endservent` function to close the `services` database file.

See Chapter 5 for detailed descriptions of the service information-retrieval functions.

Translating Addresses

Host addresses (or their components) are usually obtained from name servers or databases, where they are stored as ASCII character-string representations of the equivalent dot-notation form.

In the *dot-notation form* of an address, the network number and host address bytes of a bit-string address are represented as their decimal, octal, or hexadecimal equivalents, with each byte separated by a period. If a port number is used, a comma and the port number are appended. The address family is omitted since dot notation is used only for `AF_INET` addresses, and the eight unused bytes are also omitted.

Consider the following network number/host number.

```
11000000001101001111100100000001
```

The following table shows different dot-notation representations of the preceding network number/host number.

Address	Type of Representation
`192.52.249.1`	Decimal-base dot notation
`192.52.249.1,1234`	Decimal-base dot notation, including a port number (represented by `,1234`)
`0300.064.0371.01`	Octal-base dot notation (the leading `0` in each component identifies the component as octal)
`0xC0.0x34.0xF9.0x01`	Hexadecimal-base dot notation (the leading `0x` in each component identifies the component as hexadecimal)
`0xC034F901`	Hexadecimal-base dot notation, shown in the *extended format*, in which periods are omitted and the `0x` prefix appears only at the beginning of the address

The example host address shown earlier in this section is a Class C address: the three high-order bits in the original bit string are `110`. Therefore, the first three bytes of each version represent the network number: 192.52.249 (decimal), 0300.064.0371 (octal), and 0xC0.0x34.0xF9 (hexadecimal). The remaining byte represents the host address: `1` (decimal), `01` (octal), and `0x01` (hexadecimal).

Because addresses can be represented as either bit strings (in a `sockaddr` structure) or ASCII character-string representations of the equivalent dot-notation form, an application must be able to convert addresses between these forms. The following list describes the functions that STCP provides to handle these conversions. Some of these functions convert addresses between bit strings and dot notation; others perform

related activities, such as extracting the network number portion of a dot-notation address.

- The `inet_addr` function converts a character string representing the dot-notation form of an `AF_INET` address to an unsigned long integer containing the address.
- The `inet_lnaof` function extracts a host number from a host address.
- The `inet_makeaddr` function combines a network number and a host portion to produce a host address.
- The `inet_netof` function extracts a network number from a character string representing the dot-notation form of a host address, and returns an equivalent unsigned long integer.
- The `inet_network` function extracts the network-number portion of a character string representing the dot-notation form of an `AF_INET` address, and returns an unsigned long integer representing the network number.
- The `inet_ntoa` function converts a host address in an `in_addr` structure to a character string representing the dot-notation form of the address.

See Chapter 5 for detailed descriptions of these functions.

Accessing the Name Server

As described in ‘‘Obtaining Information about Hosts’’ on page 3-10, host information (name and address) is available either from a name server on the network or in database files. An application calls the function `gethostbyname`, which itself calls internal functions to query either the name server on the network or the appropriate database, depending on whether the configuration file `resolv.conf` exists. Since an application does not call these internal functions directly, this manual does not describe them.

If STCP attempts to query a DNS server, and if the server(s) identified in `resolv.conf` does not respond, a timeout occurs after 28 to 32 seconds. Waiting will be terminated if an ICMP error indicates that the server is not running the name service or the server cannot be reached.

Ensuring Correct Byte Order for 16-Bit and 32-Bit Data Types

Not all hardware treats data types in the same way. Often, differences exist in how hardware handles short and long (16-bit and 32-bit) data types. Networks often handle the high-order byte (the most significant byte) first, but a given host may handle the high-order byte or the low-order (least significant) byte first, depending on the host type.

If the host does not pass 16-bit or 32-bit data items to the network in network byte order, data will be processed incorrectly. To prevent this, STCP provides a group of

byte-swapping functions that convert the data between host byte order and network byte order. Stratus **strongly** recommends that an application always uses these functions when sending or receiving data, regardless of whether or not the host byte order matches the network byte order. If the byte orders are different, the data is converted appropriately. If the byte orders are the same, the data is unaffected. In either case, the application is guaranteed portability to some other host that may not use the same host byte order.

You do not need to know the type order of the host on which your application will run to use the following byte-swapping functions.

- The `htons` function receives a 16-bit data item that is in host byte order and converts it to a 16-bit data item that is in network byte order.
- The `ntohs` function receives a 16-bit data item that is in network byte order and converts it to a 16-bit data item that is in host byte order.
- The `htonl` function receives a 32-bit data item that is in host byte order and converts it to a 32-bit data item that is in network byte order.
- The `ntohl` function receives a 32-bit data item that is network byte order and converts it to a 32-bit data item that is in host byte order.

For example, consider a process that intends to send a 32-bit data item to a peer process. To ensure that the data item is in network byte order, the sending process calls the `htonl` function before sending the data, as follows:

```
netlong = htonl(hostlong);
```

In the preceding example, the `hostlong` argument is an unsigned long integer that contains data in host byte order. The function converts the data to network byte order and returns it to the variable `netlong`, which is also an unsigned long integer.

The peer process, after receiving the data, calls the `ntohl` function to ensure that it is in host byte order, as follows:

```
hostlong = ntohl(netlong);
```

In the preceding example, the `netlong` argument is an unsigned long integer that contains data in network byte order. The function converts the data to host byte order and returns it to the variable `hostlong`, which is also an unsigned long integer.

The `htons` and `ntohs` functions are used in the same manner for conversions of 16-bit data items.

See Chapter 5 for detailed descriptions of these functions.

Chapter 4
Programming Considerations

This chapter explains how to compile and bind an STCP application. It also contains recommendations on programming issues that are of concern when developing and testing applications. It contains the following sections.

NOTES

1. STCP supports only applications written in the C language (see Chapter 1). Stratus **strongly recommends** that you use the OpenVOS Standard C compiler, which is ANSI C- and POSIX.1-compatible, to compile your applications. See the *OpenVOS Standard C Reference Manual* (R363) and the *OpenVOS Standard C User’s Guide* (R364) for more information about OpenVOS Standard C.
2. Stratus recommends that you run the `check_posix` command if your system is running an OpenVOS STCP application. This command checks that the current module’s configuration meets constraints imposed by the OpenVOS POSIX.1 implementation. For more information about the `check_posix` command, see the *OpenVOS POSIX.1 Reference Guide* (R502).

Compiling and Binding an Application

You must compile and bind an application into an executable program module (a file with the suffix `.pm`) before you can execute it. You must also ensure that the compiler and binder search required libraries. The following sections describe how to perform these tasks.

NOTE

If you are porting an OS TCP/IP application to STCP, you **must** rebind the application with the STCP compatibility library in order for it to run with STCP. In some cases, you must rewrite and recompile the application. See the *OpenVOS STREAMS TCP/IP Migration Guide* (R418) for more information about how to port OS TCP/IP applications to STCP.

Adding Library Search Paths

When compiling and binding an application, you must ensure that the compiler and binder search required libraries. To do so, use one of the following methods to add the library path names to the library search paths of the process that you use to compile and bind the program.

- Add the `add_library_path` *`pathname`* command to the `start_up.cm` file for your process.
- Add the `add_default_library_path` *`pathname`* command to the `module_start_up.cm` file.
- Execute the `add_library_path` commands in your current process from the command line.

See the *OpenVOS Commands Reference Manual* (R098) for more information about the `add_library_path` and `add_default_library_path` commands.

Compiling an Application

To compile an STCP application, issue the OpenVOS Standard C compile command `cc`, specifying the source module and any pertinent compilation options.

An example of a `cc` command follows.

```
cc stcp_server_1 -u
```

If the compilation is successful, the compiler produces an object module (a file with the suffix `.obj`) for your application. The *OpenVOS Commands Reference Manual* (R098) and the *OpenVOS Standard C User's Guide* (R364) document the `cc` command.

For information about compiling an application that is POSIX.1-compliant, see ''Creating Source Code That Is POSIX.1- and ANSI C-Compliant'' on page 4-12.

Binding an Application

After you compile your application, you must bind the resulting object modules before executing the program. To bind an application, issue the `bind` command, specifying the application name and any bind options. This command produces a program module.

When binding an application, you must ensure that the binder searches the OpenVOS Standard C object library **before** it searches the standard object library. You must set the object search paths of the process that is used to bind the program as follows:

```
(current_dir)
>system>c_object_library
>system>object_library
```

Any other necessary object search paths, such as those for layered products or application object libraries, must appear before `>system>c_object_library`.

If you are using the STCP compatibility library to convert OS TCP/IP programs to STCP, add the following path name **before** the path names listed above (for complete information on using the STCP compatibility library, see the *OpenVOS STREAMS TCP/IP Migration Guide* (R418)).

```
>system>stcp>object_library>complib
```

If you are creating a POSIX.1-compliant application, add the following path name before the `>system>c_object_library` path name. (For additional information about creating a POSIX.1-compliant application, see ''Creating Source Code That Is POSIX.1- and ANSI C-Compliant'' on page 4-12.)

```
>system>posix_object_library
```

You can specify these libraries at bind time, or you can add these search paths using one of the two methods for adding search paths described in ''Adding Library Search Paths'' on page 4-2. To specify these libraries at bind time use either the `-search` argument of the `bind` command or a binder control file. The following example shows

how to use the `-search` argument. These examples assume that your object search paths are set to the OpenVOS default values.

```
bind stcp_server_1 -search >system>c_object_library
```

A binder control file should include the object-library information. You then specify the name of the file using the `-control` argument of the `bind` command. The following file, `simple_stcp_accept.bind`, is an example of a binder control file.

```
name: simple_stcp_accept;

modules: 'simple_stcp_accept';
search:  >system>c_object_library;
end;
```

See the *OpenVOS Commands Reference Manual* (R098) and the *OpenVOS Standard C User's Guide* (R364) for more information about the `bind` command and binder control files.

Header Files

The STCP software includes C header files (that is, include files) that contain information required by STCP applications. These files reside in the `(master_disk)>system>include_library` directory or its subdirectories, and have names in the form *`file_name`*`.h`. Table 4-1 lists and describes the files.

Table 4-1. The `(master_disk)>system>include_library` Files *(Page 1 of 2)*

File	Description
`bsd.h`	Provides BSD application compatibility.
`errno.h`	Defines symbolic names for various OpenVOS errors. To simplify the porting of applications code, the file also equates the conventional `Exxx` symbolic names (such as `EPERM`) for UNIX TCP/IP errors with the names of the corresponding OpenVOS errors. You can use the definitions in this file to simplify error-handling routines in an application. To use these definitions, an application must contain an `#include` statement specifying the `errno.h` file. Note: The global variable `errno` is set with an STCP-specific error value whenever an STCP function returns a value that is less than zero. If an STCP function does not return a negative value, the value of the global variable errno is not modified.
`error_codes.incl.c`	Defines OpenVOS error codes.

Table 4-1. The (master_disk)>system>include_library Files *(Page 2 of 2)*

File	Description
netdb.h	Defines the functions and structures (for example, hostent, netent, protoent, and servent) used to map host, network, protocol, and service names into numeric values.
streamio.h	Provides STREAMS system call compatibility.
sysexits.h	Provides exit status codes for system programs.

The subdirectories of (master_disk)>system>include_library that contain information required by STCP applications are as follows:

- “The >arpa Directory” on page 4-5
- “The >net Directory” on page 4-6
- “The >netinet Directory” on page 4-6
- “The >sys Directory” on page 4-7
- “The >compat Directory” on page 4-7

To include the STCP header files that exist in subdirectories in source files, reference header files in sub-directories as partial pathnames, using / as a separator, as in the following example:

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

This method is compatible with the -u option for the ANSI C compiler and with the method POSIX specifies for including these headers in application programs.

Certain STCP header files also exist in the compatibility library. For information, see “The >compat Directory” on page 4-7.

The >arpa Directory

The (master_disk)>system>include_library>arpa directory contains header files related to STCP application-layer services. (The arpa directory contains no subdirectories.) Table 4-2 describes these files.

Table 4-2. The (master_disk)>system>include_library>arpa Files *(Page 1 of 2)*

File	Description
ftp.h	Provides definitions for the ftpd daemon process.
inet.h	Provides network address definitions.

Table 4-2. The (master_disk)>system>include_library>arpa Files *(Page 2 of 2)*

File	Description
nameser.h	Provides definitions for Domain Name Service (DNS) name servers, which resolve host names and addresses.
telnet.h	Provides definitions for TELNET.

The >net Directory

The (master_disk)>system>include_library>net directory contains header files that support internal TCP/IP interfaces. Table 4-3 describes these files. (The net directory contains no subdirectories.)

Table 4-3. The (master_disk)>system>include_library>net Files

File	Description
if.h	Contains structures that define a network interface, thus providing a packet transport mechanism.
if_arp.h	Provides Address Resolution Protocol (ARP) structure definitions.
if_flags.h	Contains definitions for IP-related flags.

The >netinet Directory

The (master_disk)>system>include_library>netinet directory contains header files that support the Internet protocols, IP, TCP, and UDP. Table 4-4 describes these files. (The netinet directory contains no subdirectories.)

Table 4-4. The (master_disk)>system>include_library>netinet Files *(Page 1 of 2)*

File	Description
in.h	Defines the constants and structures (for example, in_addr and sockaddr_in) used when specifying Internet addresses and making control requests on TCP and UDP endpoints.
in_systm.h	Contains miscellaneous internetwork definitions for the kernel.
in_var.h	Contains UNIX-compatible data structures.
ip.h	Contains definitions for byte order (low- or high-address byte significance).
ip_icmp.h	Contains Interface Control Message Protocol (ICMP) definitions.
ip_var.h	Contains an overlay for the IP header used by TCP and UDP.
tcp.h	Provides TCP definitions.

Table 4-4. The (master_disk)>system>include_library>netinet Files *(Page 2 of 2)*

File	Description
tcpip.h	Provides TCP/IP definitions.
udp.h	Provides UDP definitions.
udp_var.h	Provides definitions for UDP kernel structures and variables.

The >sys Directory

The (master_disk)>system>include_library>sys directory contains header files that define the internal socket interface code. Table 4-5 describes the files in the >sys directory.

Table 4-5. The (master_disk)>system>include_library>sys Files

File	Description
bsd_time.h	Contains UNIX-compatible time data structures.
debug.h	Provides compatibility with some programs written for other operating systems.
socket.h	Defines constants and data structures for socket-related concepts such as socket options and types, and address families. For example, it defines the sockaddr structure (used to store IP addresses and port numbers), which is required by the bind, connect, and accept functions.

The >compat Directory

The (master_disk)>system>stcp>include_library>compat directory contains header files that provide compatibility between the STCP software and applications developed using the OS TCP/IP software. Use this directory only if you are converting programs from OS TCP/IP to STCP. For complete information on using this directory, see *OpenVOS STREAMS TCP/IP Migration Guide* (R418).

Table 4-6 describes these files. (The compat directory contains no subdirectories.)

Table 4-6. The (master_disk)>system>stcp>include_library>compat Files *(Page 1 of 2)*

File	Description
ifreq.h	Empty file. Exists for compatibility with OS TCP/IP.

Table 4-6. The `(master_disk)>system>stcp>include_library>compat` Files *(Page 2 of 2)*

File	Description
`ostcp_to_stcp.h`	Contains macros mapping OS TCP/IP functions to STCP functions. Use the STCP compatibility library instead of the functionality provided by this file. For more information about the compatibility library, refer to the *OpenVOS STREAMS TCP/IP Migration Guide* (R418). This file will be removed in a future release of this product.
`tcp_errno.h`	Includes the header file `errno.h`. Exists for compatibility with OS TCP/IP, which used its own error codes.
`tcp_socket.h`	Includes the header files `netinet/in.h`, `netdb.h`, `sys/types.h`, and `sys/socket.h`, which define constants and data structures, as well as the header files `inet_proto.h` and `ostcp_to_stcp.h`, which define the STCP function calls.
`tcp_types.h`	Provides compatibility with the OS TCP/IP file of the same name.

To include these STCP header files in source files, add the directory `(master_disk)>system>stcp>include_library` to the include search paths, and reference header files in sub-directories as partial pathnames, using `/` as a separator, as in the following example:

```
#include <compat/ostcp_to_stcp.h >
```

This method is compatible with the `-u` option for the ANSI C compiler and with the method POSIX specifies for including these headers in application programs. In addition, the `(master_disk)>system>stcp>include_library` directory has appropriate links to support this method.

Application Responsibilities

The TCP/IP protocols are responsible for establishing connections between applications and for delivering data from one application to another. However, the application also has a number of responsibilities, which the following sections describe.

- ‘‘Connection Issues’’ on page 4-9
- ‘‘Data Delivery and Record Boundaries’’ on page 4-11
- ‘‘Security’’ on page 4-12

Connection Issues

A client process (a process requesting a connection) must know the IP address and port number of the socket that the server process is using before issuing the connect call.

A server process (a process accepting connection requests) receives from the `accept` function the IP address and port number of the socket whose connection request it accepts.

If either process needs to obtain this information after the connection has been established, it can call the `getpeername` function to return the IP address and port number of the socket on the other end of the connection.

The following sections provide additional information related to connection issues.

- "Using connect" on page 4-9
- "Using accept" on page 4-10

Using `connect`

An application must be designed to handle failed connections or to retry unsuccessful connection requests. A failed connection does not necessarily mean that the process with which the application wanted to connect does not exist. Connections can fail for many reasons, including network problems.

You can establish asynchronous connections with the `connect` function. Processes initiating STCP connections can be interrupted at any time during the connection, whether the socket is in default blocking mode or in nonblocking mode.

When you use blocking mode, each connection request waits until it receives an acknowledgment or rejection (possibly from a remote location). By using nonblocking mode, you can reduce the time required to complete a series of socket connections. A process can initiate a connection for each socket and then poll the sockets. The process then receives a notification when each socket is connected. Meanwhile, the process can proceed with the remaining connection requests, thereby reducing the time required to complete all connections.

To use nonblocking mode, call the `fcntl` function with the value of *`cmd`* set to `F_SETFL` and *`arg`* set to `O_NDELAY` (or `O_NONBLOCK`) after a socket has been created. (Note that the STCP `O_NDELAY` argument is equivalent to the POSIX `O_NONBLOCK` flag.) If the `connect` function attempts another connection that is not acknowledged immediately (usually when the target is not local), the function returns `-1` and sets `errno` to `EINPROGRESS`. Make note of the socket file descriptors and use the `select` or `poll` function to wait for the connections to complete (for example, by calling the `poll` function with `POLLOUT` as the requested poll event).

Some programs may use nonblocking mode with OpenVOS subroutines (for example, `s$wait_event`) and events rather than standard C functions (for example, `poll` and `select`). With these programs, OpenVOS always returns the error `e$caller_must_wait(1277)` (similar to the STCP `EINPROGRESS` error code) one extra time even after an event was notified. So, the program must call the subroutine a second time.

In releases prior to VOS Release 14.6.1, a call to `connect` always blocked until the connection was established or failed, even if the socket was set to nonblocking mode. Now, when you rebind, any program that calls `connect` for a socket in nonblocking mode must be prepared to handle a return value of `-1` with `errno` set to `EINPROGRESS`. The program must either continue to attempt the connection or continue polling (with the `poll` or `select` functions), waiting for the connection to be established.

Applications in which sockets were set to nonblocking mode, but that did not expect to see an `EINPROGRESS` error, may have worked in prior releases but will fail in VOS Release 14.6.1 or later. This failure only occurs if these applications are re-bound.

NOTE

You can determine if this potential problem exists in your source code by executing the POSIX `grep` command on the source code and searching for lines with a string match to `O_NDELAY` or `O_NONBLOCK` used with either the POSIX `fcntl` or `open` functions. Then, check that the code can handle the return value `-1`, with `errno` set to `EINPROGRESS`. Alternatively, you could use the OpenVOS command `display -match` (for example, `display * -match NDELAY`) to accomplish the same thing.

To disable nonblocking behavior in newly-bound modules on a per-system basis and to specify that connections should always block, use the `set_stcp_param` request of the `analyze_system` subsystem to set the `nonblk_connects` parameter to `off`, as in the following example.

```
set_stcp_param nonblk_connects off
```

Using `accept`

A server process that has called the `accept` function cannot refuse an incoming connection request. When the call completes, the TCP/IP protocols assume that the process has accepted a pending request and that the connection has been established. If the server process does not want to communicate with that client process, it must call the `close` function to terminate the connection, as described in Chapter 2.

The sequence of requests and responses that STCP uses to acknowledge client SYN requests is different from the sequence used by many other TCP implementations. In a typical TCP implementation, the server immediately acknowledges a client's SYN request with a SYN/ACK response for any socket that is listening (that is, a socket for which the server has called the listen function). With STCP, the server acknowledges the client's SYN request with a SYN/ACK response only after the server calls the accept function. This sequence, called *lazy accept*, results in the following behavior.

- A client that expects a quick response may need to wait longer to connect, since the response with STCP comes after the server has issued the accept function.

  To solve this problem, change the client code to use nonblocking mode. This enables the client to initiate and then poll all required connections, thereby gaining control as each connection is complete. By using nonblocking mode, the client does not need to wait for each server response.

- In STCP, if a server places a socket in nonblocking mode before it issues the accept function, the server may gain control after calling accept but before the client has completed the three-way handshake (that is, before the client has sent the ACK response to the server's SYN/ACK response).

  You can solve this problem in one of two ways. The simplest solution is to place the listening socket in blocking mode immediately before it calls accept. In blocking mode, accept returns only after the three-way handshake is complete and the socket is fully connected. The disadvantage of this solution is that if the three-way handshake cannot be completed, blocking can continue for up to 75 seconds. If you use this solution, you must return the listening socket to nonblocking mode.

  A more complex solution is to leave the listening socket in nonblocking mode. How you implement this solution depends on the logic of the application. If the application calls select and then waits for the client to send it something (that is, the application does **not** immediately send the client a message after accept completes), you do not need to make additional changes. The select function will not indicate that the accepted socket is ready for reading until the client has sent it a message or the three-way handshake has failed. If, on the other hand, the logic of the application calls for it to send a message as soon as accept completes, the application must be able to handle the ENOTCONN error from send or the application must have special code that tests the connection before using it.

Data Delivery and Record Boundaries

For UDP sockets of the type SOCK_DGRAM, OpenVOS restricts to 32K bytes the amount of data that an application can send or receive in a single write or read function call. If an application attempts to transfer more than 32K bytes of data, the write or read function returns an error to the application.

Successful completion of a write function call indicates that the data to be written has been passed to the protocol stack. The application can then free or reuse the buffer space used by the previous write function call. For a SOCK_DGRAM socket, the data then moves onto the media in a timely fashion, depending on network activity on the media. For a SOCK_STREAM socket, the data is sent to the destination as soon as the sending protocol receives permission from the receiving protocol.

The protocols used for SOCK_STREAM sockets do not preserve record boundaries for the incoming data streams. The application must determine boundaries or where messages end. In particular, when an application calls one of the read functions, the protocol returns all available data up to the size of the buffer passed from the protocol stack. To ensure that it reads the complete message, the application should perform multiple reads and maintain a running tally of the amount of data returned by each read.

Security

The application is responsible for security. For example, the TCP/IP protocols do not check access rights and passwords. The application must terminate connections that do not have proper access.

Creating Source Code That Is POSIX.1- and ANSI C-Compliant

The STCP programming interface complies with the POSIX.1 and ANSI C standards. POSIX.1 refers to Part 1 of the IEEE POSIX standard, which is a system API. POSIX.1 support enables OpenVOS programmers to port applications that conform to the POSIX.1 standard, with minimal source-code modification. You can import most POSIX.1-compliant TCP/IP applications directly to OpenVOS with no modifications and run them correctly using STCP and the OpenVOS POSIX.1 environment.

The OpenVOS Standard C compiler, which you invoke using the cc command, allows you to create programs that are ANSI C- and/or POSIX.1-compliant. Stratus recommends that you use the cc command, which invokes the OpenVOS Standard C compiler, rather than the c command, which invokes the older OpenVOS C compiler. For information about the cc command, see the *OpenVOS Commands Reference Manual* (R098) or the *OpenVOS Standard C User's Guide* (R364).

For POSIX.1-compliant applications, you do not need to add special include libraries for the compiler, but the source code must contain one (and only one) of the following definitions before any headers are included.

```
#define _POSIX_SOURCE /* 1990 version */
#define _POSIX_SOURCE 1 /* 1990 version */
#define _POSIX_C_SOURCE 199309L /* 1993 version */
#define _POSIX_C_SOURCE 199506L /* 1996 version */
#define _POSIX_C_SOURCE 200112L /* 2001 version */
```

NOTES

1. If you use a `_POSIX_C_SOURCE` definition, you must bind with the POSIX object library.
2. If you use the `_POSIX_C_SOURCE 200112L` definition, the declarations in the STCP header files become strictly POSIX compliant (2001 is the first version of POSIX to define the socket and Internet interfaces). This definition simplifies compiling ported code, but will cause some compilation errors with older VOS code.

The OpenVOS Standard C compiler may generate numerous compiler warnings the first time you use it to compile a TCP/IP program if you had previously used the OpenVOS C compiler to compile the program. A common cause of warnings is the absence of function prototypes. You may need to include additional header files (such as `string.h`, `stdio.h`, and `stdlib.h`) that contain the prototypes for many standard C language functions in order to remove these warnings. For example, if your program uses the `strcpy` function, the header-file definition list must include the header file `string.h`. For information about the OpenVOS Standard C header files and the functions that require them, see the *OpenVOS Standard C Reference Manual* (R363).

You can eliminate other compiler compatibility problems by using one or all of the compatibility arguments of the `cc` command: `-compatible_bitfields`, `-compatible_search`, and `-compatible_generics`. For information about these arguments, see the *OpenVOS Standard C User's Guide* (R364).

To create POSIX.1-compliant source code for OpenVOS, you should also read the following additional documentation:

- "Binding an Application" on page 4-3, which describes the path name that you need to add to the object-library search paths of the process that you use to bind a POSIX.1-compliant application.
- *OpenVOS POSIX.1: Conformance Guide* (R217M), which describes how the OpenVOS POSIX.1 implementation adheres to or deviates from the POSIX standard. This document is available only on the OpenVOS StrataDOC Web site: `http://stratadoc.stratus.com`.
- *OpenVOS POSIX.1 Reference Guide* (R502), which documents the OpenVOS POSIX features.

The stcp_calls Command

The stcp_calls command is a tool that assists application development by simulating an STCP environment in which you can issue requests that mimic the application.

Effective use of this tool requires a terminal and a process that acts as either the server or client side of an application, or as both. You issue the stcp_calls command from OpenVOS command level and enter an STCP-like subsystem. Within this subsystem, you can issue requests that represent a subset of the STCP functions. Additional requests, such as buf_copy and buf_read, are available for modifying the input and output buffers used when reading and writing data. You issue stcp_calls requests directly (that is, at STCP request level) rather than from within an application.

After you issue the stcp_calls command, you can issue requests to establish a connection between the client and the server, simulating the activities that an application would perform to establish the connection. Output from the stcp_calls requests issued on the client side appear on the terminal's screen on which the client process is running, and output from the stcp_calls requests issued on the server side appear on the terminal's screen on which the server process is running. For information on the stcp_calls command, see *OpenVOS POSIX.1 Reference Guide* (R502).

Displaying and Changing Values of STCP Variables

You can display information about STCP variables by using the netstat command with the -statistics argument and with the value tcp or tcp/ext for the -protocol argument. You can also display information or change values of variables by using the access_info_monitor command. For information on these commands, see the *OpenVOS STREAMS TCP/IP Administrator's Guide* (R419).

The analyze_system subsystem (which you enter by issuing the analyze_system command) includes the following STCP requests (for complete information about these requests, see the *OpenVOS System Analysis Manual* (R073)):

- list_stcp_params displays information about the STCP parameters that you can change, and set_stcp_param enables you to change the value of an individual STCP parameter. However, the access_info_monitor command provides more complete access to variables affecting STCP performance.
- stcp_meters displays active TCP meters. You can either interactively monitor running totals over a period of time or scroll through relevant data in a spreadsheet format. However, the netstat command with the value tcp/ext for the -protocol argument provides a more complete display of STCP statistics.

Window Size

In STCP, the default receive-window size is 8192 bytes. TCP advertises this size to the peer, and the peer may send data in amounts up to that size without waiting to receive an acknowledgement. You can use the SO_RCVBUF socket-level option of the setsockopt function to adjust the receive-window size (see the description of SO_RCVBUF in Table 5-9). You must use the SO_RCVBUF socket-level option of the setsockopt if you want a receive-window size larger than 65,535.

STCP allows you to set the maximum number of users (that is, the maximum number of sockets) for each of several preconfigured receive-window sizes that are larger than the default size, and that are assigned to sockets as they are created. To do so, use the following variables, which you can set using the access_info_monitor command (these variables also exist as parameters of the set_stcp_param request of analyze_system, listed in parentheses, below):

- tcpVosMaxNum256kWindows (max_256k_windows) to set the maximum number of users for receive windows whose size is 256K.
- tcpVosMaxNum64kWindows (max_64k_windows) to set the maximum number of users for receive windows whose size is 64K.
- tcpVosMaxNum32kWindows (max_32k_windows) to set the maximum number of users for receive windows whose size is 32K.
- tcpVosMaxNum16kWindows (max_16k_windows) to set the maximum number of users for receive windows whose size is 16K.

For example, the following procedures set the maximum number of users to 100 for receive windows whose size is 64K bytes for sockets that require a receive-window size greater than the default:

- Using the access_info_monitor command—First, determine the object identifier (OID) for the variable tcpVosMaxNum64kWindows. Issue the access_info_monitor command with no arguments. In the command output, search for tcpVosMaxNum64kWindows and record its OID. Then, issue the access_info_monitor command, setting the OID to 100, as in the following example:

  ```
  access_info_monitor 1.3.6.1.4.1.458.114.1.7.3.353 set 100
  ```

- Using the analyze_system subsystem:

  ```
  set_stcp_param max_64k_windows 100
  ```

As another example, the following requests set the maximum number of users to 0 for receive windows that are 256K, 64K, 32K, and 16K bytes, thereby preventing TCP from allocating to non-privileged users receive windows that are larger than the default size.

- Using the `access_info_monitor` command—First, determine the OID for the variables. Issue the `access_info_monitor` command with no arguments. In the command output, search for the variables and record their OIDs. Then, issue the `access_info_monitor` command, setting the OID to 0, as in the following examples:

```
access_info_monitor 1.3.6.1.4.1.458.114.1.7.3.354 set 0
access_info_monitor 1.3.6.1.4.1.458.114.1.7.3.353 set 0
access_info_monitor 1.3.6.1.4.1.458.114.1.7.3.352 set 0
access_info_monitor 1.3.6.1.4.1.458.114.1.7.3.351 set 0
```

- Using the `analyze_system` subsystem:

```
set_stcp_param max_256k_windows 0
set_stcp_param max_64k_windows 0
set_stcp_param max_32k_windows 0
set_stcp_param max_16k_windows 0
```

Be aware of system resources when you change the maximum number of users for receive windows and size of receive windows because allocating many large windows can exhaust STREAMS memory.

In some cases, larger receive-window sizes cause significant performance gains because larger sizes allow the sender to send more data without waiting for an acknowledgement—TCP throughput is limited by the window size divided by the round trip latency. The downside of large receive-window sizes is that retransmission can become inefficient on unreliable connections.

NOTE

These tuning parameters have no effect on sending data.

By default, STCP has no maximum send-window size; 1,073,725,440 bytes is the maximum size that the TCP header can advertise. Also by default, STCP buffers outgoing data up to the peer's advertised window size. You can reduce this window size, thereby requiring fewer memory resources, by using the `max_send_ws` (maximum send-window size) parameter, which has a range of 4,096 to 1,073,725,440 bytes and a default value of 1,073,725,440 bytes. This parameter places a limit on how much data STCP will buffer for the local application while waiting for the remote side to acknowledge the data already sent.

You can display the number of windows that are larger than the default size and that are currently in use (though you cannot set a value for windows currently in use) using

the command `netstat -64 -statistics -protocol tcp/ext` (or by using the `list_stcp_params` request of the `analyze_system` command, in parentheses, below). In the command output, the following lines provide this information.

```
tcpVosCurNum256kWindows (current 256k windows n)
tcpVosCurNum64kWindows (current 64k windows n)
tcpVosCurNum32kWindows (current 32k windows n)
tcpVosCurNum16kWindows (current 16k windows n)
tcpVosCurNum8kWindows (current 8k windows n)
```

If you set lower values for the corresponding parameters (`max_256k_windows`, `max_64k_windows`, `max_32k_windows`, and `max_16k_windows` for windows of 256K, 64K, 32K, and 16K bytes, respectively), the number of windows currently in use decreases only after a socket that has been allocated this sized window is closed.

The `netstat` command (`list_stcp_params` request of the `analyze_system` command) also includes the `tcpVosUseBigWindows` variable (or `big_windows` parameter), which indicates if windows that are larger than the default size are available. The value `no` indicates that no windows larger than the default size are available until a socket closes. After increasing the number of `max-`*`nn`*`k_windows`, you should also change the value of the `big_windows` parameter to `yes` so that new maximum values can be immediately used. Note that setting the `big_windows` parameter to `yes` has no effect on sockets that already use a window size that is larger than the default. You must close and reopen such a socket in order to use the new, larger size.

For information on the commands `access_info_monitor` and `netstat`, see *OpenVOS STREAMS TCP/IP Administrator's Guide* (R419). For information on the `list_stcp_params` and `set_stcp_param` requests of the `analyze_system` command, see the *OpenVOS System Analysis Manual* (R073).

Handling the PSH Bit in Received TCP Packets

To control handling of the push (PSH) bit in received TCP packets, you can set a value for the STCP object identifier (OID) variable `tcpVosRecvPshFlagHandling` (1.3.6.1.4.1.458.114.1.7.3.254) using the `access_info_monitor` command. You can also use this command to view the variable's setting. Controlling how the PSH bit is handled may be necessary because some peers do not adhere to RFC-1122, which mandates that TCP applications use the PSH bit.

Values for `tcpVosRecvPshFlagHandling` are as follows:

- `always_use_psh` (`0`) specifies the behavior defined by RFC-1122.
- `never_use_psh` (`1`) specifies that STCP ignores receive PSH bits. Setting this value may result in severe performance degradation.

- `use_psh_if_seen` (`2`) specifies that STCP ignores the PSH bit on a connection until the first time the peer sets it. This value enables STCP to handle peers that do not set the PSH bit correctly, but the value also allows for the performance enhancement that results when most peers set the PSH bit correctly.

  The performance enhancement is not achieved, however, in the rare case when a peer explicitly manipulates the push bit by leaving it off for a long time, possibly even until the last segment.

Ideally, peers should be fixed to implement the PSH bit, and then `tcpVosRecvPshFlagHandling` should be set to `always_use_psh`.

Specifying How the TCP Maximum Segment Size Is Determined

The `tcpVosSendLegacyMSS` option controls what TCP sends for the Maximum Segment Size (MSS) on the initial synchronization (SYN) packets.

Values for `tcpVosSendLegacyMSS` are as follows:

- When `off`(`0`), it generates the RFC-1122 compliant value, which is derived from the MTU of the local subnet. For IPv4, the MSS is 20 bytes less than the MTU. For IPv6, the MSS is 40 bytes less than the MTU. The MSS represents the largest possible value the receiver can accept (section 4.2.2.6 in RFC-1122). In practice, the sender must further reduce the actual sent packet size to account for the MTU of other hops (if the peer is not on the same subnet), IPSec headers, IP options, and TCP options.
- When `on`(`1`), it generates an MSS value, taking routing into account. That is, it reduces the MSS if the peer is on another subnet. This may be used to reduce fragmentation when communicating with peers that do not adhere to RFC-1122. You probably will not use this option because RFC-1122 is outdated (1989).

Enabling POSIX Applications to Accept on Many Sockets

The runtime routine `super_listener` enables POSIX applications to accept many thousands of sockets (many more than the 4000 it would be limited to by the per-process port limit), with each socket being handled by an OpenVOS task. This library routine manages forking listeners as the number of incoming connections increases.

For details about the routine, see the header file `super_listener.h`, which is located in the `(master_disk)>system>include_library` directory.

The tools library (`(master_disk)>system>tools_library`) contains a simple echo client (`super_echo.pm`) that uses `super_listener`.

Source code for the programs `super_listener` and `super_echo` is located in the `(master_disk)>system>sample_programs>stcp` directory.

Chapter 5
Socket-Library Functions

This chapter describes the STCP socket-library functions, which are listed in Table 5-1.

Table 5-1. STCP Socket-Library Functions

accept	getservbyport	ntohs
accept_on	getservent	receive_socket
bind	get_socket_event	recv
connect	getsockname	recvfrom
endhostent	getsockopt	recvmsg
endnetent	htonl	send
endprotoent	htons	sendmsg
endservent	inet_addr	sendto
gethostbyaddr	inet_aton	sethostent
gethostbyname	inet_lnaof	sethostname
gethostent	inet_makeaddr	setnetent
gethostname	inet_netof	setprotoent
getnetbyaddr	inet_network	setservent
getnetbyname	inet_ntoa	setsockopt
getnetent	inet_ntop	shutdown
getpeername	inet_pton	so_recv
getprotobyname	listen	socket
getprotobynumber	map_stcp_error	stcp_spawn_process
getprotoent	ntohl	transfer_socket
getservbyname		

Table 5-2 lists functions that are routinely used in STCP applications but that are not part of the STCP socket library. Other manuals describe these functions.

Table 5-2. Functions Used in STCP Applications That Are Not in the STCP Socket Library

Function	Documentation
`close`	*OpenVOS Standard C Reference Manual* (R363)
`fcntl`	*OpenVOS Communications Software: STREAMS Programmer's Guide* (R306)
`fork`	*OpenVOS POSIX.1: Conformance Guide* (R217M), which describes the function and how the OpenVOS POSIX.1 implementation adheres to or deviates from the POSIX standard. (This document is available only on the OpenVOS StrataDOC Web site.) *OpenVOS POSIX.1 Reference Guide* (R502), which provides information about the function and other OpenVOS POSIX.1 features.
`ioctl`	This manual (see `ioctl`) as well as the *OpenVOS Communications Software: STREAMS Programmer's Guide* (R306)
`poll`	OpenVOS Subroutines manuals
`read`	*OpenVOS Standard C Reference Manual* (R363)
`readv`	
`select`	
`select_with_events`	
`write`	
`writev`	

STCP Socket-Library Function Reference

This section describes the STCP socket-library functions in alphabetical order.

Format for Socket-Library Function Descriptions

The socket-library function descriptions have the following format.

function_name

The name of the function is at the top of the first page of the description.

Purpose

Explains briefly what the function does.

Syntax

Lists the header files needed by the function and shows the function's prototype.

Arguments

Describes the function's arguments.

Explanation

Describes how to use the function.

Return Values

Provides the function's return values, if any.

Error Codes

Describes the error codes returned by the function. Note that this manual refers to *generic networking error codes*, which are similar to but not necessarily equivalent to UNIX error codes.

NOTE

Since the numeric values of STCP error codes can differ from the numeric values of actual UNIX error codes, you should not hard-code numeric error-code values in your applications.

Related Functions

Lists other functions similar to or useful with this function.

accept

Purpose

The accept function accepts a connection request on a socket from a peer process.

Syntax

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

int accept(int s, struct sockaddr *a, int *al);
```

Arguments

- `s` (input)
  A descriptor for a socket that the calling process created using the `socket` function. The descriptor is returned by the function when the process creates the socket. The process must have bound the socket to an IP address and port number with the `bind` function and must be listening for connection requests on the socket using the `listen` function.

- `a` (output)
  The IP address and port number of the peer process that is requesting the connection. The address family of the connecting sockets determines the exact format of `a`.

- `al` (input/output)
  On input, the length, in bytes, of the space to be initialized for `a`. On output, the actual length, in bytes, of the returned IP address and port number.

Explanation

The `accept` function extracts the first connection request from the queue of pending connection requests, creates a new socket with the same properties as the original socket (`s`), and allocates a new socket descriptor for the new socket.

If no connection requests are pending on the queue, `accept` blocks the calling process until a connection request is present. If no connection requests are pending and the

socket is in nonblocking mode, `accept` returns `-1` and sets `errno` to `EWOULDBLOCK` (or the equivalent value `EAGAIN`).

The accepted socket cannot be used to accept additional connections. The original socket (`s`) remains open.

Return Values

If successful, `accept` returns a non-negative integer that is the descriptor for the new socket with which the connection will be established. If unsuccessful, it returns the value `-1`.

Error Codes

If `accept` is unsuccessful, it sets the global variable `errno` (an `int` value) to the error code for one of the following error messages to indicate the specific error.

Error Message	Description
`EBADF`	The specified socket descriptor is invalid.
`EINTR`	A signal was caught during the `accept` call.
`EINVAL`	The address specified by `a` is not writable or `al` is not equal to `sizeof(struct sockaddr)`.
`EIO`	An internal error has occurred.
`EMFILE`	No more file descriptors are available for this process (a maximum of 4096 are available).
`ENFILE`	No more sockets of the type `SOCK_STREAM` are available for this system.
`ENOBUFS`	Insufficient system resources are available to complete the request.
`ENOTSOCK`	The descriptor does not refer to a socket.
`EOPNOTSUPP`	The referenced socket is not of type `SOCK_STREAM`.
`EWOULDBLOCK` (or the equivalent value `EAGAIN`)	The requested operation would block the process.

Related Functions

See `bind`, `listen`, `select`, and `socket`.

accept_on

Purpose

The accept_on function accepts a connection request on a specific socket from a peer process. You typically use the accept_on function in an application to accept a connection on the same socket that the application process is already listening on. If you do not use this function as documented, the listening socket is destroyed.

Syntax

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

int accept_on(int s, struct sockaddr *a, int *al, int nsock);
```

Arguments

- s (input)
  A descriptor for a socket created by the calling process with socket. The descriptor is returned by socket when the process creates the socket. The process must have bound the socket to a socket address with bind and must be listening for connection requests on the socket using listen.

- a (output)
  The socket address of the peer process that is requesting the connection. The exact format of a is determined by the address family of the connecting sockets.

- al (input/output)
  On input, the length, in bytes, of the space to be initialized for a. On output, the actual length, in bytes, of the returned socket address.

- nsock (input)
  The specified socket. The accept_on function does not create a new socket for nsock; rather, it uses the one you specify. This socket must be identical to the first socket (that is, the value for nsock must equal the value for s) or this socket must have been bound with a null address (for example, bind(socket, NULL, 0). If this socket and the first socket are identical, the listen function must have specified a backlog of 1 (that is, listen(socket, 1)). In addition, the

parameters used in the socket call that creates nsock must equal the parameters used to create s.

Explanation

The accept_on function extracts the first connection request from the queue of pending connection requests. It uses nsock as the specific socket on which you want a connection request.

If no connection requests are pending on the queue, accept_on blocks the calling process until a connection request is present. If no connection requests are pending and the socket is in nonblocking mode, accept_on returns the error code for EWOULDBLOCK. The original socket (s) remains open. The accepted socket (nsock) cannot be used to accept additional connections.

Return Values

If successful, accept_on returns a non-negative integer that is the descriptor for nsock. If unsuccessful, it returns the value -1.

Error Codes

If accept_on is unsuccessful, it sets the global variable errno (an int value) to the error code for one of the following error messages to indicate the specific error.

Error Message	Description
EBADF	The specified socket descriptor is invalid.
EINTR	A signal was caught during the accept_on call.
EINVAL	The address specified by a is not writable or al is not equal to sizeof(struct sockaddr).
EIO	An internal error has occurred.
EMFILE	The operating system cannot allocate any more sockets.
ENFILE	The TCP driver file table has run out of sockets.
ENOSR	Insufficient system resources are available to complete the acceptance of the incoming request.
ENOSTR	The file descriptor s is not associated with a stream.
ENOTTY	The descriptor references a file, not a socket.
ENXIO	No more sockets can be created for the socket type.
EOPNOTSUPP	The referenced socket is not of type SOCK_STREAM.
EWOULDBLOCK	The requested operation would block the process.

Related Functions

See accept, bind, listen, select, and socket.

bind

Purpose

For sockets of the type SOCK_STREAM and SOCK_DGRAM, the bind function binds an IP address and port number to a socket that was previously created with the socket function. For sockets of the type SOCK_RAW, the bind function sets a protocol value for the socket.

Syntax

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

int bind(int s, struct sockaddr *a, int al);
```

Arguments

- s (input)
  A descriptor for a socket that the calling process created using the socket function. The descriptor is returned by the function when the process creates the socket.

- a (input)
  The IP address and port number to be bound to the specified socket(s), for sockets of the type SOCK_STREAM and SOCK_DGRAM.

  For sockets of the type SOCK_RAW, use the sin_port field of the sockaddr_in structure to specify a protocol value. When you call bind and you have set up the required value(s) in a sockaddr_in structure, you must cast the pointer to the sockaddr_in structure to a pointer to a sockaddr structure.

- al (input)
  The length, in bytes, of the IP address and port number (a). You must set this value to sizeof(struct sockaddr).

Explanation

For sockets of the type SOCK_STREAM and SOCK_DGRAM, the bind function assigns an IP address and port number to an unnamed socket. If you do not specify a port number, STCP dynamically assigns one to the socket from the dynamic port range.

When a process creates a socket with socket, that socket is associated with an address family but does not yet have its own IP address and port number. Other processes cannot connect to this socket until the process owning the socket assigns it an address using bind.

A socket can bind to only **one** unicast or multicast address. To bind to a specific IP address, you must specify the local address.

After binding an IP address and port number to a socket, the calling process can listen for connection requests on that socket. (See the descriptions of the accept, connect, and listen functions.)

Except for multicast addresses, addresses bound to sockets always use the network/host address where the socket resides. The only part of the address that you need to change is the port address. By convention, to specify the current network/host, you specify an address of INADDR_ANY. Therefore, to bind to port 2000 on the current network/host, you specify a bind address of INADDR_ANY,2000. (For examples, see the sample programs in Appendix A.)

NOTE

Binding to INADDR_ANY actually binds to all local IP addresses at the same time, not just to the current network/host address.

When you use the bind function to associate a protocol with a socket of the type SOCK_RAW, use protocol values as defined in the netinet/in.h header file. The value IPPROTO_IP enables the socket to receive messages for all protocols. Specify the protocol value in the sin_port field of the sockaddr_in structure, which is a redefinition of the sockaddr structure. When you call bind and you have set up the required value(s) in a sockaddr_in structure, you must cast the pointer to the sockaddr_in structure to a pointer to a sockaddr structure.

Return Values

If successful, bind returns the value 0. If unsuccessful, it returns the value -1.

Error Codes

If `bind` is unsuccessful, it sets the global variable `errno` (an `int` value) to the error code for one of the following error messages to indicate the specific error.

Error Message	Description
`EACCES`	The caller has insufficient access or privileges to perform the requested operation.
`EADDRINUSE`	The specified address is already in use and is not reusable.
`EADDRNOTAVAIL`	The specified address cannot be used.
`EAFNOSUPPORT`	The specified address family is not supported in this software version.
`EBADF`	The specified socket descriptor is invalid.
`EFAULT`	The `a` argument is not in a valid part of the user address space.
`EINTR`	A signal was caught during the `bind` call.
`EINVAL`	The socket is already bound to an address or al is not equal to `sizeof(struct sockaddr)`, or the `listen` function has already been called on this socket.
`EISCONN`	The socket is already connected.
`ENETDOWN`	The network is down.
`ENOBUFS`	Insufficient system resources are available to complete the request.
`ENOTSOCK`	The socket descriptor `s` does not refer to a socket.

Related Functions

See `accept`, `connect`, `listen`, and `socket`.

close

This function is provided by the OpenVOS Standard C library. For information about using the `close` function with sockets, see ‘‘The close Function’’ on page 2-21. For information about using `close` with file descriptors, see the *OpenVOS Standard C Reference Manual* (R363).

connect

Purpose

The connect function requests a connection on a socket.

Syntax

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

int connect(int s, struct sockaddr *a, int al);
```

Arguments

- s (input)
  A descriptor for a socket that the calling process created using the socket function. The descriptor is returned by the function when the process creates the socket.

- a (input)
  The IP address and port number of a peer process to which the calling process wants to send data. For a connection to be successful, the peer process must have bound this address to a socket and called listen for that socket.

- al (input)
  The length, in bytes, of the address (a). You must set this value to sizeof(struct sockaddr).

Explanation

The connect function requests a connection with a specified IP address and port number (a). You cannot use the connect function with sockets of the type SOCK_RAW.

If the socket (s) is of type SOCK_DGRAM (datagram), the IP address and port number identifies a particular peer process to which the calling process wants to send data. For more information about using the connect function in datagram communications, see "UDP Communications" in Chapter 2.

If the socket (s) is of type SOCK_STREAM (stream), a protocol-specific connection request is targeted to the entire address structure to which a points.

Return Values

If successful, connect returns the value 0. If unsuccessful, it returns the value -1.

Error Codes

If connect is unsuccessful, it sets the global variable errno (an int value) to the error code for one of the following error messages to indicate the specific error.

Error Message	Description
EAFNOSUPPORT	Addresses in the specified address family cannot be used with this socket.
EALREADY	A connection request is already in progress for the specified socket.
EBADF	The specified socket descriptor is invalid.
ECONNREFUSED	The attempt to connect was forcefully rejected.
ECONNRESET	The remote host reset the connection request.
EFAULT	The a argument specifies an area outside the process address space.
EINPROGRESS	The current operation is still in progress.
EINTR	A signal was caught during the connect call.
EINVAL	The value of al is not equal to sizeof(struct sockaddr).
EISCONN	The specified socket is connection mode and is already connected.
ENETDOWN	The network is down.
ENETUNREACH	No route to the specified address is defined.
ENOBUFS	Insufficient system resources are available to complete the request.
ENOTSOCK	The descriptor does not refer to a socket.
EOPNOTSUPP	The socket is listening and cannot be connected.
ETIMEDOUT	Connection establishment timed out without a connection being made.

Related Functions

See accept, getsockname, select, and socket.

See the *OpenVOS Communications Software: STREAMS Programmer's Guide* (R306) for additional information on using the fcntl function.

See the OpenVOS Subroutines manuals for additional information on using the `poll` function.

See the *OpenVOS Standard C Reference Manual* (R363) for information about the `select` function.

See the *OpenVOS System Analysis Manual* (R073) for information about the `analyze_system` subsystem.

endhostent

Purpose

The endhostent function closes the connections used by gethostbyaddr and gethostbyname to communicate with a name server.

Syntax

```
#include <netdb.h>

void endhostent();
```

Arguments

The endhostent function has no arguments.

Explanation

After the gethostbyaddr and gethostbyname functions have finished executing, an application should use the endhostent function to close the connections that have been set up with a name server and to clear any flags that may have been set using sethostent.

Return Values

The endhostent function returns no values.

Error Codes

None.

Related Functions

See gethostbyaddr, gethostbyname, gethostent, and sethostent.

endnetent

Purpose

The endnetent function closes the networks database file.

Syntax

```
#include <netdb.h>

void endnetent();
```

Arguments

The endnetent function has no arguments.

Explanation

The endnetent function closes the networks database file. (The networks database file resides in the directory (master_disk)>system>stcp.) This function must always be called after using getnetent to access the networks database file because getnetent does not close the file.

Return Values

The endnetent function returns no values.

Error Codes

None.

Related Functions

See getnetbyaddr, getnetbyname, getnetent, and setnetent.

endprotoent

Purpose

The endprotoent function closes the protocols database file.

Syntax

```
#include <netdb.h>

void endprotoent();
```

Arguments

The endprotoent function has no arguments.

Explanation

The endprotoent function closes the protocols database file. (The protocols database file resides in the directory (master_disk)>system>stcp.) This function must always be called after using getprotoent to access the protocols database file because getprotoent does not close the file.

Return Values

The endprotoent function returns no values.

Error Codes

None.

Related Functions

See getprotobyname, getprotobynumber, getprotoent, and setprotoent.

endservent

Purpose

The endservent function closes the services database file.

Syntax

```
#include <netdb.h>

void endservent();
```

Arguments

The endservent function has no arguments.

Explanation

The endservent function closes the services database file. (The services database file resides in the directory (master_disk)>system>stcp.) This function must always be called after using getservent to access the services database file because getservent does not close the file.

Return Values

The endservent function returns no values.

Error Codes

None.

Related Functions

See getservbyname, getservbyport, getservent, and setservent.

fcntl

This function is provided by the OpenVOS Standard C library. For detailed information about `fcntl`, see the *OpenVOS Communications Software: STREAMS Programmer's Guide* (R306). For information about using `fcntl` to set blocking and nonblocking mode, see ''Setting and Checking I/O Mode'' on page 3-1.

fork

This function is provided by the POSIX object library. For further information, see the *OpenVOS POSIX.1 Reference Guide* (R502) and the *OpenVOS POSIX.1: Conformance Guide* (R217M), which is available only on the OpenVOS StrataDOC Web site.

gethostbyaddr

Purpose

The gethostbyaddr function provides access to information about a host that has been specified by its host address.

Syntax

```
#include <netdb.h>

struct hostent *gethostbyaddr(char *addr, int len, int type);
```

Arguments

- addr (input)
  A pointer to the binary representation of the address of the host about which you want information. (For the address family AF_INET, this could be a pointer to the s_addr element of the in_addr structure.) (input)

- len (input)
  The length, in bytes, of the address specified as addr.

- type (input)
  The format of the address specified as addr. The value of type must be 2, to indicate the AF_INET address family.

Explanation

The gethostbyaddr function provides a pointer to information about a host that has been specified by its host address.

If the file resolv.conf exists, the function attempts to query a name server on the network to see if the name server recognizes the host address. (If resolv.conf exists, it resides in the directory (master_disk)>system>stcp.) By default, gethostbyaddr uses datagrams to communicate with the name server. To specify that it uses virtual circuits instead of datagrams, call the sethostent function and specify a nonzero integer value for the stayopen argument. This keeps the connection to the name server open until a response is received or the request times

out. If querying the name server is unsuccessful, the gethostbyaddr function queries the hosts file. If queries to both the name server and the hosts file are unsuccessful, the gethostbyaddr function returns a NULL pointer.

If the resolv.conf file does not exist, the gethostbyaddr function opens the hosts database file (located in the directory (master_disk)>system>stcp) and searches for an entry for a particular host. Unless otherwise specified, the function then closes the file.

If STCP attempts to query a DNS server, and if the server(s) identified in resolv.conf does not respond, a timeout occurs after 28 to 32 seconds. Waiting will be terminated if an ICMP error indicates that the server is not running the name service or the server cannot be reached.

An application can call sethostent (specifying a nonzero integer for the stayopen argument) before calling gethostbyaddr to specify that the file remain open after the function has finished executing. This allows the application to step through the remainder of the file entry by entry.

The hosts database file contains a one-line entry for each host known on the network. Each entry consists of the host's AF_INET address, its name, and any aliases.

The *OpenVOS STREAMS TCP/IP Administrator's Guide* (R419) describes how to set up the hosts database file.

If gethostbyaddr finds a match using either the name server or the hosts database file, it returns to the application a pointer to a hostent structure containing information about the specified host. The hostent structure is defined (in the header file netdb.h) as follows:

```
struct hostent {
     char          *h_name;
     char          **h_aliases;
     int           h_addrtype;
     int           h_length;
     char          **h_addr_list;
};
#define h_addr h_addr_list[0]
```

A description of each member of the hostent structure follows.

- The h_name field specifies the name of the host.
- The h_aliases field specifies an array of alternative names (aliases) for the host. Each alias is terminated by a zero.
- The h_addrtype field specifies the format of the host address. The value of h_addrtype is always 2, which indicates the AF_INET address family.

- The h_length field specifies the length, in bytes, of the host address.
- The h_addr_list field specifies a list of IP addresses for the host. The addresses are returned in network byte order (high-order byte first).

The definition of hostent defines h_addr as h_addr_list[0] for backward compatibility with BSD UNIX Version 4.2 structures.

NOTE

The hostent structure is static data that will be overwritten by subsequent calls.

Return Values

If successful, gethostbyaddr returns a pointer to a hostent structure containing information about the specified host. If unsuccessful (the host cannot be found, an end-of-file condition exists, or an error exists), it returns a NULL pointer.

Error Codes

If gethostbyaddr is unsuccessful, it sets the variable h_errno (an int value) to one of the following error codes to indicate the specific error. These error codes are defined in the system>stcp>include_library>netdb.h header file.

Error Codes	Description
HOST_NOT_FOUND	The specified host was not found.
TRY_AGAIN	The specified host was not found, or SERVERFAIL was returned.
NO_ADDRESS	No address was found.
NO_DATA	No data record exists of the requested type.
NO_RECOVERY	gethostbyname encountered a non-recoverable error (FORMERR, REFUSED, and NOTIMP).

Related Functions

See endhostent, gethostbyname, gethostent, and sethostent.

gethostbyname

Purpose

The gethostbyname function provides access to information about a host that has been specified by its host name.

Syntax

```
#include <netdb.h>
extern int h_errno;

struct hostent *gethostbyname(char *name);
```

Arguments

- name (input)
  A pointer to the name of the host about which you want information. The name must be a character string.

Explanation

The gethostbyname function provides a pointer to information about a host that has been specified by its host name.

If the file resolv.conf exists, the function attempts to query a name server on the network to see if the name server recognizes the host address. (If resolv.conf exists, it resides in the directory (master_disk)>system>stcp.) By default, gethostbyname uses datagrams to communicate with the name server. To specify that it uses virtual circuits instead of datagrams, call the sethostent function and specify a nonzero integer value for the stayopen argument. This keeps the connection to the name server open until a response is received or the request times out. If querying the name server is unsuccessful, the gethostbyname function returns a NULL pointer.

If the file resolv.conf does not exist, the gethostbyname function opens the hosts database file (located in the directory (master_disk)>system>stcp) and searches for an entry for a particular host. Unless otherwise specified, the function then closes the file.

If STCP attempts to query a DNS server, and if the server(s) identified in `resolv.conf` does not respond, a timeout occurs after 28 to 32 seconds. Waiting will be terminated if an ICMP error indicates that the server is not running the name service or the server cannot be reached.

An application can call `sethostent` (specifying a nonzero integer for the `stayopen` argument) before calling `gethostbyname` to specify that the file remain open after the function has finished executing. This allows the application to step through the remainder of the file entry by entry.

The `hosts` database file contains a one-line entry for each host known on the network. Each entry consists of the host's `AF_INET` address, its name, and any aliases. (The *OpenVOS STREAMS TCP/IP Administrator's Guide* (R419) explains how to set up the `hosts` database file.)

If `gethostbyname` finds a match using either the name server or the `hosts` database file, it returns to the application a pointer to a `hostent` structure containing information about the specified host. See the description of the `gethostbyaddr` function for more information about the `hostent` structure.

Return Values

If successful, `gethostbyname` returns a pointer to a `hostent` structure containing information about the specified host. If unsuccessful (the host cannot be found, an end-of-file condition exists, or an error exists), it returns a `NULL` pointer.

Error Codes

If `gethostbyname` is unsuccessful, it sets the variable `h_errno` (an `int` value) to one of the following error codes to indicate the specific error. These error codes are defined in the `system>stcp>include_library>netdb.h` header file.

Error Codes	Description
`HOST_NOT_FOUND`	The specified host was not found.
`TRY_AGAIN`	The specified host was not found, or `SERVERFAIL` was returned.
`NO_ADDRESS`	No address was found.
`NO_DATA`	No data record exists of the requested type.
`NO_RECOVERY`	`gethostbyname` encountered a non-recoverable error (`FORMERR`, `REFUSED`, and `NOTIMP`).

Related Functions

See `endhostent`, `gethostbyaddr`, `gethostent`, and `sethostent`.

gethostent

Purpose

The gethostent function provides access to information about any host for which the hosts database file contains an entry. (The hosts database file resides in the directory (master_disk)>system>stcp.)

Syntax

```
#include <netdb.h>
struct hostent *gethostent();
```

Arguments

The gethostent function has no arguments.

Explanation

The gethostent function opens the hosts database file (if it is not already open), reads an entry from the file, and returns a pointer to a hostent structure containing information about the host described in that entry. The function does not close the file upon completion.

The entry that gethostent reads from the file depends on whether the hosts database file is open when the function is called. If the file is not open, the function reads the first entry. If the file is already open, the function reads the next entry after the current file position. Therefore, an application can call gethostent repeatedly to step through the file entry by entry.

An application can call sethostent before calling gethostent to guarantee that the file marker is set to the beginning of the file. However, it is redundant to call sethostent and specify a nonzero integer for the stayopen argument to specify that the file remain open when gethostent has finished executing.

The hosts database file contains a one-line entry for each host known on the network. Each entry consists of the host's AF_INET address, its name, and any aliases.

The *OpenVOS STREAMS TCP/IP Administrator's Guide* (R419) explains how to set up the `hosts` database file.

The `hostent` structure to which `gethostent` returns a pointer contains information about a host. See the description of the `gethostbyaddr` function for more information about the `hostent` structure.

An application can call `gethostent` repeatedly to step through the entries in the `hosts` database file. To reset the file marker to the beginning of the file, the application calls the `sethostent` function.

Return Values

If successful, `gethostent` returns a pointer to the `hostent` structure associated with the entry read by the function. If unsuccessful (the host cannot be found, an end-of-file condition exists, or an error exists), it returns a `NULL` pointer.

Error Codes

None.

Related Functions

See `endhostent`, `gethostbyaddr`, `gethostbyname`, and `sethostent`.

gethostname

Purpose

The gethostname function returns the name of the host module.

Syntax

```
#include <netdb.h>
int gethostname(char *name, int namelen);
```

Arguments

- name (output)
  The buffer that is to receive the returned name.

- namelen (input)
  The length, in bytes, of the buffer specified as name.

Explanation

The gethostname function returns the standard host name of the host module, which gethostname finds by reading the file (master_disk)>system>stcp>host. This name is set in the host file when STCP is configured on the module for the first time. The size of the variable name is specified by namelen. The variable name should be declared with the size MAXHOSTNAMELEN, which has a value of 256 (including the null byte). The returned name is null-terminated and may be truncated if namelen provides insufficient space.

Return Values

If successful, gethostname returns the value 0. If unsuccessful, it returns the value -1.

Error Codes

If `gethostname` is unsuccessful, it sets the global variable `errno` (an `int` value) to the error code for the following error message to indicate the specific error.

Error Message	Description
`EFAULT`	The `name` or `namelen` argument contained an invalid value.

Related Functions

See `sethostname`.

getnetbyaddr

Purpose

The `getnetbyaddr` function provides access to information about a network that has been specified by network number.

Syntax

```
#include <netdb.h>

struct netent *getnetbyaddr(unsigned long net, int type);
```

Arguments

- `net` (input)
  The network number of the network about which you want information.

- `type` (input)
  The address format of the network number about which you want information. You must set `type` to the value `AF_INET`.

Explanation

The `getnetbyaddr` function opens the `networks` database file in the directory `(master_disk)>system>stcp` (if it is not already open) and searches for an entry for a particular network (identified by its network number). If successful, the function returns a pointer to a `netent` structure that contains information about the specified network. Unless otherwise specified, the function then closes the file.

An application can call `setnetent` (specifying a nonzero integer value for the `f` argument) before calling `getnetbyaddr` to specify that the file remain open after the function has finished executing. This allows the application to step through the remainder of the file entry by entry.

The `networks` database file contains a one-line entry for each known network. Each entry consists of the network's name, its network number, and any aliases.

The *OpenVOS STREAMS TCP/IP Administrator's Guide* (R419) explains how to set up the `networks` database file.

The `netent` structure to which `getnetbyaddr` returns a pointer contains information about the specified network. The structure is defined (in the header file `netdb.h`) as follows:

```
struct netent{
     char      *n_name;
     char      **n_aliases;
     int       n_addrtype;
     long      n_net;
};
```

A description of each member of the `netent` structure follows.

- The `n_name` field specifies the name of the network.
- The `n_aliases` field specifies an array of alternative names (aliases) for the network. Each alias is terminated by a zero.
- The `n_addrtype` field specifies the format of the network number. The value of `n_addrtype` is always `2`, which indicates the `AF_INET` address family.
- The `n_net` field specifies the network number. Network numbers are returned in host byte order.

NOTE

The `netent` structure is static data that will be overwritten by subsequent calls.

Return Values

If successful, `getnetbyaddr` returns a pointer to a `netent` structure containing information about the specified network. If unsuccessful (an end-of-file condition exists or an error exists), it returns a `NULL` pointer.

Error Codes

None.

Related Functions

See `endnetent`, `getnetbyname`, `getnetent`, and `setnetent`.

getnetbyname

Purpose

The getnetbyname function provides access to information about a network that has been specified by name.

Syntax

```
#include <netdb.h>

struct netent *getnetbyname(char *name);
```

Arguments

- name (input)
  The name of the network about which you want information.

Explanation

The getnetbyname function opens the networks database file in the directory (master_disk)>system>stcp (if it is not already open) and searches for an entry for a particular network (identified by its network name). If successful, the function returns a pointer to a netent structure that contains information about the specified network. Unless otherwise specified, the function then closes the file.

An application can call setnetent (specifying a nonzero integer value for the f argument) before calling getnetbyname to specify that the file remain open after the function has finished executing. This allows the application to step through the remainder of the file entry by entry.

The networks database file contains a one-line entry for each known network. Each entry consists of the network's name, its network number, and any aliases.

The *OpenVOS STREAMS TCP/IP Administrator's Guide* (R419) explains how to set up the networks database file.

The `netent` structure to which `getnetbyname` returns a pointer contains information about the specified network. See the description of the `getnetbyaddr` function for more information about the `netent` structure.

Return Values

If successful, `getnetbyname` returns a pointer to a `netent` structure containing information about the specified network. If unsuccessful (an end-of-file condition exists or an error exists), it returns a `NULL` pointer.

Error Codes

None.

Related Functions

See `endnetent`, `getnetbyaddr`, `getnetent`, and `setnetent`.

getnetent

Purpose

The getnetent function provides access to information about any network for which the networks database file contains an entry. (The networks database file resides in the directory (master_disk)>system>stcp.)

Syntax

```
#include <netdb.h>

struct netent *getnetent();
```

Arguments

The getnetent function has no arguments.

Explanation

The getnetent function opens the networks database file in the directory (master_disk)>system>stcp (if it is not already open), reads an entry from the file, and returns a pointer to a netent structure containing information about the network described in that entry. The function does not close the file upon completion.

The entry that getnetent reads from the file depends on whether the networks database file is open when the function is called. If the file is not open, the function reads the first entry. If the file is already open, the function reads the next entry after the current file position. Therefore, an application can call getnetent repeatedly to step through the file entry by entry.

An application can call setnetent before calling getnetent to guarantee that the file marker is set to the beginning of the file. However, it is redundant to call setnetent and specify a nonzero integer value for the stayopen argument to specify that the file remain open when getnetent has finished executing.

The networks database file contains a one-line entry for each known network. Each entry consists of the network's name, its network number, and any aliases.

The *OpenVOS STREAMS TCP/IP Administrator's Guide* (R419) explains how to set up the `networks` database file.

The `netent` structure to which `getnetent` returns a pointer contains information about the specified network. See the description of the `getnetbyaddr` function for more information about the `netent` structure.

Return Values

If successful, `getnetent` returns a pointer to the `netent` structure associated with the entry read by the function. If unsuccessful (an end-of-file condition exists or an error exists), it returns a `NULL` pointer.

Error Codes

None.

Related Functions

See `endnetent`, `getnetbyaddr`, `getnetbyname`, and `setnetent`.

getpeername

Purpose

The `getpeername` function returns the socket address (that is, the name) of the peer process connected to the specified socket.

Syntax

```
#include <sys/socket.h>
int getpeername(int s, struct sockaddr *sad, int *sal);
```

Arguments

- `s` (input)
  The descriptor for the socket to which the peer process is connected.

- `sad` (output)
  The socket address of the peer host and port number of the process.

- `sal (input/output)`
  On input, the length, in bytes, of the space to which `sad` points. This value must be set to `sizeof(struct sockaddr)`.

  On output, the actual length, in bytes, of the returned socket address (`sad`).

Explanation

The `getpeername` function returns, in the `sad` argument, the address of the socket to which your socket is connected. You must initialize `sal` to indicate the length, in bytes, of the space to which the address points. To do so, set the value of `sal` to `sizeof(struct sockaddr)`.

On output, `sal` contains the length, in bytes, of the returned address.

Return Values

If successful, `getpeername` returns the value `0`. If unsuccessful, it returns the value `-1`.

Error Codes

If `getpeername` is unsuccessful, it sets the global variable `errno` (an `int` value) to the error code for one of the following error messages to indicate the specific error.

Error Message	Description
`EBADF`	The specified socket descriptor is invalid.
`EFAULT`	The `sad` argument points to memory in an invalid part of the process address space.
`EINTR`	A signal was caught during the `getpeername` call.
`EINVAL`	The name specified by `sad` is invalid, the socket is no longer connected, or `sal` is not equal to `sizeof(struct sockaddr)`.
`ENOBUFS`	Insufficient system resources are available to complete the request.
`ENOTCONN`	The specified socket has no connected peer.
`ENOTSOCK`	The descriptor does not refer to a socket.
`ENOTTY`	The argument `s` is a file, not a socket.
`EOPNOTSUPP`	The operation is not supported for the socket protocol.

Related Functions

See `accept`, `bind`, `getsockname`, and `socket`.

getprotobyname

Purpose

The getprotobyname function provides access to information about a protocol that has been specified by name.

Syntax

```
#include <netdb.h>

struct protoent *getprotobyname(char *name);
```

Arguments

- name (input)
  The name of the protocol about which you want information.

Explanation

The getprotobyname function opens the protocols database file in the directory (master_disk)>system>stcp (if it is not already open) and searches for an entry for a particular protocol (identified by its protocol name). If successful, the function returns a pointer to a protoent structure that contains information about the specified protocol. Unless otherwise specified, the function then closes the file.

An application can call setprotoent (specifying a nonzero integer value for the stayopen argument) before calling getprotobyname to specify that the file remain open after the function has finished executing. This allows the application to step through the remainder of the file entry by entry.

The protocols database file contains a one-line entry for each known protocol. Each entry consists of the protocol's name, its protocol number, and any aliases.

The *OpenVOS STREAMS TCP/IP Administrator's Guide* (R419) explains how to set up the protocols database file.

The `protoent` structure to which `getprotobyname` returns a pointer contains information about the specified protocol. The structure is defined (in the header file `netdb.h`) as follows:

```
struct protoent{
     char      *p_name;
     char      **p_aliases;
     int       p_proto;
};
```

A description of each member of the `protoent` structure follows.

- The `p_name` field specifies the name of the protocol.
- The `p_aliases` field specifies an array of alternative names (aliases) for the protocol. Each alias is terminated by a zero.
- The `p_proto` field specifies the protocol number. Protocol numbers are returned in network byte order.

NOTE

The `protoent` structure is static data that will be overwritten by subsequent calls.

Return Values

If successful, `getprotobyname` returns a pointer to a `protoent` structure that contains information about the specified protocol. If unsuccessful (an end-of-file condition exists or an error exists), it returns a `NULL` pointer.

Error Codes

None.

Related Functions

See `endprotoent`, `getprotobynumber`, `getprotoent`, and `setprotoent`.

getprotobynumber

Purpose

The `getprotobynumber` function provides access to information about a protocol that has been specified by protocol number.

Syntax

```
#include <netdb.h>

struct protoent *getprotobynumber(int proto);
```

Arguments

- `proto` (input)
  The protocol number of the protocol about which you want information.

Explanation

The `getprotobynumber` function opens the `protocols` database file in the directory `(master_disk)>system>stcp` (if it is not already open) and searches for an entry for a particular protocol (identified by its protocol number). If successful, the function returns a pointer to a `protoent` structure that contains information about the specified protocol. Unless otherwise specified, the function then closes the file.

An application can call `setprotoent` (specifying a nonzero integer value for the `stayopen` argument) before calling `getprotobynumber` to specify that the file remain open after the function has finished executing. This allows the application to step through the remainder of the file entry by entry.

The `protocols` database file contains a one-line entry for each known protocol. Each entry consists of the protocol's name, its protocol number, and any aliases.

The *OpenVOS STREAMS TCP/IP Administrator's Guide* (R419) explains how to set up the `protocols` database file.

The `protoent` structure to which `getprotobynumber` returns a pointer contains information about the specified protocol. See the description of the `getprotobyname` function for more information about the `protoent` structure.

Return Values

If successful, `getprotobynumber` returns a pointer to a `protoent` structure containing information about the specified protocol. If unsuccessful (an end-of-file condition exists or an error exists), it returns a `NULL` pointer.

Error Codes

None.

Related Functions

See `endprotoent`, `getprotobyname`, `getprotoent`, and `setprotoent`.

getprotoent

Purpose

The getprotoent function provides access to information about any protocol for which there is an entry in the protocols database file. (The protocols database file resides in the directory (master_disk)>system>stcp.)

Syntax

```
#include <netdb.h>

struct protoent *getprotoent();
```

Arguments

The getprotoent function has no arguments.

Explanation

The getprotoent function opens the protocols database file in the directory (master_disk)>system>stcp (if it is not already open), reads an entry from the file, and returns a pointer to a protoent structure that contains information about the protocol described in that entry. The function does not close the file upon completion.

The entry that getprotoent reads from the file depends on whether the protocols database file is open when the function is called. If the file is not open, the function reads the first entry. If the file is already open, the function reads the next entry after the current file position. Therefore, an application can call getprotoent repeatedly to step through the file entry by entry.

An application can call setprotoent before calling getprotoent to guarantee that the file marker is set to the beginning of the file. However, it is redundant to call setprotoent and specify a nonzero integer value for the stayopen argument to specify that the file remain open when getprotoent has finished executing.

The protocols database file contains a one-line entry for each known protocol. Each entry consists of the protocol's name, its protocol number, and any aliases.

The *OpenVOS STREAMS TCP/IP Administrator's Guide* (R419) explains how to set up the `protocols` database file.

The `protoent` structure to which `getprotoent` returns a pointer contains information about the specified protocol. See the description of the `getprotobyname` function for more information about the `protoent` structure.

Return Values

If successful, `getprotoent` returns a pointer to the `protoent` structure associated with the entry read by the function. If unsuccessful (an end-of-file condition exists or an error exists), it returns a `NULL` pointer.

Error Codes

None.

Related Functions

See `endprotoent`, `getprotobyname`, `getprotobynumber`, and `setprotoent`.

getservbyname

Purpose

The `getservbyname` function provides access to information about a service that has been specified by its service name and the name of the associated protocol.

Syntax

```
#include <netdb.h>

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

Arguments

- `name` (input)
  The name of the service about which you want information.

- `proto` (input)
  The name of the protocol associated with the service about which you want information.

Explanation

The `getservbyname` function opens the `services` database file in the directory `(master_disk)>system>stcp` (if it is not already open) and searches for an entry for a particular service (identified by its service name and the name of the protocol that must be used when contacting the service). If successful, the function returns a pointer to a `servent` structure that contains information about the specified service. Unless otherwise specified, the function then closes the file.

The `services` database file contains a one-line entry for each known service. Each entry consists of the service's name, its port number, the name of the associated protocol, and any aliases.

The *OpenVOS STREAMS TCP/IP Administrator's Guide* (R419) explains how to set up the `services` database file.

The servent structure to which getservbyname returns a pointer contains information about the specified service. The structure is defined (in the header file netdb.h) as follows:

```
struct servent{
     char     *s_name;
     char     **s_aliases;
     long int s_port;
     char     s_proto;
};
```

A description of each member of the servent structure follows.

- The s_name field specifies the name of the service.
- The s_aliases field specifies an array of alternative names (aliases) for the service. Each alias is terminated by a zero.
- The s_port field specifies the port number at which the service is located. Port numbers are returned in network byte order.
- The s_proto field specifies the name of the protocol associated with the service.

NOTE

The servent structure is static data that will be overwritten by subsequent calls.

Return Values

If successful, getservbyname returns a pointer to a servent structure that contains information about the specified service. If unsuccessful (an end-of-file condition exists or an error exists), it returns a NULL pointer.

Error Codes

None.

Related Functions

See endservent, getservbyport, getservent, and setservent.

getservbyport

Purpose

The getservbyport function provides access to information about a service that has been specified by its port number and the name of the associated protocol.

Syntax

```
#include <netdb.h>

struct servent *getservbyport(int port, char *proto);
```

Arguments

- port (input)
  The port number of the service about which you want information.

- proto (input)
  The name of the protocol associated with the service about which you want information.

Explanation

The getservbyport function opens the services database file in the directory (master_disk)>system>stcp (if it is not already open) and searches for an entry for a particular service (identified by its port number and the name of the protocol that must be used when contacting the service). If successful, the function returns a pointer to a servent structure that contains information about the specified service. Unless otherwise specified, the function then closes the file.

An application can call setservent (specifying a nonzero integer value for the stayopen argument) before calling getservbyport to specify that the file remain open after the function has finished executing. This allows the application to step through the remainder of the file entry by entry.

The services database file contains a one-line entry for each known service. Each entry consists of the service's name, its port number, the name of the associated protocol, and any aliases.

The *OpenVOS STREAMS TCP/IP Administrator's Guide* (R419) explains how to set up the `services` database file.

The `servent` structure to which `getservbyname` returns a pointer contains information about the specified service. See the description of the `getservbyname` function for more information about the `servent` structure.

Return Values

If successful, `getservbyport` returns a pointer to a `servent` structure that contains information about the specified service. If unsuccessful (an end-of-file condition exists or an error exists), it returns a `NULL` pointer.

Error Codes

None.

Related Functions

See `endservent`, `getservbyname`, `getservent`, and `setservent`.

getservent

Purpose

The getservent function provides access to information about any service for which the services database file contains an entry.

Syntax

```
#include <netdb.h>

struct servent *getservent();
```

Arguments

The getservent function has no arguments.

Explanation

The getservent function opens the services database file (if it is not already open), reads an entry from the file, and returns a pointer to a servent structure containing information about the service described in that entry. (The services database file resides in the directory (master_disk)>system>stcp.)

The entry that getservent reads from the file depends on whether the services database file is open when the function is called. If the file is not open, the function reads the first entry. If the file is already open, the function reads the next entry after the current file position. Therefore, an application can call getservent repeatedly to step through the file entry by entry.

An application can call setservent before calling getservent to guarantee that the file marker is set to the beginning of the file. However, it is redundant to call setservent and specify a nonzero integer value for the stayopen argument to specify that the file remain open when getservent has finished executing.

The services database file contains a one-line entry for each known service. Each entry consists of the service's name, its port number, the name of the associated protocol, and any aliases.

The *OpenVOS STREAMS TCP/IP Administrator's Guide* (R419) explains how to set up the `services` database file.

The `servent` structure to which `getservbyname` returns a pointer contains information about the specified service. See the description of the `getservbyname` function for more information about the `servent` structure.

Return Values

If successful, `getservent` returns a pointer to the `servent` structure associated with the entry read by the function. If unsuccessful (an end-of-file condition exists or an error exists), it returns a `NULL` pointer.

Error Codes

None.

Related Functions

See `endservent`, `getservbyname`, `getservbyport`, and `setservent`.

get_socket_event

Purpose

The get_socket_event function returns the event ID and event count associated with a specified socket. The get_socket_event function is provided for compatibility with the OS TCP/IP API, to facilitate the migration of applications from the OS TCP/IP API to the STCP API.

NOTE

This function is provided for OS TCP/IP compatibility. If you are writing new code, you should use the poll or select function.

Usage

```
int get_socket_event(int s, int *event_id, int *event_count);
```

Arguments

- s (input)
  A descriptor for a socket that the calling process created using the accept, receive_socket, or socket function. The descriptor is returned by the function when the process creates the socket.

- event_id (output)
  A pointer to an operating system event ID. On output, event_id contains the operating system event ID for the specified socket.

- event_count (output)
  A pointer to an operating system event count. On output, event_count contains the count associated with the operating system event ID for the specified socket.

Explanation

A socket event (a type of operating system event) is allocated whenever a socket is created. The socket event becomes invalid when the STCP protocol driver closes the

socket. This occurs at some point after the application closes the socket; the exact time depends on protocol considerations. Because the application cannot know exactly when the protocol driver closes the socket, the socket event **must** be considered invalid as soon as the application closes the socket. Referring to the socket event after the socket has been closed may produce unpredictable and undesirable results.

The STCP protocol driver notifies socket events if any of the following conditions exist.

- Data can be sent to the socket and the socket was previously blocked for sending.
- Data can be received from the socket and the socket was previously blocked for receiving.
- A connection has been accepted on the socket.
- A connection has been completed for the socket.
- A connection has been disconnected for the socket.
- An error has occurred on the socket.

NOTE

The notification process does not indicate which of the preceding conditions exists. Therefore, the application can determine which condition exists by calling one of the following functions.

```
accept
recvfrom
recvmsg
send
sendmsg
sendto
writev
```

The socket event information obtained with the `get_socket_event` function can be used in any of the operating system subroutine calls that relate to events. For example, socket events, which can be combined with other operating system events, allow an application to call the operating system subroutine `s$wait_event` (rather than the networking functions `select` or `select_with_events`) to wait for notification of socket and other events.

NOTES

1. An application **must not** call the operating system subroutine `s$notify_event` for socket events; doing so interferes with the operation of the STCP protocol driver.

2. Stratus recommends that an application not call the operating system subroutine `s$read_event`; a subsequent call to the operating system subroutine `s$wait_event` might hang the application. The behavior of an application that manipulates socket events, either by performing event notification or by reading event counts, will be affected and the results will be unpredictable and undesirable.

If you use `s$wait_event`, you must confirm that your program deals properly with spurious notifies. A *spurious notify* occurs when control is returned to a program even though the event that the program is waiting for has not occurred. You should always code for the possibility of a spurious notify, and then the program can continue to wait using the `s$wait_event` subroutine. Because of the STREAMS environment of STCP, spurious notifies are more common than with OS TCP/IP, and, thus, converted programs may fail when they had not failed in OS TCP/IP.

To check for the possibility of spurious notifies, write the program to confirm that the event it is waiting for has actually occurred (that is, allow for an error on the subsequent call), and then to return to call the `s$wait_event` subroutine if the event has not occurred.

Return Values

If successful, `get_socket_event` returns the value `0`. If unsuccessful, it returns `-1`.

Error Codes

If `get_socket_event` is unsuccessful, it sets the global variable `errno` (an `int` value) to the error code for one of the following error names to indicate the specific error. (OpenVOS-style error codes are returned for compatibility with the OS TCP/IP application programming interface.)

Error Message	Description
`e$invalid_arg`	The address specified by `event_id` or `event_count` is invalid.
`e$invalid_socket`	The specified socket is invalid.
`ENOTCONN`	The connection peer has closed its socket.

Related Functions

See `select` and `select_with_events`. See the OpenVOS Subroutines manuals for additional information about the `poll` function and the *OpenVOS Standard C Reference Manual* (R363) for information about the `select` function.

getsockname

Purpose

The `getsockname` function returns the address (that is, the local IP address and port number) associated with the specified socket.

Syntax

```
#include <sys/socket.h>
int getsockname(int s, struct sockaddr *sad, int *sal);
```

Arguments

- `s` (input)
  A descriptor for a socket that the calling process created using the `accept`, `receive_socket`, or `socket` function. The descriptor is returned by the function when the process creates the socket.

- `sad` (output)
  The host's address.

- `sal` (input/output)
  On input, the length, in bytes, of the space to which `sad` points. This value must be set to `sizeof(struct sockaddr)`.

  On output, the actual length, in bytes, of the returned IP address and port number (`sad`).

Explanation

The `getsockname` function returns, in the `sad` argument, the IP address and port number of the specified socket. You must initialize `sal` to indicate the length, in bytes, of the space to which `sad` points. To do so, set the value of `sal` to `sizeof(struct sockaddr)`.

On output, `sal` contains the length, in bytes, of the returned IP address and port number.

Return Values

If successful, `getsockname` returns the value `0`. If unsuccessful, it returns the value `-1`.

Error Codes

If `getsockname` is unsuccessful, it sets the global variable `errno` (an `int` value) to the error code for one of the following error messages to indicate the specific error.

Error Message	Description
`EBADF`	The specified socket descriptor is invalid.
`EFAULT`	The `sad` argument points to memory that is in an invalid part of the process address space.
`EINTR`	A signal was caught during the `getsockname` call.
`EINVAL`	The name specified by `sad` is invalid, the socket is no longer connected, or `sal` is not equal to `sizeof(struct sockaddr)`.
`ENOBUFS`	Insufficient system resources are available to complete the request.
`EOPNOTSUPP`	The operation is not supported by the socket protocol.

Related Functions

See `bind`, `getpeername`, and `socket`.

getsockopt

Purpose

The getsockopt function obtains information about an option associated with a specified socket.

Syntax

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

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

Arguments

- s (input)
  A descriptor for a socket that the calling process created using the accept, receive_socket, or socket function. The descriptor is returned by the function when the process creates the socket. (output)

- level (input)
  The protocol level at which the option specified by optname exists. The following values are possible.

  - SOL_SOCKET, which indicates a socket-level (application-level) option. This value is defined in the header file sys/socket.h as the integer value associated with the socket level.
  - IPPROTO_IP, which indicates an IP-level option. This value is defined in the header file netinet/in.h.
  - IPPROTO_TCP, which indicates a TCP-level option that is used only with TCP_NODELAY.

- optname (input)
  The name of the option whose option value you want to obtain. Options are available at both the socket level and the IP level. See the Explanation for descriptions of these options.

- `optval` (output)
  The name of a buffer in which the option value of the specified option is stored. See the Explanation for a list of the option values associated with specific options.

- `optlen` (input/output)
  On input, the length, in bytes, of the space to be initialized for `optval`. On output, the actual length, in bytes, of the option value.

Explanation

The `getsockopt` function obtains information about an option associated with a specified socket. Options are available at the socket level and the IP level. Each option name is defined in `sys/socket.h` as an integer value associated with the option.

The Socket-Level Options

To obtain information about an option associated with a specified socket, the calling process passes a pointer to the option value. An option value can be, for example, a single byte integer, an IP address, a linger structure, or a binary string. Table 5-3 describes the socket-level options and lists the data type of the values.

Table 5-3. The `getsockopt` Socket-Level Options *(Page 1 of 2)*

Option	Data Type	Description
`SO_BROADCAST`	`int`	A nonzero value indicates that the calling process sends broadcast datagrams on the specified socket. The value `0` indicates the option is off.
`SO_DEBUG`	`int`	A nonzero value indicates that debugging is enabled in the underlying protocol modules. Debugging information is displayed on the system console. The value `0` indicates the option is off.
`SO_ERROR`	`int`	Reports and then clears the value of the last setting of `errno` by the protocol driver associated with this socket.
`SO_KEEPALIVE`	`struct linger`	Indicates that keepalive functionality is enabled and a keepalive time interval is set. For information, see "The SO_KEEPALIVE Option" on page 3-4.
`SO_LINGER`	`struct linger`	Indicates that lingering is enabled and a time-out period, known as a linger interval, is set. For information, see "The SO_LINGER Option" on page 3-8.
`SO_NODELAY`	`int`	A nonzero value indicates that the socket forwards data expediently. The value `0` indicates the option is off.
`TCP_NODELAY`	`int`	Identical to `SO_NODELAY` except that `TCP_NODELAY` uses the `IPPROTO_TCP` level.

Table 5-3. The `getsockopt` Socket-Level Options *(Page 2 of 2)*

Option	Data Type	Description
`SO_RCVBUF`	`int`	A nonzero value indicates the size of the receive buffer.
`SO_REUSEADDR`	`int`	A nonzero value indicates that the rules used to validate addresses supplied in a `bind` call permit reuse of local addresses. The value `0` indicates the option is off. For information, see "The SO_REUSEADDR Option" on page 3-8.
`SO_SNDBUF`	`int`	For UDP, a nonzero value indicates the maximum amount of data that the UDP driver can buffer on output. By default, the size is 98,304 bytes, allowing for about 51 maximum-size packets. If datagrams are sent faster than the application can process them, one of the following situations occurs when the limit set by `SO_SNDBUF` is exceeded: – If the socket is set for blocking mode, the application blocks datagrams. – If the socket is set for nonblocking mode, STCP sets `errno` to `EWOULDBLOCK` (or the equivalent value `EAGAIN`). For TCP, a nonzero value indicates the maximum amount of outgoing data that TCP buffers. TCP does not drop packets when the send buffer is full; instead, TCP blocks output on the sending socket. The default buffer size is 16K bytes. The value `0` indicates the option is off.
`SO_URGENT`	`int`	Indicates that the socket receives urgent data. This option is on by default, and remains on, regardless of its setting.
`_TCP_STDURG`	`int`	A nonzero value indicates that the urgent pointer sent to and received from the client points at the last octet of urgent data, as required by RFC 1122. The value `0` indicates that the urgent pointer points at the first octet after the urgent data, which is the historical implementation.
`_TCP_SHARED_DYNAMIC_PORT`		A nonzero value specifies that, when binding, STCP does not assign a local port to the socket, which typically occurs automatically when binding. Instead, when connecting, STCP assigns to the socket a local port from a range of numbers for shared dynamic ports. STCP assigns the local port number, taking into account the local address and local port as well as the remote address and remote port. Thus, STCP can reuse the same local port number with a different remote address; in fact, STCP can use the same local port number with hundreds of thousands of different peers.

The IP-Level Options

Table 5-4 summarizes the options available at the IP level.

Table 5-4. The `getsockopt` IP-Level Options

Option	Setting	Description
`IP_ADD_MEMBERSHIP`	Settable (IP address)	Indicates that the socket at the specified IP address allows multicast group reception for the socket and interface.
`IP_DROP_MEMBERSHIP`	Settable (IP address)	Indicates that the socket at the specified IP address does not allow multicast group reception for the socket and interface.
`IP_MULTICAST_IF`	Settable (IP address)	Indicates the default interface socket for multicast transmission is that of the specified source address, or clears the interface if `INADDR_ANY` is specified.
`IP_MULTICAST_LOOP`	Toggled (`unsigned char`)	Indicates that the loopback of transmitted multicast messages on the socket is disabled. That is, typically, sockets registered for a multicast on the same machine as the sender will receive the multicast; this option disables this feature.
`IP_MULTICAST_TTL`	Settable (`unsigned char`)	Indicates that the socket allows time-to-live (TTL) for outgoing multicast datagrams. See Table 5-11 for information about TTL values.
`IP_TTL`	Settable (`unsigned char`)	Indicates that the socket allows TTL for outgoing unicast datagrams and broadcast datagrams. Specify a large value; the default is `240`.

Return Values

If successful, `getsockopt` returns the value `0`. If unsuccessful, it returns the value `-1`.

Error Codes

If `getsockopt` is unsuccessful, it sets the global variable `errno` (an `int` value) to the error code for one of the following error messages to indicate the specific error.

Error Message	Description
`EBADF`	The specified socket descriptor is invalid.
`EFAULT`	The address pointed to by `optval` or the value specified by `optlen` is not in a valid part of the process address space.

Error Message	Description
EINTR	A signal was caught during the getsockopt call.
EINVAL	The value specified by optname is invalid, or the address specified by optval is invalid.
ENOBUFS	The system cannot perform the operation due to insufficient resources.
ENOPROTOOPT	The option is not supported by the protocol.
ENOTSOCK	The socket descriptor does not refer to a socket.

Related Functions

See setsockopt and socket.

htonl

Purpose

The `htonl` function converts a long (32-bit) value from host byte order to network byte order.

Syntax

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

unsigned long htonl(unsigned long hostlong);
```

Arguments

- `hostlong` (input)
  A long value (such as an Internet address) that is in host byte order.

Return Values

The `htonl` function returns a 32-bit value in network byte order, in an unsigned long integer.

Error Codes

None.

Related Functions

See `htons`, `ntohl`, and `ntohs`.

htons

Purpose

The `htons` function converts a short (16-bit) value from host byte order to network byte order.

Syntax

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

unsigned short htons (unsigned short hostshort);
```

Arguments

- `hostshort` (input)
  A short value (such as a port number) that is in host byte order.

Return Values

The `htons` function returns a 16-bit value in network byte order, in an unsigned short integer.

Error Codes

None.

Related Functions

See `htonl`, `ntohl`, and `ntohs`.

inet_addr

Purpose

The inet_addr function converts a character string representing the dot-notation form of an AF_INET address to an unsigned long integer containing the address.

NOTE

Do not use the inet_addr function in new code because the calling program cannot distinguish between failure and a local broadcast address. Instead, use the inet_aton function.

Syntax

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

unsigned long inet_addr(char *cp);
```

Arguments

- cp (input)
  A character string representing the dot-notation form of an AF_INET address. You can specify any of the formats shown in Table 5-5.

Table 5-5. Formats for the cp Argument of the inet_addr Function *(Page 1 of 2)*

Format	Description
a.b.c.d	The values *a*, *b*, *c*, and *d* correspond to the first four bytes of an AF_INET address, in network byte order (with the high-order byte first).
a.b.c	The values *a* and *b* correspond to the first two bytes of an AF_INET address. The value *c* represents a 16-bit string that fills the two low-order bytes of that address. This format is convenient for specifying Class B addresses.

Table 5-5. Formats for the `cp` Argument of the `inet_addr` Function *(Page 2 of 2)*

Format	Description
a.b	The value *a* corresponds to the first byte of an `AF_INET` address. The value *b* represents a 24-bit string that fills the three low-order bytes of that address. This format is convenient for specifying Class A addresses.
a	The value *a* represents a 32-bit string that the function accepts directly as the `AF_INET` address.

An example of the *a.b.c.d* format shown in Table 5-5 is `128.121.1.61`. An example of the *a.b.c* format is `128.121.317`.

You can supply decimal, octal, or hexadecimal values for *a*, *b*, *c*, and *d*. By default, the value is interpreted in decimal. To indicate an octal value, prefix the byte with a zero. To indicate a hexadecimal value, prefix the byte with `0x` or `0X`.

Return Values

If successful, `inet_addr` returns an unsigned long integer that contains the address. If unsuccessful, it returns the value `-1`.

Error Codes

None.

Related Functions

See `inet_aton`, `inet_lnaof`, `inet_makeaddr`, `inet_netof`, `inet_network`, `inet_ntoa`, `inet_ntop`, and `inet_pton`.

inet_aton

Purpose

The inet_aton function converts an Internet address from a character string to numeric form.

Syntax

```
#include <arpa/inet.h>

int inet_aton (const char *cp, struct in_addr *addr);
```

Arguments

- cp (input)
  A character string representing the dot-notation form of an AF_INET address. You can specify any of the formats shown in Table 5-5.

Table 5-6. Formats for the cp Argument of the inet_addr Function

Format	Description
a.b.c.d	The values *a*, *b*, *c*, and *d* correspond to the first four bytes of an AF_INET address, in network byte order (with the high-order byte first).
a.b.c	The values *a* and *b* correspond to the first two bytes of an AF_INET address. The value *c* represents a 16-bit string that fills the two low-order bytes of that address. This format is convenient for specifying Class B addresses.
a.b	The value *a* corresponds to the first byte of an AF_INET address. The value *b* represents a 24-bit string that fills the three low-order bytes of that address. This format is convenient for specifying Class A addresses.
a	The value *a* represents a 32-bit string that the function accepts directly as the AF_INET address.

An example of the *a.b.c.d* format shown in Table 5-5 is 128.121.1.61. An example of the *a.b.c* format is 128.121.317.

You can supply decimal, octal, or hexadecimal values for *a*, *b*, *c*, and *d*. By default, the value is interpreted in decimal. To indicate an octal value, prefix the byte with a zero. To indicate a hexadecimal value, prefix the byte with `0x` or `0X`.

- `addr` (output)
  A pointer to the structure that is to receive the converted Internet host address.

Explanation

The `inet_aton` function converts the string pointed to by `cp`, in Internet standard host format, to numeric form. This function is similar to the `inet_addr` function, except that `inet_aton` returns a code indicating success or failure, rather than returning the address itself. The following program provides an example of using the `inet_aton` function.

```
#include <arpa/inet.h>
#include <stdio.h>
#include <string.h>

int main ()
{
struct in_addr dst;
char src[16];
int r;

     strcpy (src, "192.168.255.255");
     r = inet_aton (src, &dst);
     if (r == 0)
          printf ("inet_aton failed. %s\n", src);
     else printf ("0x%x\n", dst.s_addr);

     strcpy (src, "192.168.");
     r = inet_aton (src, &dst);
     if (r == 0)
          printf ("inet_aton failed. %s\n", src);
     else printf ("0x%x\n", dst.s_addr);

     return 0;
}
```

Return Values

The `inet_aton` function returns `1` if the conversion succeeds and `0` if the conversion fails.

Error Codes

None.

Related Functions

See inet_addr, inet_lnaof, inet_makeaddr, inet_netof, inet_network, inet_ntoa, inet_ntop, and inet_pton.

inet_lnaof

Purpose

The inet_lnaof function extracts a host address from an AF_INET address. It receives the address as an in_addr structure.

Syntax

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

unsigned long inet_lnaof(struct in_addr in);
```

Arguments

- in (input)

  An in_addr structure containing an AF_INET address. The address is stored as an unsigned long integer.

  The in_addr structure is defined in the header file netinet/in.h as follows:

  ```
  struct in_addr {
       unsigned long s_addr;
  };
  ```

Return Values

The inet_lnaof function returns an unsigned long integer representing the host-address portion of an AF_INET address.

Error Codes

None.

Related Functions

See inet_addr, inet_aton, inet_makeaddr, inet_netof, inet_network, inet_ntoa, inet_ntop, and inet_pton.

inet_makeaddr

Purpose

The inet_makeaddr function combines a network number and a host address to produce an AF_INET address. It receives the network number and host address as integers and returns the resulting address.

Syntax

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

struct in_addr inet_makeaddr(unsigned long net, unsigned long host);
```

Arguments

- net (input)
  An integer representation of a network number.
- host (input)
  An integer representation of a host address.

Return Values

The inet_makeaddr function returns the structure in_addr representing an AF_INET address. The inet_lnaof function shows the in_addr structure.

Error Codes

None.

Related Functions

See inet_addr, inet_aton, inet_lnaof, inet_netof, inet_network, inet_ntoa, inet_ntop, and inet_pton.

inet_netof

Purpose

The inet_netof function extracts a network number from an AF_INET address. It receives the address as an in_addr structure.

Syntax

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

unsigned long inet_netof(struct in_addr in);
```

Arguments

- in (input)
  An in_addr structure that contains an AF_INET address. The address is stored as an unsigned long integer. The inet_lnaof function shows the in_addr structure.

Return Values

The inet_netof function returns an unsigned long integer representing the network-number portion of an AF_INET address.

Error Codes

None.

Related Functions

See inet_addr, inet_aton, inet_lnaof, inet_makeaddr, inet_network, inet_ntoa, inet_ntop, and inet_pton.

inet_network

Purpose

The `inet_network` function extracts the network-number portion of a character string representing the dot-notation form of an `AF_INET` address, and returns an unsigned long integer representing the network number.

Syntax

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

unsigned long inet_network(char *cp);
```

Arguments

- `cp` (input)
  A character string representing the dot-notation form of an `AF_INET` address. You can specify any of the formats shown in Table 5-7.

Table 5-7. Formats for the `cp` Argument of the `inet_network` Function

Format	Description
a.b.c.d	The values *a*, *b*, *c*, and *d* correspond to the first four bytes of an `AF_INET` address, in network byte order (with the high-order byte first).
a.b.c	The values *a* and *b* correspond to the first two bytes of an `AF_INET` address. The value *c* represents a 16-bit string that fills the two low-order bytes of that address. This format is convenient for specifying Class B addresses.
a.b	The value *a* corresponds to the first byte of an `AF_INET` address. The value *b* represents a 24-bit string that fills the three low-order bytes of that address. This format is convenient for specifying Class A addresses.
a	The value *a* represents a 32-bit string that the function accepts directly as the `AF_INET` address.

An example of the *a.b.c.d* format shown in Table 5-7 is `128.121.1.61`. An example of the *a.b.c* format is `128.121.317`.

You can supply decimal, octal, or hexadecimal values for *a*, *b*, *c*, and *d*. By default, the value is interpreted in decimal. To indicate an octal value, prefix the byte with a zero. To indicate a hexadecimal value, prefix the byte with `0x` or `0X`.

Return Values

If successful, `inet_network` returns an unsigned long integer representing the network-number portion of an `AF_INET` address. If unsuccessful, it returns the value `-1`.

Error Codes

None.

Related Functions

See `inet_addr`, `inet_aton`, `inet_lnaof`, `inet_makeaddr`, `inet_netof`, `inet_ntoa`, `inet_ntop`, and `inet_pton`.

inet_ntoa

Purpose

The inet_ntoa function converts an AF_INET address in an in_addr structure to a character string representing the dot-notation form of the address.

Syntax

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

char *inet_ntoa(struct in_addr in);
```

Arguments

- in (input)
  An in_addr structure containing an AF_INET address. The address is stored as an unsigned long integer. The inet_lnaof function shows the in_addr structure.

Return Values

The inet_ntoa function returns a character string representing the dot-notation form of an AF_INET address.

Error Codes

None.

Related Functions

See inet_addr, inet_aton, inet_lnaof, inet_makeaddr, inet_netof, inet_network, inet_ntop, and inet_pton.

inet_ntop

Purpose

The `inet_ntop` function converts an Internet address from numeric form to presentation form.

Syntax

```
#include <arpa/inet.h>
#include <sys/socket.h>

const char *inet_ntop (int af, const void *src, char *dst, socklen_t size);
```

Arguments

- `af` (input)
  The address format to be used. The value of `af` must be `AF_INET` to indicate the `AF_INET` address family. The `AF_INET` address family supports IPv4 Internet communications.

- `src` (input)
  A pointer to the Internet address being passed to the function.

- `dst` (output)
  A pointer to the buffer that is to receive the presentation form of the Internet address.

- `size` (input)
  The length, in bytes, of the return buffer.

Explanation

The inet_ntop function converts the Internet host address specified by src to a string in Internet standard dot notation. The following program provides an example of using the inet_ntop function; the program displays the value 192.168.255.255.

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

int main ()
{
int address;
char dst[16];
char *p;

     address = 0x0C0A8FFFF;
     p = (char *) inet_ntop (AF_INET, &address, dst,
          sizeof dst);

     if (p == NULL)
          printf ("inet_ntop failed. errno=%d\n", errno);
     else printf ("%s\n", p);

     return 0;
}
```

Return Values

If successful, inet_ntop returns a pointer to the converted value. If unsuccessful, inet_ntop returns a NULL pointer and sets errno to a nonzero value.

Error Codes

If inet_ntop is unsuccessful, it sets the variable errno (an int value) to a nonzero value, to indicate the specific error.

Error Codes	Description
ENOAFSUPPORT	An address family other than AF_INET was specified.
ENOSPC	Insufficient room in the buffer to return the entire string.

Related Functions

See `inet_addr`, `inet_aton`, `inet_lnaof`, `inet_makeaddr`, `inet_netof`, `inet_network`, `inet_ntoa`, and `inet_pton`.

inet_pton

Purpose

The `inet_pton` function converts an Internet host address from presentation form to numeric form.

Syntax

```
#include <arpa/inet.h>
#include <sys/socket.h>

int inet_pton (int af, const char *src, void *dst);
```

Arguments

- `af` (input)
  The address format to be used. The value of `af` must be `AF_INET` to indicate the `AF_INET` address family. The `AF_INET` address family supports IPv4 Internet communications.

- `src` (input)
  A pointer to the Internet address being passed to the function.

- `dst` (output)
  A pointer to the buffer that is to receive the presentation form of the Internet address.

Explanation

The `inet_pton` function converts the string pointed to by `src`, in Internet standard dot notation, to an integer value. The following program provides an example of using the `inet_pton` function; the program displays the value `0xc0a8ffff`.

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

int main ()
{
int dst;
char src[16];
int r;

     strcpy (src, "192.168.255.255");

     r = inet_pton (AF_INET, src, &dst);
     if (r <= 0)
          printf ("inet_pton failed. errno=%d\n", errno);
     else printf ("0x%x\n", dst);

     return 0;
}
```

Return Values

The `inet_pton` function returns the value `1` if successful, `0` for an unparseable address, and `-1` if an invalid address family was specified.

Error Codes

If `inet_pton` returns the value `0`, `errno` is **not** set.

Related Functions

See `inet_addr`, `inet_aton`, `inet_lnaof`, `inet_makeaddr`, `inet_netof`, `inet_network`, `inet_ntoa`, and `inet_ntop`.

ioctl

Purpose

The `ioctl` function obtains and controls characteristics of devices. When used with a socket, it provides information about network interfaces. (This function is provided by the OpenVOS Standard C library. For additional information about the `ioctl` function, see the *OpenVOS Communications Software: STREAMS Programmer's Guide* (R306).)

Syntax

```
#include <net/if.h>
#include <ioclt.h>
#include <net/if_flags.h> /* optionally if SIOCGIFFLAGS is used */

int ioctl(int s, int request, void *arg);
```

Arguments

- `s` (input)
  A descriptor for a socket that the calling process created with the `socket` function. The descriptor is returned by the function when the process creates the socket. The socket must be of the type `SOCK_DGRAM` or `SOCK_RAW`.

- `request` (input)
  Specifies the `ioctl` control operation.

- `arg` (input/output)
  A pointer to a structure containing values that are required for input and returned from the specific request as output.

Explanation

The `SIOCGIFCONF` request of the `ioctl` function returns information about all interfaces. The other requests return specific information about a particular interface (its net address, broadcast address, subnet mask, or values of flags). Table 5-8 lists and briefly describes these requests. (The structures `ifconf` and `ifreq` are defined

in the if.h header file in the (master_disk)>system>stcp>include_library>net directory.)

Table 5-8. Interface-Operation Requests of the ioctl Function

Requests	Description	Data Type
SIOCGIFCONF	Get list of all interfaces.	struct ifconf
SIOCGIFADDR	Get interface address.	struct ifreq
SIOCGIFFLAGS	Get interface flags.	struct ifreq
SIOCGIFBRDADDR	Get broadcast address.	struct ifreq
SIOCGIFNETMASK	Get subnet mask.	struct ifreq

The SIOCGIFCONF request returns a list of the names of all interfaces and their associated Internet addresses. If an interface has aliases, its name appears multiple times with different addresses. The caller passes a pointer to the ifconf structure, which is defined in the <net/if.h> header file, as follows:

```
struct ifconf {
    int  ifc_len;       /* size of associated buffer */
    union {
           char     *ifcu_buf;
           struct    ifreq *ifcu_req;
    } ifc_ifcu;
```

This structure contains a pointer to an ifreq structure (typically, an array of ifreq structures), also defined in the <net/if.h> header file. This structure is used as an argument to the other ioctl requests. The relevant lines of the declaration are as follows:

```
struct ifreq {
    char ifr_name[IFNAMSIZ]; /* if name,
                              e.g.”%sys#sdlmuxB.m102.11.1” */
    union {
           struct    sockaddr ifru_addr;
           struct    sockaddr ifru_broadaddr;
           short     ifru_flags;
    } ifr_ifru;
```

IFNAMSIZ is defined as 68. If it is predefined to a different value in the source code prior to where this header file is included, then no error occurs, but the ioctl requests cannot be used (their definitions will not be visible).

To use SIOCGIFCONF, you must first approximate how many interfaces (and aliases) you typically expect, and allocate an array containing that number of ifreq structures (for example, 20 or 30). Set ifcu_req to point to that array, and set ifc_len to the length of that array, in bytes.

On output, the array is set and the ifc_len value in the ifconf structure is set to the actual number of bytes moved to the array. If more entries exist than can fit in the provided space, EINVAL is returned, but the information in the array is set, regardless, and the value of ifc_len is not changed (that is, the value remains at the maximum value set by the caller, which was not large enough). You can work with the set of interfaces given, or re-execute the ioctl function, allocating a larger space. Note that ifc_len is in bytes, not entries. One entry occupies 84 bytes. It is not necessary to provide space in terms of an integral number of entries, but no space over an integral number will be changed. For example, if you specify 167 as the length and there is one interface, the first 84 bytes are set and ifc_len is set to 84. If two interfaces exist, then 84 bytes are set, ifc_len is left at 167, and EINVAL is returned.

You can use the information in the ifreq array to get further information about a specific interface, or if you know the interface name, you can fill in an ifreq structure to obtain information. The ifreq structure has a second component that is a union and thus can be used to represent data of different types. You can get 16-bit flag data or either mask or address data in a sockaddr structure. The if_flags.h header file contains constants that you can use to interpret flag bits (for example, IFF_MULTICAST, IFF_UP, IFF_NOKEEPALIVE, and so on).

Return Values

If successful, ioctl returns the value 0. If unsuccessful, it returns the value -1.

Error Codes

If ioctl is unsuccessful, it sets the global variable errno (an int value) to the error code for one of the following error messages to indicate the specific error.

Error Message	Description
EBADF	The specified socket descriptor is invalid.
EINVAL	The value specified by request is invalid (for example, the specified interface does not exist) or, for the SIOCGIFCONF request, insufficient space has been provided to return the names and addresses of all interfaces.
ENOBUFS	The system cannot perform the operation due to insufficient resources.
ENOPROTOOPT	The option is not supported by the protocol. Specifically, one of the ioctl values has been used with a SOCK_STREAM socket.

listen

Purpose

The listen function listens for connection requests on a specified socket.

Syntax

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

Arguments

- s (input)
  A descriptor for a socket that the calling process created using the socket function. The descriptor is returned by the function when the process creates the socket.

- backlog (input)
  The maximum number of pending requests permitted on the connection queue for the socket(s). The maximum value is 1024. A value less than 0 has the same effect as the value 0.

Explanation

The listen function prepares a socket to accept connections.

A server process that wants to accept connections creates a socket with the socket function and then binds it to an IP address and port number with the bind function. The server process then uses the listen function to prepare the socket to accept connections.

The calling process then calls the accept function for any incoming connection requests.

Connection requests wait in a queue to be accepted. As part of using the listen function, a calling process specifies the maximum length of this queue. If the queue is full when a connection request arrives, the peer process requesting the connection receives the error code for ECONNREFUSED.

Return Values

If successful, `listen` returns the value `0`. If unsuccessful, it returns the value `-1`.

Error Codes

If `listen` is unsuccessful, it sets the global variable `errno` (an `int` value) to the error code for one of the following error messages to indicate the specific error.

Error Message	Description
`EBADF`	The specified socket descriptor is invalid.
`EDESTADDRREQ`	The socket is not bound to a local address.
`EINTR`	A signal was caught during the `listen` call.
`EINVAL`	The socket is already connected.
`ENOTSOCK`	The argument `s` is not a socket.
`EOPNOTSUPP`	The socket is not of type `SOCK_STREAM`.

Related Functions

See `accept`, `bind`, `connect`, `select`, and `socket`.

map_stcp_error

Purpose

The `map_stcp_error` function accepts an error code returned by an STCP function and returns the equivalent OS TCP/IP error code, if one exists.

Syntax

```
#include <common_proto.h>

int map_stcp_error (int code);
```

Arguments

- `code` (input/output)
  The specified OpenVOS error code. The argument also returns the equivalent OS TCP/IP error code.

Return Values

The `map_stcp_error` function accepts an error code returned by an STCP function (that is, an STCP, STREAMS, or POSIX error code) and returns the equivalent OS TCP/IP error code, if one exists. It is designed to aid you in migrating your applications from OS TCP/IP.

Error Codes

None.

Related Functions

The `map_stcp_error` function has no related functions.

ntohl

Purpose

The `ntohl` function converts a long (32-bit) value from network byte order to host byte order.

Syntax

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

unsigned long ntohl(unsigned long netlong);
```

Arguments

- `netlong` (input)
  A long value (such as an Internet address) that is in network byte order.

Return Values

The `ntohl` function returns a 32-bit value in host byte order, in an unsigned long integer.

Error Codes

None.

Related Functions

See `htonl`, `htons`, and `ntohs`.

ntohs

Purpose

The `ntohs` function converts a short (16-bit) value from network byte order to host byte order.

Syntax

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

unsigned short ntohs(unsigned short netshort);
```

Arguments

- `netshort` (input)
  A short value (such as a port number) that is in network byte order.

Return Values

The `ntohs` function returns a 16-bit value in host byte order, in an unsigned short integer.

Error Codes

None.

Related Functions

See `htonl`, `htons`, and `ntohl`.

poll

This function is provided by the OpenVOS Standard C library. For detailed information, see the OpenVOS Subroutines manuals.

read

This function is provided by the OpenVOS Standard C library and POSIX library. For detailed information, see the *OpenVOS Standard C Reference Manual* (R363). The *OpenVOS POSIX.1 Reference Guide* (R502) also provides some information.

readv

This function is provided by the OpenVOS Standard C library and POSIX library. For detailed information, see the *OpenVOS Standard C Reference Manual* (R363). The *OpenVOS POSIX.1 Reference Guide* (R502) also provides some information.

receive_socket

Purpose

The `receive_socket` function opens a socket transferred from another process.

NOTE

This function is provided for OS TCP/IP compatibility. If you are writing new code, you should use the `fork` function.

Syntax

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

int receive_socket (char *pathname);
```

Arguments

- `pathname`
  The path name of the transferred socket.

Explanation

The `receive_socket` function opens a socket for the device specified by the `pathname` argument. The `pathname` argument must be the same value returned by the `transfer_socket` function in the sending process. You must pass the path name from the sending process to the receiving process.

After this function returns successfully, you must notify the transferring process to close its file descriptor for the socket.

Return Values

If `receive_socket` successfully opens the transferred socket, it returns a valid file descriptor for the opened socket. If it cannot open the socket, it returns the value `-1`.

Error Codes

If `receive_socket` cannot open the socket, it sets the global variable `errno` (an `int` value) to the error code that indicates the specific error. The most common error messages follow.

Error Message	Description
`EMFILE`	No more file descriptors are available for this process (a maximum of 4096 are available).
`ENOBUFS`	Insufficient system resources are available to complete the request.
`ENXIO`	The pathname used is not valid.

Access Requirements

None.

Related Functions

See `transfer_socket`.

recv

Purpose

The recv function receives data from a connection-mode or connectionless-mode socket. (The received data is sometimes referred to as a *message*.) This function is typically used with connected sockets because it does not allow the application to obtain the source address of received data.

Syntax

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

int recv(int s, char *msg, int len, int flags);
```

Arguments

- s
  A descriptor for a socket that the calling process created using the accept, receive_socket, or socket function. The descriptor is returned by the function when the process creates the socket.

- msg (output)
  The buffer that is to contain the received data.

- len (input)
  The length, in bytes, of the buffer specified as msg.

- flags (input)
  The type of message reception. The following values are possible:

  - MSG_PEEK—Specifies that the function peeks at an incoming message. The data is treated as unread and the next recv or similar function still returns this data.

- MSG_WAITALL—Specifies, for SOCK_STREAM sockets, that the function blocks until the full amount of data can be returned. The recv function may return the smaller amount of data in the following situations.
  - the socket is a stream-based socket
  - a signal is caught
  - the connection is terminated
  - MSG_PEEK was specified
  - an error is pending for the socket

Explanation

The recv function returns the length of the message written to the buffer pointed to by the msg argument. For sockets of the type SOCK_DGRAM, the entire message is read in a single operation. If the length of a message exceeds the buffer and MSG_PEEK is not specified in the flags argument, the excess bytes are discarded. For sockets of the type SOCK_STREAM, message boundaries are ignored; data is returned to the user as soon as it becomes available, and no data is discarded.

If the MSG_WAITALL flag is not set, data is returned only up to the end of the first message.

For sockets of the type SOCK_DGRAM, if no messages are available at the socket and O_NONBLOCK is not set on the socket's file descriptor, recv blocks until a message arrives. If no messages are available at the socket and O_NONBLOCK is set on the socket's file descriptor, recv fails and sets errno to EWOULDBLOCK (or the equivalent value EAGAIN).

For sockets of the type SOCK_STREAM, if no bytes are available at the socket and O_NONBLOCK is not set on the socket's file descriptor, recv blocks until some data bytes arrive. If no bytes are available at the socket and O_NONBLOCK is set on the socket's file descriptor, recv fails and sets errno to EWOULDBLOCK (or the equivalent value EAGAIN).

NOTE

You should handle urgent data by using the so_recv function. See the description of the so_recv function for more information.

Return Values

If successful, recv returns the length of the data moved to msg in bytes. No more than len bytes are moved. If len is 0 and data is available at the socket, the operation is considered successful and recv returns 0. If no data is available to be received and

the peer has performed an orderly shutdown, recv returns 0. Otherwise, recv returns -1 and sets errno to indicate the error.

Error Codes

When recv returns -1, it sets the global variable errno (an int value) to the error code for one of the following error messages to indicate the specific error.

Error Message	Description
EBADF	The specified socket descriptor is invalid.
ECONNREFUSED	The attempt to connect was forcibly rejected.
ECONNRESET	The connection associated with socket s was forcibly closed by a peer.
EFAULT	The data was specified to be received into a nonexistent or protected part of the process address space.
EINTR	The receive was interrupted by delivery of a signal before any data was available.
EINVAL	The len argument is less than zero.
EIO	An internal error has occurred.
ENOBUFS	Insufficient system resources are available to complete the request.
ENOTCONN	The requested operation was attempted on a connection-mode socket that is not connected.
ENOTSOCK	The file descriptor s is not associated with a socket.
EOPNOTSUPP	The specified flags are not supported for this socket type or protocol.
EWOULDBLOCK (or the equivalent value EAGAIN)	The file descriptor s is marked O_NONBLOCK and no data is waiting to be received.

Related Functions

See getsockopt, read, recvfrom, recvmsg, select, so_recv, and socket.

recvfrom

Purpose

The recvfrom function receives data from a connection-mode or connectionless-mode socket. (The received data is sometimes called a *message*.) The function is typically used with connectionless-mode sockets because it enables the application to retrieve the source address of the received data.

Syntax

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

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

Arguments

- s (input)
  A descriptor for a socket that the calling process created using the accept, receive_socket, or socket function. The descriptor is returned by the function when the process creates the socket.

- msg (output)
  The buffer that is to contain the received data.

- len (input)
  The length, in bytes, of the buffer specified as msg.

- flags (input)
  The type of message reception. The following values are possible:

  - MSG_BLOCKING—Specify to override any nonblocking state. This flag does not exist if the application is coded for POSIX compliance; instead, specify _VOS_MSG_BLOCKING.
  - MSG_NONBLOCKING—Specify to override any blocking state. This flag does not exist if the application is coded for POSIX compliance; instead, specify _VOS_MSG_NONBLOCKING.

- MSG_PEEK—Specifies that the function peeks at an incoming message. The data is treated as unread and the next recvfrom or similar function still returns this data.
- MSG_WAITALL—Specifies, for SOCK_STREAM sockets, that the function blocks until the full amount of data can be returned. The recvfrom function may return the smaller amount of data in the following situations.
  - the socket is a stream-based socket
  - a signal is caught
  - the connection is terminated
  - MSG_PEEK was specified
  - an error is pending for the socket

► from (output)
A null pointer or a pointer to a sockaddr structure that stores the IP address and port number of the socket from which the message was sent.

► fromlen (input/output)
The length, in bytes, of the sockaddr structure to which the from argument points.

Explanation

The recvfrom function returns the length of the message written to the buffer pointed to by the buffer argument. For message-based sockets (sockets of the type SOCK_RAW or SOCK_DGRAM), the entire message is read in a single operation. If the length of the message exceeds the buffer length, and MSG_PEEK is not set in the flags argument, the excess bytes are discarded. For sockets of the type SOCK_STREAM, message boundaries are ignored. In this case, data is returned to the user as soon as it becomes available, and no data is discarded.

If the MSG_WAITALL flag is not set, data is returned only up to the end of the first message.

Not all protocols provide the source address for messages. If the address argument is not a null pointer and the protocol provides the source address of messages, the source address of the received message is stored in the sockaddr structure to which the from argument points, and the length of this address is stored in the object to which the fromlen argument points.

If the actual length of the address is greater than the length of the supplied sockaddr structure, the stored address is truncated.

If the from argument is not a null pointer and s refers to a streams-based socket, the value stored in the object pointed to by from is not modified.

For sockets of the type SOCK_DGRAM, if no messages are available at the socket and O_NONBLOCK is not set on the socket's file descriptor, recvfrom blocks until a message arrives. If no messages are available at the socket and O_NONBLOCK is set on the socket's file descriptor, recvfrom returns -1 and sets errno to EWOULDBLOCK (or the equivalent value EAGAIN).

For sockets of the type SOCK_STREAM, if no bytes are available at the socket and O_NONBLOCK is not set on the socket's file descriptor, recvfrom blocks until some data bytes arrive. If no bytes are available at the socket and O_NONBLOCK is set on the socket's file descriptor, recvfrom returns -1 and sets errno to EWOULDBLOCK (or the equivalent value EAGAIN).

Three related functions also receive data. The recvmsg and recv functions receive data from either connected or unconnected sockets. The so_recv function receives data from connected sockets only. However, urgent data is handled using the so_recv function.

Return Values

If successful, recvfrom returns the number of bytes in the message. If unsuccessful, it returns the value -1 and sets errno to indicate the error.

Error Codes

If recvfrom is unsuccessful, it sets the global variable errno (an int value) to the error code for one of the following error messages to indicate the specific error.

Error Message	Description
EBADF	The specified socket descriptor is invalid.
ECONNRESET	The connection associated with socket s was forcibly closed by a peer.
EFAULT	The data was specified to be received into a nonexistent or protected part of the process address space.
EINTR	The receive was interrupted by delivery of a signal before any data was available.
EINVAL	The len argument is less than zero.
EIO	An internal error has occurred.
ENOBUFS	Insufficient system resources are available to complete the request.
ENOTCONN	The requested operation was attempted on a connection-mode socket that is not connected.
ENOTSOCK	The file descriptor s is not associated with a socket.
EOPNOTSUPP	The specified flags are not supported for this socket type or protocol.

Error Message	Description
EWOULDBLOCK (or the equivalent value EAGAIN)	The file descriptor s is marked O_NONBLOCK and no data is waiting to be received.

Related Functions

See getsockopt, read, recv, recvmsg, select, send, so_recv, and socket.

recvmsg

Purpose

The recvmsg function receives data from a connection-mode or connectionless-mode socket, and returns the length of the message. (The received data is sometimes called a *message*.) The function is typically used with connectionless-mode sockets because it enables the application to retrieve the source address of the received data.

Syntax

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

int recvmsg(int s, struct msghdr *msg, int flags);
```

Arguments

- s (input)
  A descriptor for a socket that the calling process created using the accept, receive_socket, or socket function. The descriptor is returned by the function when the process creates the socket.

- msg (input/output)
  A pointer to the msghdr structure that specifies the optional peer address and buffers for the received data.

- flags (input)
  The type of message reception. The following values are valid:

  - MSG_PEEK—Specifies that the function peeks at an incoming message. The data is treated as unread and the next recvmsg or similar function still returns this data.

- MSG_WAITALL—Specifies, for SOCK_STREAM sockets, that the function blocks until the full amount of data can be returned. The recvmsg function may return the smaller amount of data in the following situations.
  - the socket is a streams-based socket
  - a signal is caught
  - the connection is terminated
  - MSG_PEEK was specified
  - an error is pending for the socket

Explanation

The recvmsg function returns the total length of the message. For message-based sockets (that is, sockets of the type SOCK_DGRAM), the entire message is read in a single operation. If the size of a message exceeds the size of the supplied buffers, and MSG_PEEK is not set in the flags argument, the excess bytes are discarded and MSG_TRUNC is set in the msg_flags member of the msghdr structure. For streams-based sockets (that is, sockets of the type SOCK_STREAM), message boundaries are ignored. In this case, data is returned to the user as soon as it becomes available, and no data is discarded.

If the MSG_WAITALL flag is not set, data is returned only up to the end of the first message.

If no messages are available at the socket and O_NONBLOCK is not set on the socket's file descriptor, recvmsg blocks until a message arrives. If no messages are available at the socket and 0_NONBLOCK is set on the socket's file descriptor, recvmsg fails and sets errno to EWOULDBLOCK (or the equivalent value EAGAIN).

In the msghdr structure, the msg_name and msg_namelen members specify the source address if the socket is unconnected. If the socket is connected, the msg_name and msg_namelen members are ignored. The msg_name member may be a null pointer if no names are desired or required. The msg_iov and msg_iovlen fields are used to specify where the received data is stored. The msg_iov field points to an array of iovec structures; the msg_iovlen field is set to the dimension of this array, which is restricted to 1 when the MSG_WAITALL flag is set. In the iovec structure, the iov_base field specifies a storage area and the iov_len field gives its size in bytes. The storage area indicated by msg_iov is filled with received data.

Return Values

If successful, recvmsg returns the number of bytes in the message. If unsuccessful, it returns the value -1.

Error Codes

If recvmsg is unsuccessful, it sets the global variable errno (an int value) to the error code for one of the following error messages to indicate the specific error.

Error Message	Description
EBADF	The specified socket descriptor is invalid.
ECONNRESET	The connection associated with socket s was forcibly closed by a peer.
EFAULT	The data was specified to be received into a nonexistent or protected part of the process address space.
EINTR	The receive was interrupted by delivery of a signal before any data was available.
EINVAL	The len argument is less than zero.
EIO	An internal error has occurred.
EMSGSIZE	The msg_iovlen member of the msghdr structure to which message points is not equal to 1.
ENOBUFS	Insufficient system resources are available to complete the request.
ENOTCONN	The requested operation was attempted on a connection-mode socket that is not connected.
ENOTSOCK	The file descriptor s is not associated with a socket.
EOPNOTSUPP	The specified flags are not supported for this socket type or protocol.
EWOULDBLOCK (or the equivalent value EAGAIN)	The file descriptor s is marked O_NONBLOCK and no data is waiting to be received.

Related Functions

See getsockopt, read, recv, recvfrom, select, send, so_recv, and socket.

select

This function is provided by the OpenVOS Standard C library and POSIX library. For detailed information, see the *OpenVOS Standard C Reference Manual* (R363) and the *OpenVOS POSIX.1 Reference Guide* (R502).

select_with_events

This function is provided by the OpenVOS Standard C library and POSIX library. For detailed information, see the *OpenVOS Standard C Reference Manual* (R363) and the *OpenVOS POSIX.1 Reference Guide* (R502).

send

Purpose

The send function initiates transmission of a message from the specified socket to its peer. The send function sends a message only when the socket is connected (including when the peer of a connectionless socket has been set using connect).

Syntax

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

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

Arguments

- s (input)
  A descriptor for a socket that the calling process created using the accept, receive_socket, or socket function. The descriptor is returned by the function when the process creates the socket.

- msg (input)
  The buffer that contains the data to be copied.

- len (input)
  The length, in bytes, of the buffer specified in msg.

- flags (input)
  A flag name that specifies handling for the message being sent. Its values, which are defined in the header file sys/socket.h, are as follows:

  - MSG_OOB or 0—Specify to mark as urgent the last byte of the data being copied. It is defined as a hexadecimal value associated with this flag.
  - _VOS_MSG_PARTIAL—Specify to inform the driver that the application will be sending more data. How the driver responds to this flag is driver-specific. The driver is not required to transport the data until the application uses send without the _VOS_MSG_PARTIAL flag. The stcp driver uses this flag to optimize segmentation and to compute the TCP protocol push bit more

efficiently. The stcp driver will not time-out and will not transmit partial segments, as it does with Nagling, which is a means of improving the efficiency of TCP/IP networks. The application must send data without the partial flag in order to signal STCP that data is to be transmitted.

- MSG_BLOCKING—Specify to override any nonblocking state. This flag does not exist if the application is coded for POSIX compliance; instead, specify _VOS_MSG_BLOCKING.
- MSG_NONBLOCKING—Specify to override any blocking state. This flag does not exist if the application is coded for POSIX compliance; instead, specify _VOS_MSG_NONBLOCKING.
- _VOS_MSG_BLOCKING—Functionally equivalent to MSG_BLOCKING; use for POSIX compliance.
- _VOS_MSG_NONBLOCKING—Functionally equivalent to MSG_NONBLOCKING; use for POSIX compliance.

Explanation

The send function sends data from the buffer msg to the remote peer. The send function returns the length of the data actually copied.

Successful completion of a send call means that the message has been copied to the socket. It does not guarantee successful reception by the receiving socket. For most protocols, a returned value of -1 simply indicates that an error has been detected locally. A partial copy can occur. The number of bytes actually copied is returned. It is the application's responsibility to recopy the data that was not sent.

If a message cannot be copied immediately because the socket does not have sufficient space, the send function blocks the calling process until space is available. However, if the socket is in nonblocking mode, send returns the value -1 and sets the errno global variable to the error code for EWOULDBLOCK (or the equivalent value EAGAIN). Use the select function to determine when more data can be copied.

POSIX.1 requires that the send function raise the SIGPIPE signal in the sending thread when the function attempts to write to a SOCK_STREAM socket that has been disconnected. In releases prior to OpenVOS Release 19.1.0, the POSIX.1 runtime failed to do this and instead returned the error code ENXIO. Beginning in OpenVOS Release 19.1.0, the send function raises SIGPIPE correctly in this situation.

If you have a program that previously failed because of this situation:

- Dynamically linked programs: You do not need to do anything. The program works automatically in OpenVOS Release 19.1.0 and later.
- Statically linked programs: Rebind the program.

If this change to OpenVOS causes problems with your program, perform one of the following actions:

- In the program's initialization, call the signal function:

  ```
  signal(SIGPIPE, SIG_IGN);
  ```

  In this case, the `send` function returns the `EPIPE` error in `errno`. This is the recommended action.

- Set the `s$bug_fix_stcp_3549` variable to zero. You declare this variable as:

  ```
  extern int s$bug_fix_stcp_3549
  ```

  You can set this variable either in the program's initialization or with the `set_external_variable` command. See the *OpenVOS Commands Reference Manual* (R098) for information about `set_external_variable`.

Return Values

If successful, `send` returns the number of bytes actually copied. If unsuccessful, it returns the value `-1`.

Error Codes

If `send` is unsuccessful, it sets the global variable `errno` (an `int` value) to the error code for one of the following error messages to indicate the specific error.

Error Message	Description
`EBADF`	The specified socket descriptor is invalid.
`ECONNRESET`	A connection was forcibly closed by a peer.
`EDESTADDRREQ`	The socket is not connection-mode and no peer address is set.
`EFAULT`	An invalid user-space address was specified for a parameter.
`EINTR`	A signal was caught during the `send` call.
`EMSGSIZE`	The socket requires that messages be copied atomically (as a single unit), and the size of the message to be copied made this impossible.
`ENETDOWN`	The local network interface used to reach the destination is down.
`ENETUNREACH`	No route to the network is present.
`ENOBUFS`	Insufficient system resources are available to complete the request.
`ENOTCONN`	The socket is not connected.
`ENOTSOCK`	The descriptor does not refer to a socket.

Error Message	Description
EOPNOTSUPP	The socket argument is associated with a socket that does not support one or more of the values specified in the flags argument.
EWOULDBLOCK (or the equivalent value EAGAIN)	The requested operation would block the process.

Related Functions

See getsockopt, select, sendmsg, sendto, socket, and write.

sendmsg

Purpose

The sendmsg function copies data to either a connected socket or an unconnected socket. (The data being copied is sometimes called a *message*.)

Syntax

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

int sendmsg(int s, struct msghdr *msg, int flags);
```

Arguments

- s (input)
  A descriptor for a socket that the calling process created using the accept, receive_socket, or socket function. The descriptor is returned by the function when the process creates the socket.

- msg (input)
  A pointer to the msghdr structure, which contains both the destination address and the buffer for the outgoing message. The length and format of the address determines the address family of the socket. The msg_flags member is ignored.

- flags (input)
  A flag name that specifies handling for the message being sent. You can specify 0 or MSG_OOB to mark as urgent the last byte of the data being copied. The MSG_OOB flag name is defined in the header file sys/socket.h as a hexadecimal value associated with this flag.

  You can specify _VOS_MSG_PARTIAL (in sys/socket.h) to inform the driver that the application will be sending more data. How the driver responds to this flag is driver-specific. The driver is not required to transport the data until the application uses send without the _VOS_MSG_PARTIAL flag. The stcp driver uses this flag to optimize segmentation and to compute the TCP protocol push bit more efficiently. The stcp driver will not time-out and will not transmit partial segments, as it does with Nagling, which is a means of improving the efficiency of TCP/IP

networks. The application must send data without the partial flag in order to signal STCP that data is to be transmitted.

Explanation

The sendmsg function sends a message through a connection-mode or connectionless-mode socket. If the socket is connectionless-mode, sendmsg sends the message to the address that the msghdr structure contains. If the socket is connection-mode, sendmsg ignores the destination address in the msghdr structure.

The msg_iov and msg_iovlen fields of msg specify zero or one buffer, which contains the data to be sent. The msg_iov field points to an array of iovec structures; the msg_iovlen field shall be set to the dimension of this array, which must be 1. In the iovec structure, the iov_base field specifies a storage area, and the iov_len field specifies its size in bytes. The size can be zero. The data from the storage area indicated by msg_iov is sent.

Successful completion of a call to sendmsg does not guarantee delivery of the message. A return value of -1 indicates only locally-detected errors.

If the sending socket does not have sufficient space to hold the message to be transmitted and the socket file descriptor does not have O_NONBLOCK set, sendmsg blocks until space is available. If the sending socket does not have sufficient space to hold the message to be transmitted and the socket file descriptor has O_NONBLOCK set, sendmsg fails.

Two related functions also send data. The sendto function sends data to either connected or unconnected sockets and returns the address of the receiving socket and the length of that address. The send function sends data to connected sockets only.

POSIX.1 requires that the sendmsg function raise the SIGPIPE signal in the sending thread when the function attempts to write to a SOCK_STREAM socket that has been disconnected. In releases prior to OpenVOS Release 19.1.0, the POSIX.1 runtime failed to do this and instead returned the error code ENXIO. Beginning in OpenVOS Release 19.1.0, the sendmsg function raises SIGPIPE correctly in this situation.

If you have a program that previously failed because of this situation:

- Dynamically linked programs: You do not need to do anything. The program works automatically in OpenVOS Release 19.1.0 and later.
- Statically linked programs: Rebind the program.

If this change to OpenVOS causes problems with your program, perform one of the following actions:

- In the program's initialization, call the signal function:

  ```
  signal(SIGPIPE, SIG_IGN);
  ```

  In this case, the `send` function returns the `EPIPE` error in `errno`. This is the recommended action.

- Set the `s$bug_fix_stcp_3549` variable to zero. You declare this variable as:

  ```
  extern int s$bug_fix_stcp_3549
  ```

  You can set this variable either in the program's initialization or with the `set_external_variable` command. See the *OpenVOS Commands Reference Manual* (R098) for information about `set_external_variable`.

Return Values

If successful, `sendmsg` returns the number of bytes actually sent. If unsuccessful, it returns the value `-1`.

Error Codes

If sendmsg is unsuccessful, it sets the global variable errno (an int value) to the error code for one of the following error messages to indicate the specific error.

Error Message	Description
EBADF	The specified socket descriptor is invalid.
ECONNRESET	A connection was forcibly closed by a peer.
EDESTADDRREQ	The socket is not connection-mode and no peer address is set.
EFAULT	An invalid user-space address was specified for a parameter.
EINTR	A signal was caught during the sendmsg call.
EMSGSIZE	The socket requires that messages be copied atomically (as a single unit), and the size of the message to be copied made this impossible, or the msg_iovlen member of the msghdr structure that msg points to is not equal to 1.
ENETDOWN	The local network interface used to reach the destination is down.
ENETUNREACH	No route to the network is present.
ENOBUFS	Insufficient system resources are available to complete the request.
ENOTCONN	The socket is not connected.
ENOTSOCK	The descriptor does not refer to a socket.
EOPNOTSUPP	The socket argument is associated with a socket that does not support one or more of the values specified in the flags argument.
EWOULDBLOCK (or the equivalent value EAGAIN)	The requested operation would block the process.

Related Functions

See getsockopt, recvfrom, select, send, sendto, and socket.

sendto

Purpose

The sendto function sends a message through a connection-mode or connectionless-mode socket. If the socket is connectionless-mode, sendto sends the message to the address specified by the to argument. If the socket is connection-mode, sendto ignores the to argument.

Syntax

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

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

Arguments

- s (input)
  A descriptor for a socket that the calling process created using the accept, receive_socket, or socket function. The descriptor is returned by the function when the process creates the socket.

- msg (input)
  The buffer that contains the data to be copied.

- len (input)
  The length, in bytes, of the buffer specified in msg.

- flags (input)
  A flag name that specifies handling for the message being sent. Its values, which are defined in the header file sys/socket.h, are as follows:

  - MSG_OOB or 0—Specify to mark as urgent the last byte of the data being copied. It is defined as a hexadecimal value associated with this flag.
  - _VOS_MSG_PARTIAL—Specify to inform the driver that the application will be sending more data. How the driver responds to this flag is driver-specific. The driver is not required to transport the data until the application uses send

without the _VOS_MSG_PARTIAL flag. The stcp driver uses this flag to optimize segmentation and to compute the TCP protocol push bit more efficiently. The stcp driver will not time-out and will not transmit partial segments, as it does with Nagling, which is a means of improving the efficiency of TCP/IP networks. The application must send data without the partial flag in order to signal STCP that data is to be transmitted.

- MSG_BLOCKING—Specify to override any nonblocking state. This flag does not exist if the application is coded for POSIX compliance; instead, specify _VOS_MSG_BLOCKING.
- MSG_NONBLOCKING—Specify to override any blocking state. This flag does not exist if the application is coded for POSIX compliance; instead, specify _VOS_MSG_NONBLOCKING.

► to (input)
A pointer to a sockaddr structure that contains the IP address and port number of the socket that is to receive the data.

► tolen (input)
The length, in bytes, of the IP address and port number of the receiving socket. The value of tolen must be sizeof(struct sockaddr_in).

Explanation

The to argument specifies the address of the target. The len argument specifies the length of the message.

Successful completion of a call to sendto does not guarantee delivery of the message. A return value of -1 indicates only locally-detected errors.

If the sending socket does not have sufficient space to hold the message to be transmitted and the socket file descriptor does not have O_NONBLOCK set, sendto blocks until space is available. If the sending socket does not have sufficient space to hold the message to be transmitted and the socket file descriptor has O_NONBLOCK set, sendto fails.

Two related functions also send data. The sendmsg function copies data to either connected or unconnected sockets. The send function copies data to connected sockets only.

POSIX.1 requires that the sendto function raise the SIGPIPE signal in the sending thread when the function attempts to write to a SOCK_STREAM socket that has been disconnected. In releases prior to OpenVOS Release 19.1.0, the POSIX.1 runtime failed to do this and instead returned the error code ENXIO. Beginning in OpenVOS Release 19.1.0, the sendto function raises SIGPIPE correctly in this situation.

If you have a program that previously failed because of this situation:

- Dynamically linked programs: You do not need to do anything. The program works automatically in OpenVOS Release 19.1.0 and later.
- Statically linked programs: Rebind the program.

If this change to OpenVOS causes problems with your program, perform one of the following actions:

- In the program's initialization, call the signal function:

  ```
  signal(SIGPIPE, SIG_IGN);
  ```

  In this case, the `send` function returns the `EPIPE` error in `errno`. This is the recommended action.

- Set the `s$bug_fix_stcp_3549` variable to zero. You declare this variable as:

  ```
  extern int s$bug_fix_stcp_3549
  ```

  You can set this variable either in the program's initialization or with the `set_external_variable` command. See the *OpenVOS Commands Reference Manual* (R098) for information about `set_external_variable`.

Return Values

If successful, `sendto` returns the number of bytes actually copied. If unsuccessful, it returns the value `-1`.

Error Codes

If sendto is unsuccessful, it sets the global variable errno (an int value) to the error code for one of the following error messages to indicate the specific error.

Error Message	Description
EBADF	The specified socket descriptor is invalid.
ECONNRESET	A connection was forcibly closed by a peer.
EDESTADDRREQ	The socket is not connection-mode and no peer address is set.
EFAULT	An invalid user-space address was specified for a parameter.
EINTR	A signal was caught during the sendto call.
EMSGSIZE	The socket requires that messages be copied atomically, and the size of the message to be copied made this impossible.
ENETDOWN	The local network interface used to reach the destination is down.
ENETUNREACH	No route to the network is present.
ENOBUFS	Insufficient system resources are available to complete the request.
ENOTCONN	The socket is not connected.
ENOTSOCK	The descriptor does not refer to a socket.
EOPNOTSUPP	The socket argument is associated with a socket that does not support one or more of the values specified in the flags argument.
EWOULDBLOCK (or the equivalent value EAGAIN)	The requested operation would block the process.

Related Functions

See getsockopt, recvfrom, select, send, sendmsg, and socket.

sethostent

Purpose

When querying a name server about a host, the `sethostent` function specifies that communications with a name server take place using virtual circuits rather than datagrams.

When searching for host information in the local `hosts` database file (located in the directory `(master_disk)>system>stcp`), the `sethostent` function opens the file (if it is not already open) and resets the file marker to the beginning of the file. The function can also specify that the file remain open after calls to `gethostbyaddr` and `gethostbyname` have finished executing.

Syntax

```
#include <netdb.h>

void sethostent(int stayopen);
```

Arguments

- `stayopen` (input)
  A flag. When the application specifies virtual-circuit communications with a name server, or when it specifies that the local `hosts` database file remain open after calls to `gethostbyaddr` and `gethostbyname` have finished executing, the value must be a nonzero integer.

  When the application specifies datagram communications with a name server, or when it specifies that the local `hosts` database file close after calls to `gethostbyaddr` and `gethostbyname` have finished executing (the default), the value must be `0`.

Explanation

The action that the `sethostent` function performs depends on whether the application is querying the name server or searching a database for host information.

When communicating with a name server, the gethostbyaddr and gethostbyname functions use datagrams (sockets of type SOCK_DGRAM) by default. The sethostent function allows you to specify that virtual circuits (sockets of type SOCK_STREAM) are used instead. Specifying virtual-circuit communications keeps the connection to the name server open between requests.

When the application is searching the local hosts database file, the sethostent function performs two steps. First, it opens the file (if it is not already open) and resets the file marker to the beginning of the file. This is useful if the file has already been opened and the location of the file marker is unknown. Second, if the value of the stayopen argument is a nonzero integer, sethostent specifies that the local hosts database file remain open after calls to gethostbyaddr and gethostbyname have finished executing. (By default, the file is closed.)

It is redundant to call sethostent and specify a nonzero integer value for the stayopen argument before calling gethostent because gethostent does not close the file.

Return Values

The sethostent function returns no values.

Error Codes

None.

Related Functions

See endhostent, gethostbyaddr, gethostbyname, and gethostent.

sethostname

Purpose

The sethostname function sets the name of the host module.

Syntax

```
#include <sys/socket.h>
int sethostname(char *name, int namelen);
```

Arguments

- name (output)
  The name of the host module.

- namelen (input)
  The length, in bytes, of the name specified in the name argument.

Explanation

The sethostname function sets the name of the host module to the name specified for the name argument. The length of the name is specified in the namelen argument; the maximum length of a hostname is defined as 255 by the Internet standards. The macro MAXHOSTNAMELEN, which has a value of 256 (to include a trailing null byte), may be used when manipulating host names. You must be logged in as a privileged user to use the sethostname function.

Stratus recommends that the host name for a module be set with the OpenVOS command sethost. (You can issue this command from command level, but it is usually issued from the module_start_up.cm file. See your system administrator for more information.) If necessary, however, an application can call the sethostname function to set the host name for the module.

Return Values

If successful, sethostname returns the value 0. If unsuccessful, it returns the value -1.

Error Codes

If `sethostname` is unsuccessful, it sets the global variable `errno` (an `int` value) to the error code for one of the following error messages to indicate the specific error.

Error Message	Description
`EFAULT`	The `name` or `namelen` parameter supplied an invalid address.
`EPERM`	The caller lacked the proper access.

Related Functions

See `gethostname`.

setnetent

Purpose

The setnetent function opens the networks database file (if it is not already open) and resets the file marker to the beginning of the file. This function can also specify that the file remain open after calls to getnetbyaddr and getnetbyname have finished executing.

Syntax

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

void setnetent(int stayopen);
```

Arguments

- stayopen (input)
  A flag. When the application specifies that the networks database file remain open after calls to getnetbyaddr and getnetbyname have finished executing, the value of stayopen should be a nonzero integer. When the application specifies that the file close after calls to getnetbyaddr and getnetbyname have finished executing (the default), the value of stayopen should be 0.

Explanation

The setnetent function performs two steps. First, it opens the networks database file in the directory (master_disk)>system>stcp (if it is not already open) and resets the file marker to the beginning of the file. This is useful if the file has already been opened and the location of the file marker is unknown. Second, if the value of the stayopen argument is a nonzero integer, setnetent specifies that the networks database file remain open after calls to getnetbyaddr and getnetbyname have finished executing. (By default, the file is closed.)

It is redundant to call setnetent and specify a nonzero integer value for the stayopen argument before calling getnetent because getnetent does not close the file.

Return Values

The `setnetent` function returns no values.

Error Codes

None.

Related Functions

See `endnetent`, `getnetbyaddr`, `getnetbyname`, and `getnetent`.

setprotoent

Purpose

The setprotoent function opens the protocols database file (if it is not already open) and resets the file marker to the beginning of the file. This function can also specify that the file remain open after calls to getprotobyname and getprotobynumber have finished executing.

Syntax

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

void setprotoent(int stayopen);
```

Arguments

- stayopen (input)
  A flag. When the application specifies that the protocols database file remain open after calls to getprotobyname and getprotobynumber have finished executing, the value of stayopen should be a nonzero integer. When the application specifies that the file close after calls to getprotobyname and getprotobynumber have finished executing (the default), the value of stayopen should be 0.

Explanation

The setprotoent function performs two steps. First, it opens the protocols database file in the directory (master_disk)>system>stcp (if it is not already open) and resets the file marker to the beginning of the file. This is useful if the file has already been opened and the location of the file marker is unknown. Second, if the value of the stayopen argument is a nonzero integer, setprotoent specifies that the protocols database file remain open after calls to getprotobyname and getprotobynumber have finished executing. (By default, the file is closed.)

It is redundant to call setprotoent and specify a nonzero integer value for the stayopen argument before calling getprotoent because getprotoent does not close the file.

Return Values

The `setprotoent` function returns no values.

Error Codes

None.

Related Functions

See `endprotoent`, `getprotobyname`, `getprotobynumber`, and `getprotoent`.

setservent

Purpose

The setservent function opens the services database file (if it is not already open) and resets the file marker to the beginning of the file. This function can also specify that the file remain open after calls to getservbyname and getservbyport have finished executing.

Syntax

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

void setservent(int stayopen);
```

Arguments

- stayopen (input)
  A flag. When the application specifies that the services database file remain open after calls to getservbyname and getservbyport have finished executing, the value of stayopen should be a nonzero integer. When the application specifies that the file close after calls to getservbyname and getservbyport have finished executing (the default), the value of stayopen should be 0.

Explanation

The setservent function performs two steps. First, it opens the services database file in the directory (master_disk)>system>stcp (if it is not already open) and resets the file marker to the beginning of the file. This is useful if the file has already been opened and the location of the file marker is unknown. Second, if the value of the stayopen argument is a nonzero integer, setservent specifies that the services database file remain open after calls to getservbyname and getservbyport have finished executing. (By default, the file is closed.)

It is redundant to call setservent and specify a nonzero integer value for the stayopen argument before calling getservent because getservent does not close the file.

Return Values

The `setservent` function returns no values.

Error Codes

None.

Related Functions

See `endservent`, `getservbyname`, `getservbyport`, and `getservent`.

setsockopt

Purpose

The setsockopt function sets an option for a specified socket.

Syntax

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

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

Arguments

- s (input)
  A descriptor for a socket that the calling process created using the accept, receive_socket, or socket function. The descriptor is returned by the function when the process creates the socket.

- level (input)
  The protocol level at which the option specified by optname exists. The following values are valid.

  - The value SOL_SOCKET indicates a socket-level (application-level) option. This value is defined in the header file sys/socket.h as the integer value associated with the socket level.
  - The value IPPROTO_IP indicates an IP-level option. This value is defined in the header file netinet/in.h.
  - The value IPPROTO_TCP indicates a TCP-level option that is used only with TCP_NODELAY.

- optname (input)
  The name of the option whose option value you want to obtain. Options are available at both the socket level and the IP level. The Explanation describes these options.

- `optval` (input)
  A pointer to the option value for the specified option. The Explanation describes the option values.

- `optlen` (input)
  The length, in bytes, of `optval`.

Explanation

The `setsockopt` function sets an option for a specified socket. Options are available at both the socket level, the IP level, and the TCP level (for the one TCP-level option).

TCP typically sends data in amounts up to the number of bytes that the peer advertises it is willing to accept before waiting for an acknowledgement that some of the data was received. A system administrator can use the `set_stcp_param` request of the `analyze_system` subsystem to set an upper limit on the number of bytes that TCP sends. Using the `set_stcp_param` request, a system administrator can set a new value for the `max_send_ws` parameter, which overrides the existing default value and allows TCP to send the maximum number of bytes that the peer is willing to receive.

For more information about using the `set_stcp_param` request of the `analyze_system` command to set the number and sizes of windows, see ‘‘Window Size’’ on page 4-15.

The Socket-Level Options

To set most socket-level options, the calling process specifies an option value of `0` or a nonzero integer. The `optval` argument is a pointer to this value.

You can reset these options. A `0` value clears or resets most options of data type `int`, while a nonzero integer sets most of these options. For more information, see Table 5-9, which describes the socket-level options.

Table 5-9. The `setsockopt` Socket-Level Options *(Page 1 of 3)*

Option	Data Type	Description
`SO_BROADCAST`	`int`	A nonzero value enables the calling process to send broadcast datagrams on the socket. The value `0` indicates the option is off.
`SO_DEBUG`	`int`	A nonzero value enables debugging in the underlying protocol modules. Debugging information is displayed on the system console. The value `0` indicates the option is off.
`SO_KEEPALIVE`	`struct linger`	Enables keepalive functionality and sets a keepalive time interval. For information, see “The SO_KEEPALIVE Option” on page 3-4.

Table 5-9. The `setsockopt` Socket-Level Options *(Page 2 of 3)*

Option	Data Type	Description
`SO_LINGER`	`struct linger`	Enables lingering and sets a time-out period called a linger interval. For information, see "The SO_LINGER Option" on page 3-8.
`SO_NODELAY`	`int`	A nonzero value enables the socket to forward data expediently. The value `0` indicates the option is off.
`TCP_NODELAY`	`int`	Identical to `SO_NODELAY` except that `TCP_NODELAY` uses the `IPPROTO_TCP` level.
`SO_RCVBUF`	`int`	For UDP, sets the maximum amount of data that the UDP driver can buffer on input. By default, this amount is 50 datagrams of size 2K. If datagrams are received faster than the application can process them, STCP discards them after the limit set by `SO_RCVBUF` is exceeded. For TCP, sets the maximum amount of data that TCP can buffer. This value is also used to set the advertised receive-window size. If the user has write access to the STCP device, the receive window is set to the smallest preconfigured size (256K, 64K, 32K, 16K, or 8K) that is greater than the size of the specified `SO_RCVBUF`. If the user has only read access to the STCP device, the receive window is set to the smallest preconfigured size that is greater than the size of the specified `SO_RCVBUF` and whose current count of users (for that preconfigured window size) is smaller than the maximum number of users allowed. For information on setting the maximum number of users for each of several preconfigured sizes, see "Window Size" on page 4-15.
`SO_REUSEADDR`	`int`	A nonzero value specifies that the rules used to validate addresses supplied in a `bind` call permit reuse of local addresses. For information, see "The SO_REUSEADDR Option" on page 3-8.

Table 5-9. The `setsockopt` Socket-Level Options *(Page 3 of 3)*

Option	Data Type	Description
SO_SNDBUF	int	For UDP, sets the maximum amount of data that can be buffered in the UDP driver on output. By default, the size is 98,304 bytes, allowing for about 51 maximum-size packets. If datagrams are being sent at a faster rate than the application can process them, one of the following situations occurs when the limit set by SO_SNDBUF is exceeded: – If the socket is set for blocking mode, the application blocks datagrams. – If the socket is set for nonblocking mode, STCP sets errno to EWOULDBLOCK (or the equivalent value EAGAIN). For TCP, sets the maximum amount of outgoing data that TCP buffers. TCP does not drop packets when the send buffer is full; instead, TCP blocks output on the sending socket. The default buffer size is 16K bytes. The value 0 indicates the option is off.
SO_URGENT	int	Allows the socket to receive urgent data. This option is on by default and remains on, regardless of its setting.
_TCP_STDURG	int	A nonzero value indicates that the urgent pointer sent to and received from the client points at the last octet of urgent data, as required by RFC 1122. The value 0 indicates that the urgent pointer points at the first octet after the urgent data, which is the historical implementation.
_TCP_SHARED_ DYNAMIC_PORT		A nonzero value specifies that, when binding, STCP does not assign a local port to the socket, which typically occurs automatically when binding. Instead, during the connect operation, STCP assigns to the socket a local port from a range of numbers for shared dynamic ports. STCP assigns the local port number, taking into account the local address and local port as well as the remote address and remote port. Thus, STCP can reuse the same local port number with a different remote address; in fact, STCP can use the same local port number with hundreds of thousands of different peers. This option should be used by applications that need to establish more than 16,000 outgoing TCP connections.

The IP-Level Options

Table 5-10 summarizes the options available at the IP level.

Table 5-10. The `setsockopt` IP-Level Options

Option	Setting	Description
`IP_ADD_MEMBERSHIP`	Settable (IP address)	Enables multicast group reception for the socket and interface at the specified IP address.
`IP_DROP_MEMBERSHIP`	Settable (IP address)	Disables multicast group reception for the socket and interface at the specified IP address.
`IP_MULTICAST_IF`	Settable (IP address)	Sets the default interface for multicast transmission to that of the specified source address, or clears the interface if `INADDR_ANY` is specified.
`IP_MULTICAST_LOOP`	Toggled (`int` or `unsigned char`)	Disables the loopback of transmitted multicast messages. That is, typically, sockets registered for a multicast on the same machine as the sender will receive the multicast messages; this option disables this feature.
`IP_MULTICAST_TTL`	Settable (`int` or `unsigned char`)	Sets the time-to-live (TTL) for outgoing multicast datagrams. See Table 5-11 for information about recommended TTL values.
`IP_TTL`	Settable (`unsigned char`)	Indicates that the socket allows TTL for outgoing unicast datagrams and broadcast datagrams. Specify a large value; the default is `240`.

An administrator assigns threshold values when configuring an IP multicast router. These values define the scope of IP multicast data packets. The significance of an initial TTL value for IP multicast data packets is defined by the administrator's threshold policy and by the distance between the source of the data packet and the IP multicast interfaces.

Table 5-11 lists, for IP multicast datagrams, the recommended TTL values for various types of applications as well as recommended threshold values. It is important to set the proper TTL value in order to avoid long delays before a listener program starts receiving packets.

Table 5-11. Recommended TTL Values and Threshold Values for IP Multicast Datagrams
(Page 1 of 2)

Initial TTL Value	Sample Application	Scope
0		Same interface

Table 5-11. Recommended TTL Values and Threshold Values for IP Multicast Datagrams
(Page 2 of 2)

Initial TTL Value	Sample Application	Scope
1		Same subnet
31	Local event video	
32		Same site
63	Local event audio	
64		Same region
95	IETF channel 2 video	
127	IETF channel 1 video	
128		Same continent
159	IETF channel 2 audio	
191	IETF channel 1 audio	
223	IETF channel 2 low-rate audio	
255	IETF channel 1 low-rate audio unrestricted in scope	

In Table 5-11, the first column lists the initial TTL value in the IP header. The second column illustrates an application-specific use of threshold values. The third column lists the recommended scopes to associate with the TTL values.

For example, you would configure an IP multicast interface that communicates with a network outside the local site with a multicast threshold of 32. The TTL field of any data packet that starts with a TTL of 32 (or less) is less than 32 when it reaches this interface (at least one hop occurs between the source and the router). Therefore, the packet is discarded before the router forwards it to the external network, even if the TTL is still greater than 0.

An IP multicast data packet that has an initial TTL value of 128 would pass through site interfaces with a threshold of 32 (as long as it reached the interface within 96 hops (128 - 32 = 96). However, intercontinental interfaces with a multicast threshold of 128 would discard it.

NOTE

UDP is not a reliable protocol. Therefore, you should be aware that a delay could occur with the IP multicast registration joins/drops (that is, with IGMP joining

multicast groups to the router and deleting them from the router).

Return Values

If successful, `setsockopt` returns the value `0`. If unsuccessful, it returns the value `-1`.

Error Codes

If `setsockopt` is unsuccessful, it sets the global variable `errno` (an `int` value) to the error code for one of the following error messages to indicate the specific error.

Error Message	Description
`EBADF`	The specified socket descriptor is invalid.
`EFAULT`	The address pointed to by `optval` or the value specified by `optlen` is not in a valid part of the process address space.
`EINTR`	A signal was caught during the `setsockopt` call.
`EINVAL`	The value specified by `optname` is invalid, or the address specified by `optval` is invalid.
`ENOBUFS`	The system cannot perform the operation due to insufficient resources.
`ENOPROTOOPT`	The option is not supported by the protocol.
`ENOTSOCK`	The socket descriptor does not refer to a socket.

Related Functions

See `getprotoent`, `getsockopt`, `setprotoent`, and `socket`.

shutdown

Purpose

The `shutdown` function shuts down a full-duplex (read/write) socket for read and/or write operations.

Syntax

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

int shutdown(int s, int rw);
```

Arguments

- `s` (input)
  A descriptor for a socket that the calling process created using the `accept`, `receive_socket`, or `socket` function. The descriptor is returned by the function when the process creates the socket.

- `rw` (input)
  Specifies the extent of the socket shutdown. You can specify one of the following values.

  - The value `SHUT_RD` prevents subsequent read operations. Any subsequent read attempt returns an error code.
  - The value `SHUT_WR` prevents subsequent write operations. Any subsequent write attempt returns an error code.
  - The value `SHUT_RDWR` prevents subsequent read and write operations. Any subsequent read or write attempt returns an error code.

Explanation

The `shutdown` function shuts down a socket for read and/or write operations. When a socket is created with the `socket` function, by default, it is opened for both reading and writing.

Using `shutdown` does not release a socket's resources. To close down a socket completely, you must call the `close` function. Note that you need not call `shutdown` before calling `close`.

Return Values

If successful, `shutdown` returns the value `0`. If unsuccessful, it returns the value `-1`.

Error Codes

If `shutdown` is unsuccessful, it sets the global variable `errno` (an `int` value) to the error code for one of the following error messages to indicate the specific error.

Error Message	Description
`EBADF`	The specified socket descriptor is invalid.
`EINTR`	A signal was caught during the `shutdown` call.
`EINVAL`	The `rw` argument is invalid.
`ENOBUFS`	Insufficient system resources are available to complete the request.
`ENOTCONN`	The socket is not connected.
`ENOTSOCK`	The descriptor does not refer to a socket.

Related Functions

See `socket`.

so_recv

Purpose

The so_recv function receives urgent data from a connected socket. (The received data is sometimes called a *message*.)

Syntax

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

int so_recv(int s, char *msg, int len, int *uflag);
```

Arguments

- s (input)
  A descriptor for a socket that the calling process created using the accept, receive_socket, or socket function. The descriptor is returned by the function when the process creates the socket.

- msg (output)
  The buffer that is to contain the received data.

- len (input)
  The length, in bytes, of the buffer specified as msg.

- uflag (output)
  Indicates the presence of urgent data. When urgent data is pending, uflag is set to the value OOB_PEND. After the first byte of urgent data is read, uflag is set to MSG_OOB. The uflag argument is set only on the first read of each urgent message. If the urgent data does not fit completely into the buffer, subsequent reads will not set the flag. Urgent data is never concatenated with nonurgent data, however. When no urgent data is outstanding, uflag is not modified.

Explanation

The so_recv function receives messages from the socket s and places them in the buffer msg. It is used only with a connected socket of the type SOCK_STREAM. To receive data on another type of socket, use the recvfrom function.

If no data is available at the socket, and the socket is in blocking mode, so_recv waits for a data packet to arrive. However, if the socket is in nonblocking mode, so_recv returns the value -1 and sets the errno global variable to the error code for EWOULDBLOCK (or the equivalent value EAGAIN). Use the select function to determine when more data is available.

Return Values

If successful, so_recv returns the number of bytes in the message. (A returned value of 0 indicates an end-of-file condition.) If unsuccessful, it returns the value -1.

Error Codes

If so_recv is unsuccessful, it sets the global variable errno (an int value) to the error code for one of the following error messages to indicate the specific error.

Error Message	Description
EBADF	The specified socket descriptor is invalid.
ECONNRESET	The connection associated with socket s was forcibly closed by a peer.
EFAULT	The data was specified to be received into a nonexistent or protected part of the process address space.
EINTR	The receive was interrupted by delivery of a signal before any data was available.
EINVAL	The len argument is less than zero.
EIO	An internal error has occurred.
ENOBUFS	Insufficient system resources are available to complete the request.
ENOTCONN	The requested operation was attempted on a connection-mode socket that is not connected.
ENOTSOCK	The file descriptor s is not associated with a socket.
EOPNOTSUPP	The specified flags are not supported for this socket type or protocol.
EWOULDBLOCK (or the equivalent value EAGAIN)	The file descriptor s is marked O_NONBLOCK, and no data is waiting to be received.

Related Functions

See setsockopt, read, recvfrom, select, and socket.

socket

Purpose

The socket function creates a socket (that is, a communications endpoint) for the calling process and returns a descriptor for the socket. The socket is created on the local host.

Syntax

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

int socket(int af, int type, int prot);
```

Arguments

- af (input)
  The address format to be used in later operations on the socket. The value of af must be one of the following:
  - AF_INET to indicate the AF_INET address family, which supports IPv4 Internet communications
  - AF_INET6 to indicate the AF_INET6 address family, which supports IPv6 Internet communications

NOTES

1. You cannot mix usage of AF_INET and AF_INET6 on the same socket.
2. You can operate an IPv4 connection on an AF_INET6 socket using IPv4-mapped IPv6 addresses. In this case, both the source and destination must be IPv4-mapped addresses.
3. The IPv6 protocol is supported only by POSIX. For information about IPv6 support, see the *OpenVOS POSIX.1 Reference Guide* (R502).

The address format of AF_INET (and AF_INET6) is defined as an integer value in the header file sys/socket.h.

- type (input)
  The protocol type used to communicate over the socket. (This is also called the socket type.) The following values are valid: SOCK_STREAM, SOCK_DGRAM, and SOCK_RAW.

  Each of the preceding protocol types is defined as an integer value in the header file sys/socket.h.

- prot (input)
  The specific protocol used with the socket. For sockets of the type SOCK_STREAM or SOCK_DGRAM, specify the value PF_UNSPEC or PF_INET. For sockets of the type SOCK_RAW, specify a value defined in the netinet/in.h header file.

Explanation

The socket function creates a socket for the calling process on a local host. It returns a socket descriptor for the new socket. The calling process can then use the socket descriptor s to refer to the socket in subsequent operations, such as send and recv.

A system has the following limits to active sockets:

- approximately 16,000 sockets of the type SOCK_STREAM, depending on system usage, although each process is limited to 4096 open sockets and files
- 5000 sockets of the type SOCK_DGRAM

If you exceed this limit by attempting to create a new socket using socket or accept for SOCK_STREAM sockets, or by using socket or receive_socket for SOCK_DGRAM sockets, the function returns the value -1 and sets errno to ENFILE.

A system administrator can configure the number of usable sockets to be lower than the limit using the clone_limit field in the devices.tin file entry for the STCP or UDP driver. For example, a clone_limit value of 4096 limits the maximum number of sockets for the corresponding type to 4096. For information on the devices.tin file entry for the STCP driver, see *OpenVOS STREAMS TCP/IP Administrator's Guide* (R419).

After creating a socket, a process can associate an IP address and port number with that socket by issuing a call to the bind function. This IP address and port number, unlike the socket descriptor, makes the socket available to other processes on the network.

The operation of sockets is controlled by socket-level options, which are defined in the header file sys/socket.h. An application uses setsockopt to toggle options on or off or to set their values, and uses getsockopt to retrieve current option settings and

values. See the setsockopt function description for information about the available options.

Use the close function to close a socket when a session is complete.

Return Values

If successful, socket returns the socket descriptor, which references the new socket. If unsuccessful, it returns the value -1.

Error Codes

If socket is unsuccessful, it sets the global variable errno (an int value) to the error code for one of the following error messages to indicate the specific error.

Error Message	Description
EAFNOSUPPORT	The specified address family is not AF_INET, and thus is not supported.
EMFILE	No more file descriptors are available for this process (a maximum of 4096 are available).
ENFILE	No more sockets of the type specified are available for this system.
ENOBUFS	Insufficient system resources are available to complete the request.
ENXIO	No such device or address exists.
EPROTONOSUPPORT	The protocol is not supported.
EPROTOTYPE	The socket type is not supported by the protocol. This also indicates inconsistent protocol drivers (that is, IP, UDP, or TCP) are installed. All drivers must be of the same revision.
ESOCKTNOSUPPORT	The specified socket type is not supported.

Related Functions

See accept, bind, connect, getsockname, getsockopt, listen, read, recv, select, send, shutdown, and write.

stcp_spawn_process

Purpose

Creates a new process that executes the application passed to the function.

NOTE

This function is provided for first generation STCP applications. If you are writing new code, use the `fork` function.

Syntax

```
#include <system_io_constants.h>
#include <cp_arg_info.h>
#include <pdr_constants.h>
#include <start_process_info.h>
#include <string.h>

void stcp_spawn_process (const char *p_command_line,
                         const char *p_process_name,
                         char_varying(256) *p_error_path,
                         char_varying(256) *p_module,
                         char_varying(256) *current_dir,
                         char_varying(256) *p_termination_path,
                         long *p_process_id,
                         short *p_code,
                         int *ctrl);
```

Arguments

- `p_command_line`
  A pointer to the name of the application being passed to the function.

- `p_process_name`
  A pointer to the name of the created process.

- `p_error_path`
  A pointer to the standard error environment variable passed to the process.
- `p_module`
  A pointer to the current module.
- `current_dir`
  A pointer to the current directory.
- `p_termination_path`
  A pointer to a file; when the process terminates, it notifies this file.
- `p_process_id`
  A pointer to the returned value after the process is created.
- `p_code`
  A pointer to the returned error code.
- `ctrl`
  A pointer to a socket descriptor.

Explanation

The `stcp_spawn_process` function creates a new process that executes the application passed in the `p_command_line` argument. You can use this function to pass sockets from one process to another.

Return Values

None.

Error Codes

If `stcp_spawn_process` is unsuccessful, it returns one of the following error messages.

Error Message	Description
`e$too_many_processes` (`1009`)	The maximum limit of all processes has been reached.
`e$too_many_user_procs` (`2452`)	You have attempted to exceed the maximum number of subprocesses specified by your system administrator.
`e$too_many_login_procs` (`2457`)	The maximum number of non-batch processes for the module has been reached.

Related Functions

None.

transfer_socket

Purpose

The `transfer_socket` function initiates the transfer of a socket from one process to another.

NOTE

This function is provided for OS TCP/IP compatibility. If you are writing new code, you should use the `fork` function.

Syntax

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

int transfer_socket(int sd, char *pathname);
```

Arguments

- `sd` (input)
  The socket descriptor for the socket to be transferred.

- `pathname` (output)
  A pointer to a buffer large enough to hold the path name of the socket device (67 bytes).

Explanation

The `transfer_socket` function initiates transferring a socket to another process. It accepts a file descriptor (`sd`) that specifies the socket to be transferred. It sets the `pathname` output argument to a value that specifies the socket.

The caller of `transfer_socket` is responsible for sending this path name to the receiving process. The caller of `transfer_socket` cannot close its file descriptor until it is notified that the receiving process has successfully received the socket. After it has been notified that the receiving process has received the socket, it must close the socket.

Return Values

If `transfer_socket` successfully retrieves the socket path name `pathname`, it returns the value `0`. If it cannot retrieve the socket path name, it returns value `-1`.

Error Codes

If `transfer_socket` cannot retrieve the path name, it sets the global variable `errno` (an `int` value) to the error code that indicates the specific error. The most common error messages follow.

Error Message	Description
`EBADF`	The specified socket descriptor is invalid.

Access Requirements

None.

Related Functions

See `receive_socket`.

write

This function is provided by the OpenVOS Standard C library and POSIX library. For detailed information, see the *OpenVOS Standard C Reference Manual* (R363). The *OpenVOS POSIX.1 Reference Guide* (R502) also provides some information.

writev

This function is provided by the OpenVOS Standard C library and POSIX library. For detailed information, see the *OpenVOS Standard C Reference Manual* (R363). The *OpenVOS POSIX.1 Reference Guide* (R502) also provides some information.

Appendix A
STCP Sample Programs

This appendix contains sample programs that use the STCP application interface. You must bind these programs with the STCP run-time library, as described in Chapter 4.

This appendix contains the following sections.

Examples of the sample programs in this appendix converted to POSIX with IPv6 support are located in the `(master_disk)>system>stcp>sample_programs` directory.

Sample Program Using the `sockaddr` Structure

Figure A-1 illustrates the program `stcp_address_sample.c`. This program shows the use of address manipulation routines and socket address structures.

```
/*
 * This sample program shows the use of address manipulation routines
 * and socket address structures.
 *
 * The sample Internet (AF_INET) address is: 134.111.100.1, 1500  */

#include <arpa/inet.h>
#include <stdio.h>
#include <sys/socket.h>
#include <unistd.h>
```

(Continued on next page)

```
char caddr_str[20]= "134.111.75.31"; /* %sw#m1 */
char *raddr_str;                     /* address of sender of received msg */

#define BFRSIZE 4096
char buf[BFRSIZE];
#define FLAG_BITS 0

int main()
{
     struct sockaddr     saddr;
     struct sockaddr_in  *sin;
     int                 port_no = 49876;
     int                 sd;
     int                 rval;
     int                 frmlen;

     sin = (struct sockaddr_in *)&saddr;

     /* set up a socket address structure containing the address with
      * which we want to connect(). Socket addresses for use in other
      * calls are set up in similar fashion.
      */

     sin->sin_family = AF_INET;                    /* set address family */
     sin->sin_port = port_no;                      /* set port number */
     sin->sin_addr.s_addr = inet_addr(caddr_str);  /* convert address string
*/
     if (sin->sin_addr.s_addr == -1)
     {
          printf("Improper address");
          return(-1);
     }

     sd = socket(AF_INET, SOCK_STREAM, 0);
     if (sd < 0)
     {
          perror("Error on socket()");
          return(-1);
     }

     rval = connect(sd, &saddr, sizeof(struct sockaddr));
     if (rval < 0)
     {
          perror("stcp_address_sample: Error on connect()");
          return(-1);
     }
     printf ("Connected to %s, port %d\n", caddr_str, port_no);
     close(sd);
     printf ("Now closed\n");
```

(Continued on next page)

```
/* receive a (datagram) message from an unconnected peer and get the
      * address string from the returned socket address structure.
      * Internet addresses can be similarly obtained from other calls
      * which return data in a socket address structure.
      */

     sd = socket(AF_INET, SOCK_DGRAM, 0);
     if (sd < 0)
     {
          perror("stcp_address_sample: Error on socket()");
          return(-1);
     }

     rval = bind(sd, &saddr, sizeof(struct sockaddr));
     if (rval < 0)
     {
          perror("stcp_address_sample: Error on bind()");
          return(-1);
     }

     frmlen = sizeof(struct sockaddr);
     printf ("Waiting for UDP message on port %d\n", port_no);
     rval = recvfrom(sd, buf, sizeof(buf),FLAG_BITS,
                     &saddr, &frmlen);
     if (rval < 0)
     {
          perror("stcp_address_sample: Error on recvfrom()");
          return(-1);
     }

     raddr_str = inet_ntoa(sin->sin_addr);
     printf ("Received UDP packet from %s\n", raddr_str);

     close(sd);
     return(0);
}
```

Figure A-1. Sample Program Using the `sockaddr` Structure

Sample Client Program

Figure A-2 illustrates the program `simple_stcp_accept.c`.

```
/*
 * This sample program shows the use of the accept() call to establish
 * a connection and to receive and send data on it.
*/

#include <sys/socket.h>
#include <c_utilities.h>
#include <stdlib.h>
#include <stdio.h>
#include <arpa/inet.h>
#include <string.h>

/*
 * Protocol (0) is the only protocol allowed. This field in
 * the socket() call is only there for future expansion.
 */

#define DEF_PROTOCOL    0
#define FLAG_BITS       0
#define BUFSIZE 4096

void example_exit(char * exit_str);
char recv_buf[BUFSIZE];
void s$parse_command();

int main()
{
    short                error_code;
    char_varying(256)    v_iaddress;
    char                 iaddress[256];
    int                  port_no = 49876;
    int                  sd, listen_sd;
    struct sockaddr      bind_saddr;
    struct sockaddr_in   *sin;
    int                  rval;
    int                  msg_size;

    int                  listen_backlog;
    int                  msg_recv = 0;
    int                  msg_bytes_recv = 0;
    int                  addrlen ;
```

(Continued on next page)

```
/* Define the default address of this program's accepting socket */
/*
 * Even though the bind() call expects a fully qualified address,
 * the only part of the address that is of interest is the port
 * address. The convention for specifying the current network/host
 * address, whatever it happens to be, is 0.0.0.0.
 */
strcpy_vstr_nstr(&v_iaddress, "0.0.0.0");
s$parse_command(&(char_varying(32))"simple_stcp_accept",
      &error_code,
      &(char_varying)"option(-address), string, value", &v_iaddress,
      &(char_varying)"option(-port), number, longword, min(1),
           max(65535), value, =0",
      &port_no, &(char_varying(32))"end");

if (error_code != 0)
    return(error_code);

/* Convert a char_varying string into a null-terminated one */
strcpy_nstr_vstr(iaddress, &v_iaddress);

/*
 * Create a socket that can be used to "listen" for an
 * incoming TCP connection.
 */
listen_sd = socket(AF_INET, SOCK_STREAM, DEF_PROTOCOL);
if (listen_sd < 0)
    example_exit("Error on socket()");

/*
 * Convert the bind address that is supplied from s$parse_command
 * into the "sockaddr_in" structure, which can then be used for
 * the accept() call.
 */
sin = (struct sockaddr_in *)&bind_saddr;
sin->sin_family = AF_INET;
sin->sin_port = port_no;
sin->sin_addr.s_addr = inet_addr(iaddress);
if (sin->sin_addr.s_addr == -1)
    example_exit("Improper bind address");

rval = bind(listen_sd, &bind_saddr, sizeof( bind_saddr));
if (rval < 0)
    example_exit("Error on bind()");
listen_backlog = 1; /* Only need to queue up a maximum of 1 */
rval = listen(listen_sd, listen_backlog);
if (rval < 0)
    example_exit("Error on listen()");
addrlen = sizeof(bind_saddr) ;
sd = accept(listen_sd, &bind_saddr, &addrlen);
```

(Continued on next page)

```
    if (sd <= 0)
        example_exit("Error on accept()");

    printf("Established connection with %s\n", inet_ntoa(sin->sin_addr));
    for(;;)
    {
        rval = recv(sd, recv_buf, BUFSIZE, FLAG_BITS);
        if (rval < 0)
            example_exit("Error on recv()");

        if (rval == 0)
            break;     /* Normal TCP connections shutdown */

        msg_size = rval;
        msg_recv++;
        msg_bytes_recv += msg_size;

        rval = send(sd, recv_buf, msg_size, FLAG_BITS);
        if (rval != msg_size)
            example_exit("Error on send()");
    }
    close(sd);


printf("Number of recvs/sends: %d. Number of bytes: %d.\n",
        msg_recv, msg_bytes_recv);
return(0);
}

void example_exit(char * exit_str)
{

    /*
     * Use perror() to print a message for the current value in errno.
     */
    perror(exit_str);
    exit(1);
}
```

Figure A-2. Sample Client Program

Sample Program Accepting a Single Connection Request

Figure A-3 illustrates the program `simple_stcp_connect.c`.

```
/*
 * This sample program shows the use of the connect() call to establish
 * a connection and send and receive data on it. */

#include <arpa/inet.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>

/*
 * Protocol (0) is the only protocol allowed. This field in
 * the socket() call is only there for future expansion.
 */

#define DEF_PROTOCOL    0
#define FLAG_BITS       0
#define BUFSIZE 4096
char recv_buf[BUFSIZE], send_buf[BUFSIZE];

void example_exit(char * exit_str);
int buffer_compare (char * buf1, char * buf2, int size);

void s$parse_command();

int main()
{
    short                error_code;
    char_varying(256)    v_iaddress;
    char                 iaddress[256];
    int                  port_no = 49876;
    int                  sd;
    struct sockaddr      bind_saddr;
    struct sockaddr_in   *sin;
    int                  rval;
    int                  r_cnt;

    int                  msg_recv = 0;
    int                  msg_sent = 0;
    short                msg_cnt = 100;
    short                msg_size = 256;
```

(Continued on next page)

```
/* Define the default address of the destination socket */
strcpy_vstr_nstr(&v_iaddress, "134.111.75.31"); /* m1 */
s$parse_command(&(char_varying(32))"simple_stcp_connect", &error_code,
    &(char_varying)"option(-address), string, value", &v_iaddress,
    &(char_varying)"option(-port), number,longword, min(1), max (65535),
         value, =0", &port_no,
    &(char_varying)"option(-msg_cnt), number, value", &msg_cnt,
    &(char_varying)"option(-msg_size),number,max(4096),value", &msg_size,
    &(char_varying(32))"end");

if (error_code != 0)
    return(error_code);

/* Convert a char_varying string into a null-terminated one */
strcpy_nstr_vstr(iaddress, &v_iaddress);

/*
 * Create a socket that can be used to establish a TCP connection
 * to the destination socket.
 */

sd = socket(AF_INET, SOCK_STREAM, DEF_PROTOCOL);
if (sd < 0)
    example_exit("Error on socket()");

/*
 * Convert the bind address that is supplied from s$parse_command
 * into the 'sockaddr_in' structure which can then be used for
 * the connect() call.
 */

sin = (struct sockaddr_in *)&bind_saddr;
sin->sin_family = AF_INET;
sin->sin_port = port_no;
sin->sin_addr.s_addr = inet_addr(iaddress);
if (sin->sin_addr.s_addr == -1)
    example_exit("Improper connect address");

/* Post a connection request to the destination address. */

rval = connect(sd, &bind_saddr, sizeof(bind_saddr));
if (rval < 0)
 {
 printf("rval: %d\n", rval);
 printf("sd: %d\n", sd);
 example_exit("Error on connect()");
 }
```

(Continued on next page)

```
    while (msg_cnt > 0)
    {
        memset(send_buf, (msg_cnt % 256), msg_size);
        rval = send(sd, send_buf, msg_size, FLAG_BITS);
        if (rval != msg_size)
            example_exit("Error on send()");
        msg_sent++;

    /* Now receive the data back */
        r_cnt = 0;
        while (r_cnt < msg_size)
        {
            rval = recv(sd, &recv_buf[r_cnt],
                    msg_size - r_cnt, FLAG_BITS);
            if (rval <= 0)
                example_exit("Error on recv()");
            r_cnt += rval;
            msg_recv++;
        }
        buffer_compare(send_buf, recv_buf, msg_size);
        msg_cnt--;
    }

    shutdown(sd, READ_WRITE_SHUTDOWN);
    close(sd);

    printf("Number of sends: %d. Number of recvs: %d. Msg size: %d\n",
        msg_sent, msg_recv, msg_size);
}

int buffer_compare(char * buf1, char * buf2, int size)
{
    void exit();

    while (size--)
    {
        if (*buf1++ != *buf2++)
        {
            printf("Data comparison failure\\n");
            exit(0);
        }
    }
    return(0);
}
```

(Continued on next page)

```
void example_exit(char * exit_str)
{
    void  exit();
    /*
     * use perror() to print a message for the current value in errno.
     */

    perror(exit_str);
    exit(1);
}
```

Figure A-3. Sample Program Accepting a Single Connection Request

Sample Program Accepting Multiple Connection Requests

Figure A-4 illustrates the program `multiple_stcp_accepts.c`. This program shows a typical server program.

```
/*
 * This sample program shows how the functionality of the simple_tcp_accept.c
 * program can be extended to accept multiple sockets.
 *
 * This program is an example of a server program which could be started up
 * as a background process. Multiple simple_tcp_connect.pm programs could
 * run and establish connections with this program and exchange data.  */


#include <arpa/inet.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/select.h>
#include <fcntl.h>
#include <unistd.h>
/*
 * Protocol (0) is the only protocol allowed. This field in the socket()
 * call is only there for future expansion.
 */
#define DEF_PROTOCOL    0
#define FLAG_BITS       0
#define MAX_SDS 16
#define BUFSIZE 4096

int max_socket = 0;    /* The highest socket number in use */
int num_sockets = 0;   /* The number of sockets in use */
int sd[MAX_SDS];
```

(Continued on next page)

```
/*
 * The sd[] is used to store active socket numbers.  The number of sd[]
 * elements that are in use is equal to the number of active sockets.
 * Element 0 of sd[] is used for listen socket */

#define LISTEN   0

fd_set   active_sds;
fd_set   read_sds;

/* The mechanism for returning error status to the application for any
 * TCP runtime interface call, is to return -1 and set the global variable
 * 'errno' to an appropriate error code.
 */

char error_text[80];
int close_socket(int si);
int find_unused_socket(void);
void example_exit(char * exit_str);

void s$parse_command();

int main()
{
    short                error_code;
    char_varying(256)    v_iaddress;
    char                 iaddress[256+1];
    int                  port_no = 49876;
    struct sockaddr_in   bind_saddr, nsaddr;
    int                  nsaddr_sz;
    int                  rcvd_bytes;
    int                  rval, sval, nfound, i;
    int                  listen_backlog;
    char                 buffer [MAX_SDS] [BUFSIZE];

    /* Define the default address of this program's accepting socket.
     * The convention for specifying the current network/host address
     * is 0.0.0.0.
     */
    strcpy_vstr_nstr(&v_iaddress, "0.0.0.0");

    s$parse_command(&(char_varying(32))"multiple_stcp_accepts", &error_code,
        &(char_varying)"option(-address), string, value", &v_iaddress,
        &(char_varying)"option(-port), number,longword, min(0), max(65535),
        value, =0", &port_no, &(char_varying(32))"end");

    if (error_code != 0)
        return(error_code);
```

(Continued on next page)

```
/* Convert address char_varying string into a null-terminated one */
strcpy_nstr_vstr(iaddress, &v_iaddress);

/* initialize socket array */
for (i = 0; i < MAX_SDS; i++)
    sd[i] = 0;

/*
 * Create a socket that can be used to 'listen' for an
 * incoming TCP connection.
 */
sd[LISTEN] = socket(AF_INET, SOCK_STREAM, DEF_PROTOCOL);
if (sd[LISTEN] < 0)
    example_exit("Error on socket()");

/*
 * Convert the bind address that is supplied from s$parse_command
 * into the 'sockaddr_in' structure which can then be used for
 * the accept() call.
 */
bind_saddr.sin_family = AF_INET;
bind_saddr.sin_port = port_no;
bind_saddr.sin_addr.s_addr = inet_addr(iaddress);
if (bind_saddr.sin_addr.s_addr == -1)
    example_exit("Improper bind address");

rval = bind(sd[LISTEN],(struct sockaddr *)&bind_saddr, sizeof(bind_saddr));
if (rval < 0)
    example_exit("Error on bind()");

/* set incoming connection queue length */
listen_backlog = 5;
rval = listen(sd[LISTEN], listen_backlog);
if (rval < 0)
    example_exit("Error on listen()");

/* Set socket non-blocking */

if(fcntl(sd[LISTEN], F_SETFL, O_NDELAY) < 0)
      example_exit("Error on setting O_NDELAY");

 /*
 * Set read mask for our listening socket, so we will know when a
 * connection request is available.
 */

FD_SET(sd[LISTEN], &active_sds);
num_sockets = 0;

for (;;)
```

(Continued on next page)

```
{
    /* find highest socket number */
    for (i = 0; i < MAX_SDS; i++)
        if (sd[i] != 0 && sd[i] > max_socket)
            max_socket = sd[i];

    read_sds = active_sds;

    nfound = select(max_socket + 1, (fd_set *)&read_sds, (fd_set *)0,
                                (fd_set *)0, (struct timeval *)0);
    if (nfound < 0)
        example_exit("Error on select()");

    /*
     * Check which sockets were notified of something.
     */
    for (i = 0; nfound && i < MAX_SDS; i++)
    {
        if (FD_ISSET(sd[i], &read_sds))
        {
            if (i == LISTEN)   /* on the listening socket */
            {
                int nsd, nsi;

                /*
                 * An incoming connection request is pending; the accept()
                 * call must be issued to establish the connection.
                 */
                nsaddr_sz = sizeof(struct sockaddr);
                nsd = accept(sd[LISTEN], (struct sockaddr *)&nsaddr,
                      (int *) &nsaddr_sz);

                if (nsd < 0)
                {
                    perror("Error on accept()");
                    continue;           /* process next */
                }

                nsi = find_unused_socket();

                if (nsi < 0)
                {
                    printf("Connection refused, too many sockets\n");
                    close(nsd);
                    continue;           /* process next */
                }

                sd[nsi] = nsd;

                FD_SET(sd[nsi], &active_sds);
                num_sockets++;
```

(Continued on next page)

```
                printf("\nNew socket descriptor is %d.\n", sd[nsi]);
                printf("Source address of new socket is %s\n",
                                inet_ntoa(nsaddr.sin_addr));
                printf("%d connected sockets.\n", num_sockets);
                continue;
            }

            else          /* receiving socket */
            {
                rcvd_bytes = recv(sd[i], buffer [i], BUFSIZE, FLAG_BITS);
                if (rcvd_bytes < 0)
                {
                    sprintf(error_text,
                          "Error on recv() - socket %d", sd[i]);
                    perror(error_text);
                    close_socket(i);
                }
                else if (rcvd_bytes == 0)
                {
                    /*
                     * The remote side has indicated that is done sending
                     * data. Since no more data needs to be sent on this
                     * socket we can just close down the socket.
                     */
                    shutdown(sd[i], READ_WRITE_SHUTDOWN);
                    close_socket(i);
                }
                else
                {
                    /*
                     * Now just turn the data around and send it back.
                     */
                    sval = 0;
                    while (sval < rcvd_bytes)
                          {
                          rval = send(sd[i], &buffer [i] [sval],
                                              rcvd_bytes - sval, 0);
                          if (rval < 0)
                             {
                             sprintf(error_text,
                                "Error on send() - socket %d", sd[i]);
                             perror(error_text);
                             close_socket(i);
                             }
                          sval = sval + rval;
                          }
                    }
            }
            nfound--;
        }
    }
```

(Continued on next page)

```
    }
    return(0);
}

int
find_unused_socket()
{
     int  i;

     for (i = 0; i < MAX_SDS; i++)
          if (sd[i] == 0)
               return(i);
     return(-1);
}

int
close_socket(si)
int si;             /* The index into sd[] of the socket to close */
{
    printf("\nClosing socket %d.\n", sd[si]);
    close(sd[si]);
    num_sockets--;
    FD_CLR(sd[si], &active_sds);
    sd[si] = 0;
    printf("%d connected sockets.\n", num_sockets);
    return(0);
}

void example_exit(char * exit_str)
{
    void  exit();
    /*
     * use perror() to print a message for the current value in errno.
     */
    perror(exit_str);
    exit(1);
}
```

Figure A-4. Sample Program Accepting Multiple Connection Requests

Sample Program to Receive IP Multicast Messages

Figure A-5 illustrates the `mlistener.c` program. This program uses the `IPPROTO_IP`, `IP_ADD_MEMBERSHIP`, and `IP_DROP_MEMBERSHIP` IP multicast options of the `setsockopt` function.

```
#define __USE_POSIX

#include <sys/socket.h>
#include <signal.h>
#include <arpa/inet.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <c_utilities.h>
#include <ctype.h>

#define MCASTSRC_PORT 12344
#define MCASTDEF_PORT 12345
#define MCASTDEF_GROUP "225.0.0.37"

#define MSGBUFSIZE 256
void close_groups(), interrupted(), leavegroup();
char *local_addr = "0.0.0.0";
char *reg_addr = "0.0.0.0";
int testclose = 0; /* if on, test that close() unregisters groups */
int fd, num, debug = 0, debug3 = 0;
char **gargv;
unsigned int totalbytes = 0;

int  optind, opterr;
char      *optarg;
extern int getopt(int argc, char **argv, char *optstr);

#define USAGE "usage: mlistener [-n packets] [-b] [ -l local_IPaddress ]
[ -R local_IPaddress ]\n\ [ -p UDP_port_number ] [-c] [-t] [-d] [-Z]
multicast_group1 mgroup2 .. \n -c = connect, -d = debug, -Z dumpdata,
-b = bind to multicast, -t = test socket close\n"

int dumpdata(unsigned char *data, size_t len)
{
  size_t pos,col;
  unsigned char ch;

printf ("===== received %d bytes =====\n", len);
for (pos=0; pos<len; pos+=16)
    {
    for (col=0;  col<16;  col++)
      {
      if (pos+col < len)
```

(Continued on next page)

```
        {
        ch = *(data + pos + col);

        printf("%02X ",ch);
        }

    for(col=0;  col<16;  col++)
      {
      if(  pos+col < len  )
        {
        ch = *(data + pos + col);
        if(  !isprint((int)ch)  )
          ch = '.';

        printf("%c",ch);
        }
      }

    printf("\n");
    }
  }
}
dperror(s)
char *s;
{
        perror(s);
}

int main(int argc, char *argv[])
{
  char *group_addr = MCASTDEF_GROUP;
  unsigned short local_udp_port = MCASTDEF_PORT, protocol = 0;
  struct sockaddr_in addr;
  struct ip_mreq mreq;
  char msgbuf[MSGBUFSIZE], connected = 0, c;
  int bind_to_multicast = 0, on = 1, i, nbytes,addrlen;
  unsigned int mnum = 0, forever = 1;
  int err = 0;
  long time;
  int temp,optlen;
  unsigned char mcastttl;
  gargv = argv;

    while ((c = getopt(argc, argv, "l:p:R:r:n:cdtbDZ")) != (char)-1)
        switch(c) {
            case 'l':
                if ((local_addr = optarg) < 0)
                {
                    fprintf(stderr, USAGE);
                    exit(-1);
                }
```

(Continued on next page)

```
                reg_addr = local_addr;
                break;
            case 'b':
                bind_to_multicast = 1;
                break;
            case 'R':
                if ((reg_addr = optarg) < 0)
                {
                    fprintf(stderr, USAGE);
                    exit(-1);
                }
                break;
            case 'n':
                 mnum = atoi(optarg);
                forever = 0;
                break;
            case 'p':
                local_udp_port = atoi(optarg);
                break;
            case 'c':
                connected = 1;
                break;

            case 't':
                testclose = 1;
                break;
            case 'd':
                debug = 1;
                break;
            case 'Z':
                debug3 = 1;
                break;
            default:

                fprintf(stderr, USAGE);
                exit(-1);
             }

     if ((num=(argc - optind)) < 1)
    {
         fprintf(stderr, USAGE);
                exit(-1);
     }

     if (debug)
     {
     printf("\nmlistener %d bytes IP addr %s%s%s%s%s %s %d:",
             forever ? -1 : mnum, local_addr,
         reg_addr == local_addr ? "" : " interface ",
         reg_addr == local_addr ? "" : reg_addr,
         connected ? " connected" : "",
```

(Continued on next page)

```
    testclose ? " implicit close" : "",
    "UDP port", local_udp_port);

for (i = 0; i<num; i++)
{
    printf(" %s", argv[optind+i]);

    if (i == 2)
      printf("\n");
}
if (num != 3)
    printf("\n");
}

 /* create what looks like an ordinary UDP socket */
 if ((fd=socket(AF_INET, SOCK_DGRAM, 0)) < 0)
 {
  dperror("UDP socket");
  exit(-1);
 }

 group_addr = argv[optind];

 /* set up destination address */
 memset(&addr,0,sizeof(addr));
 addr.sin_family = AF_INET;
 addr.sin_addr.s_addr=inet_addr(bind_to_multicast ? group_addr:local_addr);
 addr.sin_port = htons(local_udp_port);

 /* bind to receive address  */
 if (bind(fd,(struct sockaddr *) &addr,sizeof(addr)) < 0)
 {
  dperror("bind");
  close(fd);
  exit(-1);
 }
if (debug)
printf("bound to %s port %d\n",
    bind_to_multicast ? group_addr : local_addr, local_udp_port);

 if (connected)
 {
         if (num > 1)
     {
    fprintf(stderr, "cannot use connect() with multiple destns\n");
    close(fd);
    exit(-1);
     }
```

(Continued on next page)

```
          memset(&addr,0,sizeof(addr));
          addr.sin_family=AF_INET;
          addr.sin_addr.s_addr=inet_addr(group_addr);
          addr.sin_port=htons(MCASTSRC_PORT);

          if (connect(fd,(struct sockaddr *) &addr,sizeof(addr)) < 0)
              {
           dperror("connect");
           exit(1);
          }

          if (debug)
         printf("connected to %s port %d\n", group_addr, MCASTSRC_PORT);
      }

      if (setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, (char *)&on, sizeof(on)) < 0)
      {
         dperror("SO_REUSEADDR");
         exit(1);
      }

#ifdef SO_REUSEPORT
      if (setsockopt(fd, SOL_SOCKET, SO_REUSEPORT, (char *)&on, sizeof(on)) < 0)
      {
         dperror("SO_REUSEPORT");
         exit(1);
}
#endif

      for (i = 0; i < num; i++)
      {
         group_addr = argv[optind+i];

     if (debug)
         printf("Join group %s on interface %s\n",group_addr,reg_addr);

     mreq.imr_multiaddr.s_addr=inet_addr(group_addr);
     mreq.imr_interface.s_addr=inet_addr(reg_addr);

     if (debug)
        {
        printf("mult addr %x intrfc addr %x\n",mreq.imr_multiaddr.s_addr,
                    mreq.imr_interface.s_addr);
        printf ("%x %x %x %x %x %x\n", fd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
          mreq.imr_multiaddr.s_addr, mreq.imr_interface.s_addr, sizeof(mreq));
        }
```

(Continued on next page)

```
if (setsockopt(fd,IPPROTO_IP,IP_ADD_MEMBERSHIP,(char *)&mreq,sizeof(mreq)) < 0)
{
            dperror("setsockopt:IP_ADD_MEMBERSHIP");
            close(fd);
            exit(-1);
        }
     }

     signal(SIGTERM, interrupted);
     signal(SIGINT, interrupted);
     signal(SIGQUIT, interrupted);

     /* now just enter a read-print loop */
     while (forever || mnum--)
     {
      addrlen=sizeof(addr);

      if (connected)
      {
         if ((nbytes=recv(fd,msgbuf,MSGBUFSIZE,0)) < 0)
             {
               dperror("recv");
               close_groups();
             } /* nsd */
          totalbytes += nbytes;
       }
       else
       {
         if ((nbytes=recvfrom(fd,msgbuf,MSGBUFSIZE,0,
         (struct sockaddr *) &addr,&addrlen)) < 0)

          {
              printf("recvfrom error: %d",errno);
              dperror("recvfrom");
              close_groups();
          }
          if(debug3)
            {
             dumpdata(msgbuf, nbytes);
             fflush(stdout);
            }
          totalbytes += nbytes;
        }
      }

/*      if(debug3)
      dumpdata(msgbuf, nbytes);
          fflush(stdout);
*/
```

(Continued on next page)

```
    printf("TOTAL Bytes recvd = %d\n",totalbytes);
     close_groups();
}

void leavegroup(char *gaddr)
{
     struct ip_mreq mreq;

     mreq.imr_multiaddr.s_addr=inet_addr(gaddr);
     mreq.imr_interface.s_addr=inet_addr(reg_addr);

     printf("Leave group %s on interface %s\n", gaddr, reg_addr);

if (setsockopt(fd,IPPROTO_IP,IP_DROP_MEMBERSHIP,(char *)&mreq,sizeof(mreq)) < 0)
     {
          dperror("setsockopt:IP_DROP_MEMBERSHIP");
          close(fd);
          exit(-1);
     }
}

void interrupted()
{
    printf("mlistener interrupted\n");
    printf("Interrupted total bytes recvd so far %d",totalbytes);
    close_groups();
}

void close_groups()
{
   int i;
   char *gaddr;
if (!testclose)
    for (i = 0; i<num; i++)
    {
        gaddr = gargv[optind+i];
        leavegroup(gaddr);
       }
   else
    printf("close socket, leave groups implicitly\n");

   close(fd);
   exit(-1);
}
```

Figure A-5. Sample IP Multicast Receive Program

Sample Program to Send IP Multicast Messages

Figure A-6 illustrates the `msender.c` program. This program uses the `IPPROTO_IP`, `IP_MULTICAST_TTL`, `IP_MULTICAST_IF`, and `IP_MULTICAST_LOOP` IP multicast options of the `setsockopt` function to send data packets.

```
#define __USE_POSIX

#include <sys/socket.h>
#include <signal.h>
#include <arpa/inet.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <c_utilities.h>

void s$get_process_id();

#ifdef STREMUL
#define exit s_exit
#define close s_close
#endif

#define MCASTSRC_PORT 12344
#define MCASTDEF_PORT 12345
#define MCASTDEF_GROUP "225.0.0.37"

void interrupted();
int fd, debug = 0, debug2 = 0;
unsigned int totalbytes = 0;

char      *optarg;
int  optind, opterr;
extern int getopt(int argc, char **argv, char *optstr);
extern char *itoa (int, char *, int);

dperror(s)
char *s;
{

     perror(s);
}

#define USAGE "usage: msender [-n packets ] [ -l local_IPaddress ] [ -p UDP_port
+_number ]\n     [ -i ttl ] [ -L ] [ -c ] [ -s ] [-d] [-D] multicast_group1 mgro
+up2..\n\ -L = do not loopback; -c = connect to multicast -d debug -D debug2 gro
+up; -s = specify interface\n"
```

(Continued on next page)

```
main(int argc, char *argv[])
{
     struct sockaddr_in addr, gaddr;
     struct in_addr inaddr, ginaddr;
     int cnt, i, num, specify_if = 0, soreuse = 1, optlen;
     struct ip_mreq mreq;
     unsigned char mcastttl,non = 0;
     int ttl = 1,loopback = 1;
     static char message[100], lmessage[100];
     char *group_addr = MCASTDEF_GROUP, *local_addr = "0.0.0.0";
     unsigned short target_udp_port = MCASTDEF_PORT, protocol = 0;
     char connected = 0, c, pid[10], udpport[10];
     unsigned int mnum = 0, forever = 1;
     long proc_id;

    while ((c = getopt(argc, argv, "l:p:i:r:n:LcsdD")) != (char)-1)
          switch(c) {
               case 'l':
                    if ((local_addr = optarg) < 0)
                    {
                         fprintf(stderr, USAGE);
                         exit(-1);
                    }
                    break;
               case 'n':
                    mnum = atoi(optarg);
                    forever = 0;
                    break;
               case 'd':
                    debug = 1;
                    break;
               case 'i':
                    ttl = atoi(optarg);
                    break;
               case 'p':
                    target_udp_port = atoi(optarg);
                    break;
               case 'L':
                    loopback = 0;
                    break;
               case 's':
                    specify_if = 1;
                    break;
               case 'c':
                    connected = 1;
                    break;
               case 'D':
                    debug2 = 1;
                    break;
```

(Continued on next page)

```
                default:
                     fprintf(stderr, USAGE);
                     exit(-1);
           }

if ((num=(argc - optind)) < 1)
    {
     fprintf(stderr, USAGE);
        exit(-1);
    }
    if (debug)
    {
     printf("\nmsender %d from %s ttl %d%s%s%s",
          forever ? -1 : mnum,
               local_addr, ttl, loopback ? "" : " loopback disabled",
          connected ? " connected" : "",
          specify_if ? " specified i/f" :" default i/f");

     printf(" to %s %d:",  "UDP protocol", target_udp_port);

     for (i = 0; i<num; i++)
     {
          printf(" %s", argv[optind+i]);
          if (i==2)
               printf("\n");
     }
     if (num != 3)
          printf("\n");
    }

    /* message has process ID, local address and group address embedded */
    /* porting notes: if you don't have strcat, could send these separately */

     strcpy(message, "\"multicast from ");
     strcat(message, local_addr);
     sprintf(udpport, " %d", target_udp_port);
     strcat(message, udpport);
     strcat(message, " (process ID ");

     s$get_process_id(&proc_id);
     itoa(proc_id,&pid[0],10);
     strcat(message, pid);
     strcat(message, ") to ");

     if ((fd=socket(AF_INET, SOCK_DGRAM, 0)) < 0)
     {
          dperror("UDP socket");
          exit(-1);
     }
```

(Continued on next page)

```
      signal(SIGTERM, interrupted);
      signal(SIGINT,  interrupted);
      signal(SIGQUIT, interrupted);

      memset(&addr,0,sizeof(addr));
      addr.sin_family=AF_INET;
      addr.sin_addr.s_addr=inet_addr(local_addr);
      addr.sin_port=htons(MCASTSRC_PORT);

      /* bind to local IP address for transmission on specific interface */
      if (bind(fd,(struct sockaddr *) &addr,sizeof(addr)) < 0)
      {
           dperror("bind");
           close(fd);
           exit(1);
      }

     if (debug2)
      printf("bound to %s port %d\n", local_addr, MCASTSRC_PORT);

      if (setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, (char *)&soreuse,
           sizeof(soreuse)) < 0)
      {
           dperror("SO_REUSEADDR");
           close(fd);
           exit(1);
      }

 #ifdef SO_REUSEPORT
      if (setsockopt(fd, SOL_SOCKET, SO_REUSEPORT, (char *)&soreuse,
             sizeof(soreuse)) < 0)
      {
           dperror("SO_REUSEPORT");
           close(fd);
           exit(1);
      }
 #endif
 if (ttl != 1)
      {
           printf("TTL is %d",ttl);
           if (setsockopt(fd, IPPROTO_IP, IP_MULTICAST_TTL,
                (char *)&ttl, sizeof(ttl)) < 0)
                dperror("setsockopt IP_MULTICAST_TTL");
      else
      {
                optlen = sizeof(ttl);

                if (getsockopt(fd, IPPROTO_IP, IP_MULTICAST_TTL,
                     (char *)&mcastttl, &optlen) < 0)
                     dperror("getsockopt IP_MULTICAST_TTL");
```

(Continued on next page)

```
                else if (ttl != mcastttl || debug)
                     printf("IP_MULTICAST_TTL:expected %d got %d\n",
                          ttl, mcastttl);
           }
      }

      if (specify_if)
      {
           addr.sin_addr.s_addr = htonl(inet_addr(local_addr));
                inaddr = addr.sin_addr;

           if (setsockopt(fd, IPPROTO_IP, IP_MULTICAST_IF, (char *)&inaddr,
                sizeof(struct in_addr)) < 0)
           {
                dperror("setsockopt: IP_MULTICAST_IF");
                close(fd);
                exit(-1);
           }

           optlen = sizeof(struct in_addr);

           if (getsockopt(fd,IPPROTO_IP,IP_MULTICAST_IF,
                (char *)&ginaddr,&optlen)<0)
           {
                dperror("getsockopt IP_MULTICAST_IF");
                close(fd);
                exit(-1);
           }

           if (inaddr.s_addr != ginaddr.s_addr || debug)
                printf("IP_MULTICAST_IF expected %x got %x\n",
                     inaddr.s_addr, ginaddr.s_addr);
      /* num--; cmc don't need this either */
      }
 if (connected)
      {
           if (num > 1)
           {
           fprintf(stderr, "cannot use connect() with multiple destns\n");
           close(fd);
           exit(-1);
           }

      group_addr = argv[optind];  /* assume only one */
      memset(&addr,0,sizeof(addr));
      addr.sin_family=AF_INET;
      addr.sin_addr.s_addr=inet_addr(group_addr);
      addr.sin_port=htons(target_udp_port);
```

(Continued on next page)

```
      if (connect(fd, (struct sockaddr *) &addr,sizeof(addr)) < 0)
      {
           dperror("connect");
           close(fd);
           exit(-1);
      }

      if (debug2)
           printf("connected to %s port %d\n",group_addr,target_udp_port);
      }

 /*     if (loopback == 0) take this check out so that we set/reset*/
      {
      if (setsockopt(fd, IPPROTO_IP, IP_MULTICAST_LOOP, (char *)&loopback,
           sizeof(loopback)) < 0)
      {
           dperror("setsockopt IP_MULTICAST_LOOP");
           close(fd);
           exit(-1);
      }

      optlen = sizeof(non);

      if (getsockopt(fd, IPPROTO_IP, IP_MULTICAST_LOOP, (char *)&non, &optlen)
           < 0)
      {
           dperror("getsockopt IP_MULTICAST_LOOP");
           close(fd);
           exit(-1);

      }

      if (non != loopback || debug)
           printf("IP_MULTICAST_LOOP expected %d got %d\n", loopback,non);
      }
 while (forever || mnum--)
      {
      for (i = 0; i<num; i++)  /* can send to multiple groups */
           {
           group_addr = argv[optind+i];

      /* set up destination address */
      memset(&addr,0,sizeof(addr));
      addr.sin_family=AF_INET;
      addr.sin_addr.s_addr=inet_addr(group_addr);
      addr.sin_port=htons(target_udp_port);

      /* append destination address to message */
      strcpy(lmessage, message);
      strcat(lmessage, group_addr);
```

(Continued on next page)

```
      sprintf(udpport, " %d\"\n", target_udp_port);
      strcat(lmessage, udpport);

      if (connected)      /* only works for one argument! */
      {
           if(debug2)
           printf("sending2 %d bytes to %s:%s", strlen(lmessage), group_addr,
                lmessage);

           if (send(fd, lmessage, strlen(lmessage), 0) < 0)
           {
                dperror("send");
                close(fd);
                exit(-1);
           }
           totalbytes = (strlen(lmessage)) + totalbytes;
      }
      else
      {

      if (debug2)
      printf("sending3: %d bytes %s:%s", strlen(lmessage), group_addr, lmessage);

      if (sendto(fd,lmessage,strlen(lmessage),0,(struct sockaddr *)&addr,
           sizeof(addr)) < 0)
           {
                dperror("sendto");
                close(fd);
                exit(-1);
           }
      totalbytes = (strlen(lmessage)) + totalbytes;
      }
      sleep(1);
           }
      }
      printf("TOTAL BYTES SENT %d",totalbytes);
}

void interrupted()
{
      if (debug)
      printf("msender interrupted:closing\n");
      printf("Interupted totabytes sent at this point %d",totalbytes);
      close(fd);
      exit(-1);
}
```

Figure A-6. Sample IP Multicast Send Program

Appendix B
Deprecated Socket Options

The following socket options are deprecated.

```
SO_ACCEPTCONN
SO_RDWR
_SO_USELOOPBACK
_SO_NODELAY
_SO_URGENT
_SO_CONN_TIMER
_SO_CONN_TRYS
```

Do not use these options.

Index

Misc.

A

B

C

D

E

F

G

H

I

K

L

M

N

O

T

U

V

W