

Socket Library Functions

This chapter describes the BSD Socket Library provided with the Cisco IOS for S/390 API. This library supports the communication requirements of application programs written in the C language. The chapter includes these sections:

- **Introduction to the Socket Library**
Describes the basic terminology and purpose of the socket Application Program Interface (API).
- **Overview of BSD Sockets**
Describes communication domains and socket types, creating sockets and binding names, accepting and initiating connections, sending and receiving data, using file I/O functions, shutting down connections, socket and protocol options, non-blocking options, and differences between BSD UNIX and Cisco IOS for S/390 sockets.
- **UNIX File I/O Functions**
Describes how to use UNIX system calls to read and write sockets as they can read and write disk files.
- **Socket Library Functions**
Provides detailed coding information for the API socket functions. Also included in this section are the UNIX file I/O functions that are supported by Cisco IOS for S/390 API.

Introduction to the Socket Library

The Cisco IOS for S/390 API provides a socket library to support the communication requirements of application programs written in the C language. Generally, these applications are developed in a UNIX environment and ported to operate in an MVS environment. Providing a socket library greatly reduces the necessary changes that must be made to the application program.

Sockets

The term sockets refers to the Application Program Interface (API) implemented for the Berkeley Software Distribution (BSD) of UNIX. The socket interface is used by processes executing under control of the UNIX operating system to access network services in a generalized manner. Sockets support process-to-process communications using a variety of transport mechanisms, including UNIX pipes and Internet and XNS protocols. Socket support is being extended to include ISO protocols.

The socket interface was developed by the University of California at Berkeley and was included with releases 4.1, 4.2, and 4.3 of the BSD UNIX system. Because BSD UNIX has been ported to run on many machine architectures ranging from desktop workstations to large mainframes, many applications use BSD sockets to interface to the communications network. With a variety of C compilers now available for IBM systems, more and more of these applications are being ported to run under the MVS operating system. A library of BSD socket functions can simplify the porting effort by providing the communication services required by these applications, and much of the original C code can be retained.

This chapter describes the BSD Socket Library as provided with the API. The description includes a brief overview of BSD sockets, an explanation of some of the differences between the two implementations, and descriptions of the socket functions included in the library. Familiarity with BSD sockets is assumed. The intent of this chapter is to highlight any differences between the Cisco IOS for S/390 API and BSD implementation of sockets and *not* to provide detailed instruction on the use of socket functions.

Overview of BSD Sockets

The brief overview of BSD sockets given in this section is not intended to substitute for existing documentation on this subject. In particular, any user of the API socket library is encouraged to obtain and read these documents for a more thorough discussion of sockets and how they are used in the UNIX environment:

- *An Introductory 4.3BSD Interprocess Communication Tutorial*
- *An Advanced 4.3BSD Interprocess Communication Tutorial*

These documents are provided with the 4.3 BSD release of UNIX and are available from:

Computer Systems Research Group
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

Communication Domains and Socket Types

A socket is the UNIX abstraction of a communications endpoint. As such, it must exist within a communications domain identified when the socket is created. A communications domain is represented by the standard set of services provided to the communications endpoints, standardized rules of addressing and protocols used to communicate between endpoints, and the physical communications media. A constant defined in the include file `<socket.h>` is used to identify the communications domain. The communications domain of interest to the API implementation of sockets is the `AF_INET` Internet domain.

Support for ISO-based domains will be added in future releases of the Cisco IOS for S/390 API as BSD sockets are extended to accommodate them.

A socket is associated with an abstract type that describes the semantics of communications using the socket. Properties such as reliability, ordering, and prevention of duplication of messages are determined by the socket type.

Constants

The set of socket types are indicated by these constants defined in `<socket.h>`:

Table 3-1 Socket Types

Socket Type	Description
SOCK_DGRAM	SOCK_DGRAM models the semantics of unreliable datagrams in connectionless-mode communication. Messages may be lost or duplicated and may arrive out of order. A socket of type SOCK_DGRAM requires no connection setup before communication can begin and can communicate with multiple peers.
SOCK_RDM	SOCK_RDM models the semantics of reliable datagrams in connectionless-mode communication. Messages arrive unduplicated and in order, and the sender is notified if messages are lost. A socket of type SOCK_RDM requires no connection setup before communication can begin, and can communicate with multiple peers.
SOCK_STREAM	SOCK_STREAM models connection-based virtual circuits without record boundaries and provides reliable two-way transfer of ordered byte streams without duplication of data and without preservation of boundaries. A socket of type SOCK_STREAM communicates with a single peer and requires connection setup before communication can begin.
SOCK_SEQPACKET	SOCK_SEQPACKET models connection-based virtual circuits with record boundaries and provides reliable two-way transfer of ordered byte streams without duplication of data, while preserving boundaries within the data stream. A socket of type SOCK_SEQPACKET communicates with a single peer and requires connection setup before communication may begin.
SOCK_RAW	SOCK_RAW models connectionless-mode and is normally used with the sendto() and recvfrom() socket calls. The connect() call may be used to fix the destination for future datagrams. If the connect() call is used, the read() or recv() and write() or send() calls may be used. The application must provide a complete IP header when sending. Otherwise, IPPROTO_RAW will be set in outgoing datagrams and used to filter incoming datagrams and an IP header will be generated and prepended to each outgoing datagram. In either case, received datagrams are returned with the IP header and options intact.
SOCK_ASSOC	SOCK_ASSOC models the semantics of unreliable datagrams in connectionless-mode communication with associations. Messages may be lost or duplicated and may arrive out of order. Formerly SOCK_DGRAM in previous versions of Cisco IOS for S/390 sockets.
SOCK_CONNLESS	SOCK_CONNLESS is no longer used, but is included for backwards compatibility. SOCK_DGRAM replaces SOCK_CONNLESS. Sockets of type SOCK_DGRAM and SOCK_RDM are generally more efficient and are most useful in transaction-based applications where repetitive connection setup and breakdown should be avoided. Sockets of type SOCK_STREAM and SOCK_SEQPACKET support an out-of-band transmission facility that can be used to send expedited data.

Note SOCK_RDM and SOCK_SEQPACKET are not supported by Cisco IOS for S/390 sockets.

Each socket can have a specific protocol associated with it. This protocol is used within the domain to provide the semantics required by the socket type. Not all socket types are supported by each domain; support depends on the existence and the implementation of a suitable protocol within the domain. For example, within the Internet domain, the SOCK_DGRAM type may be implemented with User Datagram Protocol (UDP), and SOCK_STREAM type can be implemented with Transmission Control Protocol (TCP). No standard protocols exist in the Internet domain to implement the SOCK_RDM and SOCK_SEQPACKET socket types.

Creating Sockets and Binding Names

A socket is created with the `socket` function:

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

Communications Domain and Socket Type

The communications domain and socket type are specified using the constants defined in `<socket.h>`. The protocol may be specified as 0, indicating any suitable protocol for the given domain and type. One of several possible protocols can be specified by indicating its protocol number obtained by this library function:

```
getprotobyname().
```

Socket Descriptor

A socket descriptor, `s`, is returned by the `socket` function and should be used in all subsequent functions that reference the socket. The socket is initially unconnected, and if it was created with a connectionless type, datagrams can be sent and received immediately without establishing a connection. Otherwise, the socket is connection-oriented and must become connected before sending and receiving data.

An unconnected socket becomes connected in one of two ways:

- By actively connecting to another socket
- By becoming bound to a name in the communications domain and accepting a connection from another socket

Communications Domain Name

A name is currently limited to 16 bytes and has this structure:

```
struct sockaddr
{
    u_short      sa_family;
    char         sa_data [ 14 ];
};
```

The address family, `sa_family`, identifies the domain within which the name exists and is selected from a set of constants similar to those used to specify the domain when a socket is created.

The format and content of a name is domain-dependent. Generally the name consists of network, host, and protocol addresses. For the Internet domain, `AF_INET`, a name has this structure (defined in `<inet.h>`):

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

Note For 3.1 and higher, `AF_INET` has been redefined to be 2 so as to be compatible with most UNIX programs. For backwards compatibility, sockets will accept either 1 (the previous value) or 2 for `AF_INET`.

For historical reasons, the Internet address `sin_addr` is defined as the structure `in_addr`:

```
struct in_addr
{
    u_long s_addr;
};
```

To accept connections or receive datagrams, a socket must be bound to a name (or address) within the communications domain. To initiate a connection or send datagrams, the application program may bind the name or let the system bind a default name. A name is bound using the `bind` function:

```
bind ( s, name, namelen );
int s;
struct sockaddr *name;
int namelen;
```

Whether the name was specified by the application program or assigned by the system (in other words, the API), any name bound to a socket can be retrieved with the `getsockname` function:

```
getsockname ( s, name, namelen );
int s;
struct sockaddr *name;
int namelen;
```

The name of the connected peer can be retrieved with the `getpeername` function:

```
getpeername ( s, name, namelen );
int s;
struct sockaddr *name;
int namelen;
```

Accepting and Initiating Connections

After a name has been bound to a connection-oriented socket, a connection must be established before the application program can send or receive data. The method used to establish a connection depends on the operating mode of the program.

Operating in Server Mode

Programs operating in server mode generally are passive and listen for connection requests using the `listen` function:

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

The `backlog` argument specifies the maximum number of connection requests that can be queued simultaneously awaiting acceptance. A connection request is accepted by executing an `accept` function:

```
ns = accept ( s, name, namelen );
int ns, s;
struct sockaddr *name;
int namelen;
```

The accepted connection is established to a new socket that has the same characteristics (in other words, domain, type, and protocol) as the listening socket. The socket descriptor of the new socket, `ns`, is used in subsequent `send` and `recv` functions to exchange data with the connected peer, and the old socket descriptor, `s`, can be reused to listen for another connection request.

Operating in Client Mode

Programs operating in client mode generally are active and initiate a connection to a peer process by specifying the name of the peer with a connect function:

```
connect ( s, name, namelen );
int s;
struct sockaddr *name;
int namelen;
```

Once Connection is Established

The server and client modes of operation affect only how connections are established. Once a socket is connected, send and receive operations are executed without regard to how the connection was established. The mode of connection used by each peer process is agreed to in advance.

Although sockets of the connectionless datagram type do not establish real connections, the connect function can be used with such sockets to create an association with a particular peer. The name provided with the connect function is recorded for use in future send functions, which then need not supply the destination name. Only datagrams arriving at the socket from the associated peer are queued for receiving with subsequent recv functions.

Sending and Receiving Data

Messages can be sent and received from both connected and unconnected sockets.

Sending Messages from Unconnected Sockets

Messages can be sent from unconnected sockets using the sendto function:

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

When sending on unconnected sockets, to and tolen indicate the destination of the message. The number of bytes sent is returned.

Sending Messages from Connected Sockets

Messages can be sent from connected sockets using the send function:

```
cc = send ( s, buf, len, flags );
int cc, s;
char *buf;
int len, flags;
```

The message to be sent is identified by the buf and len arguments, and the flags argument is used to indicate normal or out-of-band data.

Receiving Messages from Unconnected Sockets

Messages can be received from unconnected sockets using the `recvfrom` function:

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

The receive buffer is identified by the `buf` and `len` arguments, and `flags` indicate whether the buffer is to be used for receiving normal or out-of-band data. When receiving on unconnected sockets, `from` and `fromlen` identify a buffer for returning the source of the message. The function return value is the length of the message received.

Receiving Messages from Connected Sockets

Messages can be received from connected sockets using the `recv` function:

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

The receive buffer is identified by the `buf` and `len` arguments, and `flags` indicate whether the buffer is to be used for receiving normal or out-of-band data.

Sending and Receiving Messages from Noncontiguous Buffers

The functions previously described provide for the sending and receiving of messages whose contents are stored in contiguous memory buffers. You can also send and receive messages gathered from or scattered into noncontiguous buffers using the `sendmsg` and `recvmsg` functions:

```
cc = sendmsg ( s, msg, flags );
int cc, s;
struct msghdr *msg;
int flags;
cc = recvmsg ( s, msg, flags );
int cc, s;
struct msghdr *msg;
int flags;
```

The structure `msghdr` is used to pass several parameters to `sendmsg` and `recvmsg` that reduce the number of direct function arguments:

```
struct msghdr
{
    char            *msg_name;
    int             msg_namelen;
    struct iovec    *msg_iov;
    int             msg_iovlen;
    char            *msg_accrights;
    int             msg_accrightslen;
};
```

The list of noncontiguous buffer segments is defined by an array of `iovec` structures:

```
struct iovec
{
    char *iov_base;
    int iov_len;
};
```

The `sendmsg` and `recvmsg` functions can be used with connected or unconnected sockets. If the socket is unconnected, `msg_name` and `msg_namelen` specify the destination or source of the message. Otherwise, these arguments should be null. `msg_iov` specifies an array of noncontiguous buffer segments, and `msg_iovlen` is the length of the array. The `msg_accrights` and `msg_accrightslen` arguments are used to specify access rights for sockets in the UNIX domain (`AF_UNIX`) and should be null for the Internet (`AF_INET`) domain.

Using File I/O Functions

Standard UNIX file I/O functions can be used to read and write sockets as if they were UNIX files. The socket descriptor is used in place of a file descriptor.

The read and write Functions

The read and write functions are used to receive and send contiguous data:

```
cc = read ( s, buf, len );
int cc, s;
char *buf;
int len;
cc = write ( s, buf, len );
int cc, s;
char *buf;
int len;
```

The readv and writev Functions

The `readv` and `writev` functions are used to receive and send noncontiguous data:

```
cc = readv ( s, iov, iovcnt );
int cc, s;
struct iovec *iov;
int iovcnt;
cc = writev ( s, iov, iovcnt );
int cc, s;
struct iovec *iov;
int iovcnt;
```

The array specified by `iov` contains `iovcnt` structures, each of which defines the beginning address and length of a buffer segment:

```
struct iovec
{
    char *iov_base;
    int iov_len;
};
```

By using the `read`, `readv`, `write`, and `writev` functions, the application program can read from and write to sockets, terminals, and files without distinguishing the descriptor type.

Shutting Down Connections

An application program that no longer needs a connected socket can gracefully shut the connection down using the shutdown function:

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

To discontinue receiving, set direction to 0.

To discontinue sending, set direction to 1.

To shut the connection down in both directions, set direction to 2.

If the underlying protocol supports unidirectional or bidirectional shutdown, this indication is passed to the peer.

Note A socket will not terminate until a close() has been issued for the socket.

A connection can also be gracefully closed and the socket eliminated from the system by using the close function:

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

Socket and Protocol Options

Sockets, and the underlying protocols, can support options. These options control implementation or protocol-specific facilities. The getsockopt and setsockopt functions are used to manipulate these options:

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

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

The option optname is interpreted at the indicated protocol level for sockets. If a value is specified with optval and optlen, it is interpreted by the software operating at the specified level. The level SOL_SOCKET is reserved to indicate options maintained by the socket facilities. Other level values indicate a particular protocol that is to act on the option request; these values normally are interpreted as a protocol number obtained with getprotobyname.

Non-blocking I/O

Socket functions that cannot complete immediately because of flow control or synchronization requirements block the application program until the operation can be completed. This causes the task that issued the function to be suspended. If blocking is not desired, the non-blocking mode can be selected for a socket using the `ioctl` function:

```
ioctl ( s, request, argp );
int s;
unsigned long request;
char *argp;
```

If `request` is set to the constant `FIONBIO`, defined in `<socket.h>`, and the boolean argument specified by `argp` is true (in other words, non-zero), the socket is enabled for non-blocking I/O. When operating in this mode, functions that would otherwise block are completed with an error code indicating the blocking condition. The application can then reissue the function at a later time when it can complete without blocking.

MVS vs. UNIX

Many subtle differences exist between the Cisco IOS for S/390 and the BSD UNIX socket implementations. For most users, these differences go unnoticed, but for the more sophisticated application these differences should be noted and accounted for. Another area that could cause the BSD UNIX socket programmer some problems is realizing where sockets end and the UNIX operating system begins. The programmer must understand the functioning of the MVS operating system as opposed to the UNIX environment.

Here are some of the OS environment differences that must be realized before porting a UNIX socket application to the MVS environment:

- UNIX sockets are part of the kernel and run in supervisor mode. Cisco IOS for S/390 sockets are simply functions that the user program links to and therefore run in user mode. Cisco IOS for S/390 sockets do make calls to the Cisco IOS for S/390 subsystem via PC calls that run in a semi-privilege mode. The main item to note is that Cisco IOS for S/390 sockets do not have the privilege or protection of BSD sockets because they run in user mode. Thus it is impossible for them to check for access violations when passed pointers to structures and buffers. Some of these types of errors can be caught by the API, while others cause the user program to ABEND. UNIX sockets are able to verify the validity of pointers and return an error (EFAULT) if the pointer would allow access to a privileged or nonexistent area of memory.
- The UNIX operating system lets a process fork a child process that is an exact copy of the parent. This lets a parent open a socket or file and fork a child that then has total access to this open socket or file. The MVS OS does not provide this feature. In MVS a task can ATTACH another task, but the two tasks must then coordinate access to any variables that the two may share. Therefore it is impossible with sockets under MVS to pass a socket from one task to another transparently. For programs that require multiple tasks using multiple endpoints, the designer should consider an architecture where one task is placed in charge of controlling the network activity; some form of inter-task communication could be used where other tasks can inform the task in charge about when to open and close endpoints, the location of data buffers for input and output, and other control information for the endpoints, or using the augmented socket library routines provided to transfer ownership of the socket from one task to another. Sockets cannot be passed across address spaces.
- When a process terminates abnormally or does not close the sockets or files it created during execution, the UNIX OS closes these gracefully on termination of the process. With Cisco IOS for S/390 sockets, the termination routine is called via an `atexit()` entry point, and the open sockets are closed abruptly. This may be viewed by the remote endpoint as an abortive

disconnect. To alleviate this problem as much as possible, socket library programmers should explicitly close all sockets their programs create before exiting. If the program terminates abnormally, users should be aware of the consequences.

General Socket Differences

This subsection covers most of the general differences between BSD UNIX sockets and Cisco IOS for S/390 sockets. In the later discussion of each function, differences between Cisco IOS for S/390 and BSD socket functions are listed.

Supported Sockets

Cisco IOS for S/390 sockets are supported only in the Internet domain (AF_INET).

Five types of sockets are supported:

Table 3-2 Supported Socket Types

Socket Type	Description
SOCK_STREAM	Use TCP for transport and provide a reliable byte stream service.
SOCK_DGRAM	Use UDP for transport and provide an unreliable message service in connectionless mode only. Formerly SOCK_CONNLESS in previous releases of Cisco IOS for S/390 sockets.
SOCK_CONNLESS	Use UDP for transport and provide an unreliable message service in connectionless mode only. No longer actively supported, but provided for backward compatibility - use SOCK_DGRAM
SOCK_ASSOC	Use UDP for transport and provide an unreliable message service with associations only. Formerly SOCK_DGRAM
SOCK_RAW	Use with UDP for transport and provide an unreliable message service.

SOCK_DGRAM provides the same functionality as SOCK_DGRAM sockets in the UNIX world. As an added feature, it is possible for sockets of the type SOCK_ASSOC to simulate a connection-based server. A user can issue a listen() function and when a datagram is received on this endpoint, the socket library simulates a pending connection request that the user can acknowledge with the accept() function call. This facility is not provided with UNIX implementation of sockets.

Note The usage of SOCK_DGRAM and SOCK_CONNLESS has changed for release 3.1 and higher of Cisco IOS for S/390 and Cisco IOS for S/390 sockets. Existing programs using SOCK_CONNLESS should be re-compiled and re-linked substituting SOCK_DGRAM, although for backwards compatibility, SOCK_CONNLESS has been mapped to SOCK_DGRAM. Socket programs using associations previously used SOCK_DGRAM. These programs should be re-compiled and re-linked to use SOCK_ASSOC.

Endpoints

Cisco IOS for S/390 defines endpoints below 4096 as server endpoints only. On a socket bind() the user can request an endpoint below 4096 only. This endpoint can act only as a server. If no endpoint is requested, an endpoint with port number 4096 or above is assigned. This endpoint can act only as a client.

Binding a Name to a Socket

When binding a name to a socket, only a port number is allowed. The Internet address must be set to `INADDR_ANY(0)`. Cisco IOS for S/390 assigns the proper address to the endpoint when it can determine which of the possible network interfaces to use.

Urgent Data

The UNIX concept of urgent data—Out-of-Band (OOB)—is not supported in the same fashion in the Cisco IOS for S/390 socket implementation. UNIX interprets OOB data to be a single byte of data within the stream. The UNIX socket programmer can specify that this byte of data be read in-band or out-of-band. The Cisco IOS for S/390 subsystem does not allow for the urgent data being read out-of-band. Therefore, you cannot specify the type of data to read with any of the receive request functions (for example, `MSG_OOB` is not supported, nor is `SO_OOBINLINE` option). Requests specifying either of these parameters do not cause an error, but do not have any effect on the operation of the socket library. In effect, the Cisco IOS for S/390 socket implementation operates as if the `SO_OOBINLINE` socket option has been set.

For SAS/C compiler users, these methods are available to determine if urgent data has been received:

- `SIGURG` signal

This signal is generated by the socket library when urgent data is received.

- `select()` on exceptional conditions

If a `select()` call completes noting read availability and a pending exceptional condition, then urgent data is available to be read.

For IBM compiler users, only the `select()` on exceptional conditions method is available for determining if urgent data has been received.

The `ioctl()` request `SIOCATMARK` can be used to determine specifically whether the socket read pointer points to the last type of urgent data (the read pointer points to the data to be read by the next socket read-oriented operation).

Error Codes

The error codes set in the global variable, `errno`, are all defined with the prefix `E` to indicate “error”. Error codes are mapped to the corresponding error code for the compiler that you are using, either SAS/C or IBM/C370.

Note The current release of Cisco IOS for S/390 uses only `E` as the prefix to be more compatible with UNIX sockets and maps the error code according to the compiler you are using, either SAS/C or IBM/C370. For backward compatibility, the `ES` prefix will be accepted.

The value for `errno` may now be checked directly. A macro, `GET_ERRNO` had been provided in earlier releases for querying the value of `errno`. This macro is still available

```
if ( ( s = socket ( AF_INET, SOCK_STREAM, 0 ) ) < 0 )
{
    if ( GET_ERRNO == EFAULT )
    {
        /* do something special for this type of error */
    }
}
```

Extra Functions

These extra functions have been added to the socket library to assist in the development of applications using the socket library:

- `mvselect()`
Provides the same features as `select()`, plus it allows selecting on an optional generic ECB list.
- `closepass()` and `openold()`
When used together, lets a socket be passed to another task.

ANSI-C Compatible Function Prototypes

The socket library functions all have ANSI-compatible function prototypes associated with them. These prototypes provide for better error checking by the C compiler. However, to the pre-ANSI programmer, these may be more of an annoyance than a benefit. If you want to ignore these function prototypes, you can define the term `NOSLIBCK` at compile time and the ANSI function prototypes are ignored and the more common function definitions are used.

Socket Header Files

Table 3-3 Socket Header Files

Socket Header File	Description
acs.h	This header file is no longer required but is included for backward compatibility. It simply includes socket.h.
icssckt.h	This header file is for use with Cisco IOS for S/390 OpenEdition sockets. Read Chapter 7, OpenEdition (UNIX System Services) MVS Integrated Sockets for more information.
inet.h	This header file takes the place of both in.h and inet.h (for those familiar with the UNIX header file organization). It defines values common to the Internet. This header file takes the place of both <netinet/in.h> and <inet.h> on UNIX.
ip.h	This header file defines values used by the Inter-network Protocol (IP) and details the format of an IP header and options associated with IP. The current implementation of the socket library does not let the socket library user access the IP layer, and therefore, this header file is of little use for the time being. Later versions of the socket library will provide the ability to set IP options and at that time this library will be required. This file is the same as <netinet/ip.h> on UNIX.
netdb.h	This header file defines the structures used by the “get” services. It also provides the function prototypes for this family of functions. This file has the same functionality as <netdb.h> on UNIX.
errno.h	This header file defines the errors that can be returned by the socket library when requests are made to it. The value of errno can be determined directly, or, as in previous releases, the macro GET_ERRNO provides for the proper access to the errno variable of the socket library. This file has the same functionality as <errno.h> on UNIX.
sockcfg.h	This header file describes the socket configuration structure. This file has no equivalent on UNIX.
socket.h	This header file defines most of the variables and structures required to interface properly to the socket library. It also provides the function prototypes in ANSI form for those functions that are truly socket functions. This file takes the place of <sys/types.h>, <sys/socket.h>, <sys/time.h>, <sys/param.h>, <sys/stat.h>, <sys/ioctl.h>, and <sys/fcntl.h> on UNIX.
sockvar.h	This header file is needed when compiling the socket configuration file (sockcfg.c) and also is used internally by the socket library. It is not needed to interface to the socket library. This file has the same functionality as <sys/socketvar.h> on UNIX.
tcp.h	This header file describes those options that may be set for a socket of type SOCK_STREAM. This file has the same functionality as <netinet/tcp.h> on UNIX.
uio.h	This header file describes the structures necessary to use vectored buffering of the socket library. Due to ANSI checking, this header file must be included in any header file that includes socket.h. This file has the same functionality as <sys/uio.h> on UNIX.
user.h	This header file is needed when compiling the socket configuration file (sockcfg.c) and is also used internally by the socket library. It is not needed to interface to the socket library. There is no equivalent for this file on UNIX.

Options

This implementation of sockets does not support all of the option facilities of the BSD UNIX version. No options are supported at the IP or UDP levels. Also, no options can be set at the interface or driver levels. At the socket level (SOL_SOCKET), some additional options have been added. For further explanation of those options supported by this implementation, read getsockopt() and setsockopt().

Error Codes

Various error codes that are stored in the external variable `errno` have been added. Those that are particular to a function call are detailed in the section describing the function. These error codes can occur with any function call:

- **ECONFIG**

This error can occur when the application first opens a socket. It indicates that an error was detected with the socket configuration or when initializing a user session with the API. This type of error occurs only on a call to the `socket()` function. To further isolate this type of error, users should configure their socket configurations to allow extended error messages (in other words, set `EXTERRNOMSGS` and `CONFIGDEBUG` on in `sockcfg.flags`) and configuration debug. Once an error occurs, user programs should issue a call to `perror()` to view the error message generated. Details and troubleshooting action for this type of problem are covered in the Cisco IOS for S/390 Messages and Codes Reference.

- **ESYS**

This error can occur when the underlying API malfunctions. If, during normal operation, the API returns an error code that should not occur during the function execution, this error is returned. The user should ensure the proper operation of the API or Cisco IOS for S/390 before trying to isolate this problem further.

- **ETPEND**

This error code is set when the API is stopped or terminates abnormally. The user program should close all endpoints at this time. At some interval the user program can attempt to reopen sockets to see if the API has been restarted. When `ETPEND` is no longer returned, the API has restarted and the user program can continue to use the network facilities.

UNIX File I/O Functions

BSD UNIX integrates socket and file I/O facilities in such a way that UNIX system calls that normally are used to read and write disk files can also be used to read and write sockets. This is particularly convenient when a process inherits a socket or file descriptor from another process. The inheriting process can process the input or output stream without knowing whether it is associated with a socket or file.

Example

A filtering process that receives its input from `STDIN` and writes its output to `STDOUT` need not distinguish socket descriptors from file descriptors.

Using UNIX Routines in MVS

In a standard UNIX environment, if a file I/O function is issued using a socket descriptor in place of the normal file descriptor, the file I/O routine can route the request to the appropriate socket I/O routine because both execute in the UNIX kernel. However, in an MVS environment where UNIX file I/O routines are being simulated, restrictions might exist.

UNIX file I/O operations are simulated by library routines that translate I/O requests into operations compatible with MVS access methods. Generally, these library routines are provided by the vendor of the C compiler used to compile the application program. Therefore, the socket library functions provided with the API must be integrated with these routines if the application program intends to use UNIX file I/O functions to manipulate sockets and files at the same time.

Integrating API Socket Functions with UNIX File I/O

One of these techniques can be used to integrate the Cisco IOS for S/390 socket functions with UNIX file I/O:

- The file I/O function provided with the C compiler runtime library can be modified to call the appropriate socket function if the descriptor is associated with a socket. This presumes that the compiler vendor provides source code for the run-time library.
- If source code is not available, the technique is to front-end the UNIX file I/O function with a similarly-named function and to rename the original member in the library. Intercepted requests can then be routed to the appropriate function.

Descriptions are provided for those UNIX file I/O functions that also operate on sockets and are supported by the Cisco IOS for S/390 API. Since source code is furnished for these library functions, they can serve as the basis for front-ending existing functions that simulate UNIX file I/O. The descriptions provided apply only when the associated function is used to operate on a socket.

Socket Library Functions

This section provides detailed coding information for the Cisco IOS for S/390 socket functions.

The socket library functions are presented in alphabetical order. Each page that pertains to a particular function has the name of the function in the upper outside corner.

Because of the different operating system environment within which the API sockets must operate, a few additional functions have been included in the socket library. These functions simplify integration of the socket library into the IBM MVS environment.

Note These functions are `closepass()`, `mvselect()`, and `openold()`, and are detailed in the socket functions section.

Function Description Components

Following a brief introductory statement summarizing its use, each function is described using the documentation style of UNIX. This table lists the basic components of each function description:

Synopsis

A synopsis of the function is given in C language format. The function prototype statement is listed showing all function arguments, followed by declaration statements defining the format of each argument. If the function returns a value other than the normal success/failure indication, the function prototype is shown as an assignment statement.

Description

A description of the function is given, including any special rules for specifying arguments, alternative uses of the function, and any results returned.

Return Value

The function's return value, if any, is defined.

Error Codes

Error codes that are returned when the function completes abnormally are defined. Generally, these are error codes defined as constants in `<errno.h>` and are returned in the global integer `errno`.

Implementation Notes

Any difference between the Cisco IOS for S/390 API and BSD implementation of a function is noted here. Side effects that are a consequence of the operating system environment are also noted. If no implementation notes are listed, the Cisco IOS for S/390 API and BSD implementations are functionally equivalent.

See Also

References to related functions are given.

accept()

accept a connection on a socket

Synopsis

```
#include <socket.h>
#include <uio.h>
int accept ( s , name, namelen )
int s;
struct sockaddr *name;
int *namelen;
```

Description

The `accept()` function is used to accept a pending connection request queued for a socket that is listening for connections. A new socket is created for the connection, and the old socket continues to be used to queue connection requests.

The `accept()` function extracts the first connection on the queue of pending connections, creates a new socket with the same properties of `s`, and allocates a new file descriptor for the socket. If no pending connections are present on the queue, and the socket is not marked as non-blocking, `accept()` blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, `accept()` returns an error. The accepted socket, `ns`, cannot be used to accept more connections. The original socket `s` remains open.

It is possible to `select()` a socket for the purposes of doing an `accept()` by selecting it for read.

Parameters:

- | | |
|----------------------|---|
| <code>s</code> | A socket created with <code>socket()</code> , bound to a name with <code>bind()</code> , and listening for connections after a <code>listen()</code> . |
| <code>name</code> | A result parameter that is filled in with the name of the connecting entity, as known to the communications layer. The exact format of the name parameter is determined by the domain in which the communication is occurring. |
| <code>namelen</code> | A value-result parameter and should initially contain the amount of space pointed to by <code>name</code> ; on return it contains the actual length (in bytes) of the name returned. This function is used with connection-based or association-based socket types, currently with <code>SOCK_STREAM</code> and <code>SOCK_ASSOC</code> , respectively. |

Return Value

If the `accept()` function succeeds, it returns a non-negative integer that is a descriptor for the accepted socket. Otherwise, the value `-1` is returned, and the error code stored in the global integer `errno` indicates the nature of the error.

Error Codes

The `accept()` function returns these error codes (from `<errno.h>`) in the global integer `errno`:

<code>EBADF</code>	The descriptor is invalid.
<code>ECONNABORTED</code>	The incoming connection request was aborted by the remote endpoint.
<code>ECONNREFUSED</code>	The remote endpoint refused to complete the connection sequence.
<code>ECONNRESET</code>	The remote endpoint reset the connection request.
<code>EDESTUNREACH</code>	Remote destination is now unreachable.
<code>EFAULT</code>	The name pointer, name, or the name length pointer, <code>namelen</code> , points to inaccessible storage.
<code>EHOSTUNREACH</code>	Remote host is now unreachable.
<code>EINVAL</code>	The socket is not listening for connections.
<code>EMFILE</code>	The socket descriptor table is full. There is no room to save the new socket descriptor that <code>accept()</code> normally returns.
<code>ENFILE</code>	New socket cannot be opened due to API resource shortage or user endpoint allocation limits.
<code>ENETDOWN</code>	Local network interface is down.
<code>ENOMEM</code>	No memory is available to allocate space for the new socket.
<code>EOPNOTSUPP</code>	The referenced socket is not of type <code>SOCK_STREAM</code> or <code>SOCK_ASSOC</code> .
<code>ETIMEDOUT</code>	The connection request by the remote endpoint timed out.
<code>ETIMEDOUT</code>	Remote endpoint timed out connection.
<code>EWouldBlock</code>	The socket is marked non-blocking and no connections are present to be accepted.

Implementation Notes

Unlike UNIX sockets, this implementation does let `SOCK_ASSOC` (UDP sockets using association-oriented operation) listen for and accept connections.

See Also

`bind()`, `connect()`, `listen()`, `select()`, `socket()`, `closepass()`, `openold()`

bind()

bind a name to a socket

Synopsis

```
#include <socket.h>
#include <uio.h>
int bind ( s, name, namelen )
int s;
struct sockaddr *name;
int namelen;
```

Description

The `bind()` function assigns a name to an unnamed socket that represents the address of the local communications endpoint. For sockets of type `SOCK_STREAM`, the name of the remote endpoint is assigned when a `connect()` or `accept()` function is executed.

When a socket is created with `socket()`, it exists in a name space (address family), but has no name assigned. `bind()` requests that name be assigned to the socket. The rules used in name binding vary between communication domains.

Return Value

If `bind()` is successful, a value of 0 is returned. A return value of -1 indicates an error, and the error code stored in the global integer `errno` indicates the nature of the error.

Error Codes

The `bind()` function returns these error codes (from `<errno.h>`) in the global integer `errno`:

<code>EACCES</code>	The requested address is protected, and the current user has inadequate permission to access it.
<code>EADDRINUSE</code>	The specified address is already in use.
<code>EADDRNOTAVAIL</code>	The specified address is not available. This can occur if the address portion of the name is not equal to <code>INADDR_ANY</code> or the port is greater than or equal to <code>IPPORT_RESERVED</code> .
<code>EAFNOSUPPORT</code>	The address family requested in the name does not equal <code>AF_INET</code> .
<code>EBADF</code>	The argument <code>s</code> is not a valid descriptor.
<code>EFAULT</code>	The pointer, <code>name</code> , points to inaccessible memory.
<code>EINVAL</code>	The socket is already bound to an address.
<code>EINVAL</code>	The length of the name, <code>namelen</code> , does not equal the size of a <code>sockaddr_in</code> structure.

Implementation Notes

Currently, only names from the Internet domain, `AF_INET`, may be bound to a socket. The address portion of the name must equal `INADDR_ANY`, else `EINVAL` is returned. Server and client ports are reserved differently with this implementation than with UNIX. Ports 0 to 4095 are reserved for server ports, and 4096 and up are for client ports. The user should only assign a port when acting as a server and should specify a port of 0 when doing a `bind()` for a client port.

See Also

`connect()`, `listen()`, `socket()`, `getsockname()`

close()

delete a socket

Synopsis

```
#include <socket.h>
#include <uio.h>
int close ( s )
int s;
```

Description

The close() function is used to delete a socket descriptor created by the socket() function. On completion, the endpoint no longer exists in the communications domain.

close() deletes the socket descriptor, s, from the internal descriptor table maintained for the application program and terminates the existence of the communications endpoint. If the socket was connected, the connection is terminated. The connection is released as much as possible, in an orderly fashion. Data that has yet to be delivered to the remote endpoint remains queued, because the endpoint tries to deliver it before a timeout causes the local endpoint to be destroyed and removed from the system.

Sockets use MVS STIMERM services. Socket applications may use up to fifteen STIMERM calls per task control block (TCB). STIMER CANCEL ID=ALL must not be used by socket applications.

Return Value

If close() is successful, a value of 0 is returned. A return value of -1 indicates an error, and the error code stored in the global integer errno indicates the nature of the error.

Error Codes

The close() function returns these error codes (from <errno.h>) in the global integer errno:

EBADF	The s argument is not an active descriptor.
ECONNABORTED	The connection was aborted by the remote endpoint.
ECONNREFUSED	The remote endpoint refused to continue the connection.
ECONNRESET	The remote endpoint reset the connection request.
EDESTUNREACH	Remote destination is now unreachable.
EHOSTUNREACH	Remote host is now unreachable.
ENETDOWN	Local network interface is down.
ETIMEDOUT	The connection timed out.

See Also

accept(), bind(), connect(), shutdown(), socket(), closepass(),

closelog()

close log file

Synopsis

```
#include <syslog.h>
void closelog ();
```

Description

The closelog() function is used to close the log file opened with openlog().

See Also

openlog(), syslog(), vsyslog()

closepass()

prepare to pass a socket to another task

Synopsis

```
#include <socket.h>
#include <uio.h>
unsigned long closepass ( fd )
int fd;
```

Description

The closepass() function is used to retrieve a special token about a socket. This token is used later by openold() to open the socket by another task.

The closepass() function is used by the owning task of a socket to prepare the socket to be passed to another task within the same address space. fd is the socket descriptor of the socket to be passed. A token is returned at successful completion of the call to closepass.

The token must be passed by the owning task to the receiving task via application-specific interprocess communication. Once the token has been passed to the receiving task, the owning task should use the close() function to close the socket. At approximately the same time, the receiving task should call openold() to actually receive ownership of the socket. The call to close() blocks until the receiving task completes its openold() request.

This is a sample of events that must occur in time-ordered sequence. Assume that the main task has opened a socket, using socket() or accept():

Main Task	Subtask
1) Call closepass(fd).	
2) Send IPC to subtask passing token returned by closepass.	
3) Call close(fd). On return from close, main task can no longer reference this fd.	
	4) Receive token from main task via application-dependent IPC.
	5) Call openold(token).
6) Upon return from close, main task may no longer reference fd.	
	7) Subtask may now use fd returned by openold to access network.
	8) Subtask is through with socket.
	9) Call close(fd) to remove socket.

Return Value

If successful, closepass() returns a token that relates to the socket. Otherwise, the value -1 is returned, and the error code stored in the global integer errno indicates the nature of the error.

Error Codes

The `closepass()` function returns these error codes (from `<errno.h>`) in the global integer `errno`:

`EBADF`

The `fd` specifies an invalid descriptor.

Implementation Notes

The implementation of this function is provided to ease the development of server-oriented socket applications using the socket library.

See Also

`accept()`, `close()`, `openold()`, `socket()`

connect()

initiate a connection on a socket

Synopsis

```
#include <socket.h>
#include <uio.h>
int connect ( s, name, namelen )
int s;
struct sockaddr *name;
int namelen;
```

Description

The connect() function assigns the name of the peer communications endpoint. If the socket is of type SOCK_STREAM, a connection is established between the endpoints. If the socket is of type SOCK_ASSOC, a permanent association is maintained between the endpoints until changed with another connect() function.

Parameters:

- s A socket of type:
- SOCK_ASSOC - Specifies the peer with which the socket is to be associated; this address is where datagrams are to be sent and is the only address from which datagrams are to be received.
 - SOCK_STREAM - Attempts to make a connection to another socket.
- name Specifies the other socket, which is an address in the communications domain of the socket. Each communications domain interprets the name parameter in its own way.
- namelen Specifies the size of the name array

Generally, stream sockets can successfully connect() only once; datagram sockets can use connect() multiple times to change their association. Datagram sockets can dissolve the association by connecting to an invalid address, such as a null address.

Return Value

If the connection or association succeeds, a value of 0 is returned. Otherwise, the value -1 is returned, and the error code stored in the global integer errno indicates the nature of the error.

Error Codes

The connect() function returns these error codes (from <errno.h>) in the global integer errno:

- | | |
|---------------|--|
| EADDRINUSE | The address is already in use. |
| EADDRNOTAVAIL | The specified address is not available. This is caused by the name specifying a remote port of zero (0) or a remote address of INADDR_ANY (0). |
| EAFNOSUPPORT | Addresses in the specified address family cannot be used with this socket. |

EALREADY	The socket is non-blocking and a previous connection attempt has not yet been completed.
EBADF	The argument <code>s</code> is not a valid descriptor.
ECONNABORTED	The incoming connection request was aborted by the remote endpoint.
ECONNREFUSED	The attempt to connect was forcefully rejected.
ECONNRESET	The remote endpoint reset the connection request.
EDESTUNREACH	Remote destination is now unreachable.
EFAULT	The pointer name points to inaccessible memory.
EHOSTUNREACH	Remote host is now unreachable.
EINPROGRESS	The socket is non-blocking and the connection cannot be completed immediately. It is possible to <code>select()</code> for completion by selecting the socket for writing.
EINVAL	The length of the endpoint name, <code>namelen</code> , does not equal the size of a <code>sockaddr_in</code> structure.
EISCONN	The socket is already connected and does not allow multiple connects with the same socket.
ENETDOWN	Local network interface is down.
ENETUNREACH	The network is not reachable from this host.
EOPNOTSUPP	The socket is listening for incoming connection requests and therefore cannot also be used for outgoing connection requests.
EOPNOTSUPP	The socket does not support the connecting or associating of two endpoints. It operates only in connectionless mode.
ETIMEDOUT	Connection establishment timed out without establishing a connection.

Implementation Notes

Unlike UNIX sockets, this implementation does allow `SOCK_ASSOC` (UDP sockets using association oriented operation) to listen for and accept connections.

See Also

`accept()`, `select()`, `socket()`, `getsockname()`

fcntl()

file control

Synopsis

```
#include <socket.h>
#include <uio.h>
int fcntl ( s, cmd, arg )
int s;
int cmd;
int arg;
```

Description

The `fcntl()` function provides control over file descriptors. Asynchronous notification and non-blocking I/O can be turned on and off using this function call.

`fcntl()` performs a variety of control functions on a socket, indicated by `s`. An `fcntl cmd` has encoded in it whether the argument, `argp`, is an input parameter supplied by the caller or an output parameter returned by the `fcntl()` function.

Permissible values for `cmd` are defined in `<socket.h>`. The current implementation of `fcntl()` supports these requests:

<code>F_GETFD</code>	Used to get the file descriptor flags.
<code>F_SETFD</code>	Used to set the file descriptor flags.
<code>F_GETFL</code>	Used to get the socket status flags.
<code>F_SETFL</code>	Used to set the socket status r flags.

The flags for `F_GETFL` and `F_SETFL` are:

<code>FNDELAY</code>	Select non-blocking I/O; if no data is available to a read system call or if a write operation would block, the call returns a -1 with the error <code>EWOULDBLOCK</code> .
<code>FASYNC</code>	Enable the <code>SIGIO</code> , <code>SIGURG</code> , and <code>SIGPIPE</code> signals to be sent to the process when the triggering events occur.

Return Value

If `fcntl()` is successful, a value of 0 is returned. A return value of -1 indicates an error, and the error code stored in the global integer `errno` indicates the nature of the error.

Error Codes

The `fcntl()` function returns these error codes (from `<errno.h>`) in the global integer `errno`:

<code>EBADF</code>	The <code>s</code> argument is not a valid descriptor.
<code>EINVAL</code>	<code>cmd</code> or <code>arg</code> is invalid or not supported by this implementation of <code>fcntl()</code> .

Implementation Notes

When setting the socket descriptor flags with the command of `F_SETFL`, the request is destructive to the variable that saves the flags. To set both the `FNDELAY` and `FASYNC` flags, a single request must be made specifying both options.

See Also

`accept()`, `connect()`, `recv()`, `recvfrom()`, `recvmsg()`, `send()`, `sendmsg()`, `sendto()`, `write()`

gethostbyaddr()

get host information by address

Synopsis

```
#include <netdb.h>

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

Description

The `gethostbyaddr()` function is used to obtain the official name of a host when its network address is known. A name server is used to resolve the address if one is available; otherwise, the name is obtained (if possible) from a database maintained on the local system.

`gethostbyaddr()` returns a pointer to an object with the structure `hostent`. This structure contains either the information obtained from the local name server or extracted from an internal host database whose function is similar to `/etc/hosts` on a UNIX-based system. If the local name server is not running, `gethostbyaddr()` looks up the information in the internal host database.

```
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 ]
```

Parameters:

<code>h_name</code>	Official name of the host.
<code>h_aliases</code>	A zero terminated array of alternate names for the host.
<code>h_addrtype</code>	The type of address being returned; currently always <code>AF_INET</code> .
<code>h_length</code>	The length, in bytes, of the address.
<code>h_addr_list</code>	A zero-terminated array of network addresses for the host.
<code>h_addr</code>	The first address in <code>h_addr_list</code> ; this is for backward compatibility.

All information is contained in a static area so it must be copied if it is to be saved. Only the Internet address format is currently supported.

Return Value

The `gethostbyaddr()` function returns a pointer to the `hostent` structure, if successful. If unsuccessful, a null pointer is returned, and the external integer `h_errno` is set to indicate the nature of the error.

Error Codes

The `gethostbyaddr()` function can return these error codes in the external integer `h_errno`:

<code>HOST_NOT_FOUND</code>	No such host is known.
<code>NO_ADDRESS</code>	The requested name is valid, but does not have an address in the Internet domain.
<code>NO_RECOVERY</code>	This is a nonrecoverable error.
<code>TRY_AGAIN</code>	This is usually a temporary error and means that the local server did not receive a response from an authoritative server. A retry at some later time might succeed.

See Also

`gethostbyname()`

gethostbyname()

get host information by name

Synopsis

```
#include <netdb.h>
struct hostent *gethostbyname ( name )
char *name;
```

Description

The `gethostbyname()` function is used to obtain the network address (or list of addresses) of a host when its official name (or alias) is known. A name server is used to resolve the name if one is available; otherwise, the address is obtained (if possible) from a database maintained on the local system.

The `gethostbyname()` function returns a pointer to an object with the following structure. This structure contains either the information obtained from the local name server or extracted from an internal host database whose function is similar to `/etc/hosts` on a UNIX-based system. If the local name server is not running, `gethostbyname()` looks up the information in the internal host database.

```
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 ]
```

Parameters:

<code>h_name</code>	Official name of the host
<code>h_aliases</code>	A zero-terminated array of alternate names for the host
<code>h_addrtype</code>	The type of address being returned; currently always <code>AF_INET</code>
<code>h_length</code>	The length, in bytes, of the address
<code>h_addr_list</code>	A zero-terminated array of network addresses for the host
<code>h_addr</code>	The first address in <code>h_addr_list</code> (this is for backward compatibility)

All information is contained in a static area so it must be copied if it is to be saved. Only the Internet address format is currently understood.

Return Value

The `gethostbyname()` function returns a pointer to the `hostent` structure, if successful. If unsuccessful, a null pointer is returned, and the external integer `h_errno` is set to indicate the nature of the error.

Error Codes

The `gethostbyname()` function can return these error codes in the external integer `h_errno`:

<code>HOST_NOT_FOUND</code>	No such host is known.
<code>NO_ADDRESS</code>	The requested name is valid, but does not have an address in the Internet domain.
<code>NO_RECOVERY</code>	This is a nonrecoverable error.
<code>TRY_AGAIN</code>	This is usually a temporary error and means that the local server did not receive a response from an authoritative server. A retry at some later time might succeed.

See Also

`gethostbyaddr()`

gethostname()

get name of local host

Synopsis

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

Description

The `gethostname()` function is used to obtain the name of the local host. This is the official name by which other hosts in the communications domain reference it. Generally, a host belonging to multiple communication domains, or connected to multiple networks within a single domain, has one official name by which it is known.

The `gethostname()` function returns the standard host name for the local system. The parameter `namelen` specifies the size of the name array. The returned name is null-terminated unless insufficient space is provided. Host names are limited to `MAXHOSTNAMELEN` (from `<socket.h>`) characters, which is currently 64.

Return Value

If `gethostname()` is successful, a value of 0 is returned. A return value of -1 indicates an error. See *Implementation Notes* in this section for more information.

Error Codes

The `gethostname()` function returns these error codes (from `<errno.h>`) in the global integer `errno`. If the call is successful, 0 is returned to the user and the name of the local host is placed in the user-provided buffer. If unsuccessful, a minus one (-1) is returned, and the external integer `h_errno` is set to indicate the nature of the error.

The `gethostname()` function can return, in the external integer `h_errno`, the error code `HOST_NOT_FOUND`, which indicates that the host name is not found or the `namelen` parameter is less than or equal to zero (0).

Implementation Notes

On a UNIX system this request is a system call. It obeys all of the standards for system calls to include setting `errno` when an error is detected. This implementation, however, treats this function as a network database call and uses the external variable `h_errno` to describe errors to the user.

Note This function does not set `errno`.

getnetbyaddr()

get network information by address

Synopsis

```
#include <netdb.h>
struct netent *getnetbyaddr ( net, type )
long net;
int type;
```

Description

The `getnetbyaddr()` function is used to obtain the official name of a network when the network address (or network number) is known. This information is acquired from a network database maintained by the local system.

The `getnetbyaddr()` function returns a pointer to an object with this structure containing information extracted from an internal network database. This database is similar in function to the `/etc/networks` file on UNIX-based systems:

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

Parameters:

- | | |
|-------------------------|--|
| <code>n_name</code> | The official name of the network. |
| <code>n_aliases</code> | A zero-terminated list of alternate names for the network. |
| <code>n_addrtype</code> | The type of the network number returned; currently only <code>AF_INET</code> . |
| <code>n_net</code> | The network number in machine byte order. |

All information is contained in a static area and must be copied if it is to be saved. The `getnetbyaddr()` function sequentially searches from the beginning of the network database until a matching network address and type is found, or until the end of the database is reached. Only the Internet address format is currently supported.

Return Value

The `getnetbyaddr()` function returns a pointer to the `netent` structure, if successful. If unsuccessful, a null pointer is returned.

Error Codes

The `getnetbyaddr()` function may return these error codes in the external integer `h_errno`:

`NET_NOT_FOUND` No such network is known.

- | | |
|-------------|--|
| NO_ADDRESS | The requested name is valid, but does not have an address in the Internet domain. |
| NO_RECOVERY | This is a nonrecoverable error. |
| TRY_AGAIN | This is usually a temporary error and means that the local server did not receive a response from an authoritative server. A retry at some later time might succeed. |

Implementation Notes

The external variable `h_errno` is not set by UNIX implementations of this function.

See Also

`getnetbyname()`

getnetbyname()

get network information by name

Synopsis

```
#include <netdb.h>
struct netent *getnetbyname ( name )
char *name;
```

Description

The `getnetbyname()` function is used to obtain the network number (or list of network numbers) of a network when the official name of the network is known. This information is acquired from a network database maintained by the local system.

`getnetbyname()` returns a pointer to an object with this structure containing information extracted from an internal network database. This database is similar in function to the `/etc/networks` file on UNIX-based systems:

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

Parameters:

<code>n_name</code>	The official name of the network.
<code>n_aliases</code>	A zero-terminated list of alternate names for the network.
<code>n_addrtype</code>	The type of network number returned; currently always <code>AF_INET</code> .
<code>n_net</code>	The network number in machine byte order.

All information is contained in a static area and must be copied if it is to be saved. The `getnetbyname()` function sequentially searches from the beginning of the network database until a matching network name is found, or until the end of the database is reached. Only the Internet address format is currently understood.

Return Value

The `getnetbyname()` function returns a pointer to the `netent` structure, if successful. If unsuccessful, a null pointer is returned.

Error Codes

The `getnetbyname()` function may return these error codes in the external integer `h_errno`:

`NET_NOT_FOUND` No such network is known.

`NO_ADDRESS` The requested name is valid, but does not have an address in the Internet domain.

`NO_RECOVERY` This is a nonrecoverable error.

`TRY_AGAIN` This is usually a temporary error and means that the local server did not receive a response from an authoritative server. A retry at some later time may succeed.

Implementation Notes

The external variable `h_errno` is not set by UNIX implementations of this function.

See Also

`getnetbyaddr()`

getopts

parse command options

Synopsis

```
getopts optstring name [ argument... ]  
/usr/lib/getoptcvt [ -b ] filename  
/usr/lib/getoptcvt
```

Description

The `getopts` command is used to parse positional parameters and to check for valid options.

`optstring` must contain the option letters the command using `getopts` will recognize; if a letter is followed by a colon, the option is expected to have an argument, or group of arguments, which must be separated from it by white space.

getpeername()

get name of connected peer

Synopsis

```
#include <socket.h>
#include <uio.h>
int getpeername ( s, name, namelen )
int s;
struct sockaddr *name;
int *namelen;
```

Description

The `getpeername()` function is used to get the name of a connected peer. The name was assigned when a `connect()` or `accept()` function was successfully completed.

`getpeername()` returns the name of the peer connected to socket `s`. The `namelen` parameter should be initialized to indicate the amount of space pointed to by name. On return it contains the actual size of the name returned (in bytes). The name is truncated if the buffer provided is too small.

Return Value

If `getpeername()` is successful, a value of 0 is returned. A return value of -1 indicates an error, and the error code stored in the global integer `errno` indicates the nature of the error.

Error Codes

The `getpeername()` function returns these error codes (from `<errno.h>`) in the global integer `errno`:

EBADF	The argument <code>s</code> is not a valid descriptor.
EFAULT	Either the name pointer or name length pointer points to inaccessible memory.
EINVAL	The name length passed by the user is less than or equal to zero (0).
ENOTCONN	The socket is not connected.

See Also

`accept()`, `bind()`, `socket()`, `getsockname()`

getprotobyname()

get protocol information by name

Synopsis

```
#include <netdb.h>
struct protoent *getprotobyname ( name )
char *name;
```

Description

The `getprotobyname()` function is used to obtain the protocol number when the official protocol name is known. This information is acquired from a protocol database maintained by the local system.

`getprotobyname()` returns a pointer to an object with this structure containing information extracted from an internal network protocol database. This database is similar in function to the `/etc/protocols` file on UNIX-based systems:

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

<code>p_name</code>	The official name of the protocol
<code>p_aliases</code>	A zero-terminated list of alternate names for the protocol
<code>p_proto</code>	The protocol number

All information is contained in a static area and must be copied if it is to be saved. The `getprotobyname()` function sequentially searches from the beginning of the network protocol database until a matching protocol name is found, or until the end of the database is reached.

Return Value

The `getprotobyname()` function returns a pointer to the `protoent` structure, if successful. If unsuccessful, a null pointer is returned.

Error Codes

The `getprotobyname()` function may return these error codes in the external integer `h_errno`:

<code>NO_ADDRESS</code>	The requested name is valid, but does not have an address in the Internet domain.
<code>NO_RECOVERY</code>	This is a nonrecoverable error.
<code>PROTO_NOT_FOUND</code>	No such protocol is known.
<code>TRY_AGAIN</code>	This is usually a temporary error and means that the local server did not receive a response from an authoritative server. A retry at some later time might succeed.

Implementation Notes

The external variable `h_errno` is not set by UNIX implementations of this function.

See Also

`getprotobynumber()`

getprotobynumber()

get protocol information by number

Synopsis

```
#include <netdb.h>
struct protoent *getprotobynumber ( proto )
int proto;
```

Description

The `getprotobynumber()` function is used to obtain the official name of a protocol when the protocol number is known. This information is acquired from a network database maintained by the local system.

`getprotobynumber()` returns a pointer to an object with this structure containing information extracted from an internal network protocol database. This database is similar in function to the `/etc/protocols` file on UNIX-based systems:

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

Parameters

<code>p_name</code>	The official name of the protocol
<code>p_aliases</code>	A zero-terminated list of alternate names for the protocol
<code>p_proto</code>	The protocol number

All information is contained in a static area and must be copied if it is to be saved. The `getprotobynumber()` function sequentially searches from the beginning of the network protocol database until a matching protocol number is found, or until the end of the database is reached.

Return Value

If successful, the `getprotobynumber()` function returns a pointer to the `protoent` structure. If unsuccessful, a null pointer is returned.

Error Codes

The `getprotobynumber()` function may return these error codes in the external integer `h_errno`:

<code>NO_ADDRESS</code>	The requested name is valid, but does not have an address in the Internet domain.
<code>NO_RECOVERY</code>	This is a nonrecoverable error.
<code>PROTO_NOT_FOUND</code>	No such protocol is known.
<code>TRY_AGAIN</code>	This is usually a temporary error and means that the local server did not receive a response from an authoritative server. A retry at some later time might succeed.

Implementation Notes

The external variable `h_errno` is not set by UNIX implementations of this function.

See Also

`getprotobyname()`

getservbyname()

get service information by name

Synopsis

```
#include <netdb.h>
struct servent *getservbyname ( name, proto )
char *name, *proto;
```

Description

The `getservbyname()` function is used to obtain the port number associated with a service when its official name is known. This information is acquired from a service database maintained by the local system.

`getservbyname()` returns a pointer to an object with this structure containing information extracted from an internal network service database. This database is similar in function to the `/etc/services` file on UNIX-based systems:

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

<code>s_name</code>	The official name of the service.
<code>s_aliases</code>	A zero-terminated list of alternate names for the service.
<code>s_port</code>	The port number at which the service resides.
<code>s_proto</code>	The name of the protocol to use when contacting the service.

All information is contained in a static area and must be copied if it is to be saved. The `getservbyname()` function sequentially searches from the beginning of the network service database until a matching service name is found, or until the end of the database is reached. If a protocol name is also supplied (not null), searches must also match the protocol.

Return Value

The `getservbyname()` function returns a pointer to the `servent` structure, if successful. If unsuccessful, a null pointer is returned.

Error Codes

The `getservbyname()` function may return these error codes in the external integer `h_errno`:

- `NO_ADDRESS` The requested name is valid, but does not have an address in the Internet domain.
- `NO_RECOVERY` This is a nonrecoverable error.
- `SERV_NOT_FOUND` No such service is known.
- `TRY_AGAIN` This is usually a temporary error and means that the local server did not receive a response from an authoritative server. A retry at some later time might succeed.

Implementation Notes

The external variable `h_errno` is not set by UNIX implementations of this function.

See Also

`getservbyport()`, `getprotobyname()`

getservbyport()

get service information by port

Synopsis

```
#include <netdb.h>
struct servent *getservbyport ( port, proto )
int port;
char *proto;
```

Description

The getservbyport() function is used to obtain the official name of a service when the port number associated with the service is known. This information is acquired from a service database maintained by the local system.

getservbyport() returns a pointer to an object with this structure containing information extracted from an internal network service database. This database is similar in function to the /etc/services file on UNIX-based systems:

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

Parameters

s_name	The official name of the service.
s_aliases	A zero-terminated list of alternate names for the service.
s_port	The port number at which the service resides.
s_proto	The name of the protocol to use when contacting the service.

All information is contained in a static area and must be copied if it is to be saved. The getservbyport() function sequentially searches from the beginning of the network service database until a matching port number is found, or until the end of the database is reached. If a protocol name is also supplied (not null), searches must also match the protocol.

Return Value

The getservbyport() function returns a pointer to the servent structure, if successful. If unsuccessful, a null pointer is returned.

Error Codes

The getservbyport() function can return these error codes in the external integer h_errno:

NO_ADDRESS	The requested name is valid, but does not have an address in the Internet domain.
------------	---

- NO_RECOVERY This is a nonrecoverable error.
- SERV_NOT_FOUND No such service is known.
- TRY_AGAIN This is usually a temporary error and means that the local server did not receive a response from an authoritative server. A retry at some later time may succeed.

Implementation Notes

The external variable `h_errno` is not set by UNIX implementations of this function.

See Also

`getservbyname()`, `getprotobyname()`

getsockname()

get socket name

Synopsis

```
#include <socket.h>
#include <uio.h>
int getsockname ( s, name, namelen )
int s;
struct sockaddr *name;
int *namelen;
```

Description

The `getsockname()` function obtains the name assigned to a socket, which is the address of the local endpoint and was assigned with a `bind()` function.

`getsockname()` returns the current name for the specified socket. The `namelen` parameter should be initialized to indicate the amount of space pointed to by `name`. On return, it contains the actual size of the name returned (in bytes).

Return Value

If `getsockname()` is successful, a value of 0 is returned. A return value of -1 indicates an error, and the error code stored in the global integer `errno` indicates the nature of the error.

Error Codes

The `getsockname()` function returns these error codes (from `<errno.h>`) in the global integer `errno`:

- | | |
|--------|---|
| EBADF | The argument <code>s</code> is not a valid descriptor. |
| EFAULT | Either the name pointer or name length pointer points to inaccessible memory. |
| EINVAL | The name length passed by the user is less than or equal to zero (0). |

See Also

`bind()`, `socket()`

getsockopt()

get options on a socket

Synopsis

```
#include <socket.h>
#include <uio.h>
int getsockopt ( s, level, optname, optval, optlen )
int s, level, optname;
char *optval;
int *optlen;
```

Description

The `getsockopt()` function is used to retrieve options currently associated with a socket. Options always exist at the socket level and can also exist at layers within the underlying protocols. Options are set with the `setsockopt()` function. Options can exist at multiple protocol levels; they are always present at the uppermost socket level.

When retrieving socket options, the level at which the option resides and the name of the option must be specified. To retrieve options at the socket level, level is specified as `SOL_SOCKET`. To retrieve options at any other level, the protocol number of the appropriate protocol controlling the option is supplied. For example, to indicate that an option is to be returned by the TCP protocol, level should be set to the protocol number of TCP. Read `getprotobyname()` for additional information.

The parameters `optval` and `optlen` identify a buffer in which the value for the requested option is to be returned. The `optlen` parameter is a value-result parameter, initially containing the size of the buffer pointed to by `optval` and modified on return to indicate the actual size of the value returned. If no option value is to be returned, `optval` can be supplied as 0.

The `optname` parameter is passed uninterpreted to the appropriate protocol module for interpretation. The include file `<socket.h>` contains definitions for socket level options, described in the following table. Options at other protocol levels vary in format and name; consult the appropriate appendix for additional information.

Most socket-level options return an `int` parameter for `optval`. For boolean options, a non-zero value indicates the option is enabled, and a zero value indicates the option is disabled. `SO_LINGER` uses a struct `linger` parameter, defined in `<socket.h>`, which specifies the desired state of the option and the linger interval.

These options are recognized at the socket level. Except as noted, each can be examined with `getsockopt()` and set with `setsockopt()`:

Table 3-4 Socket-level Options

Socket Option	Description
<code>SO_BROADCAST</code>	Requests permission to send broadcast datagrams on the socket.
<code>SO_DEBUG</code>	Enables debugging in the socket modules. This option is implemented slightly differently than UNIX, in that, on UNIX it allows for debugging at the underlying protocol modules.
<code>SO_DONTROUTE</code>	Indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.
<code>SO_ERROR</code>	Returns any pending error on the socket and clears the error status. This option can be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors.

Table 3-4 Socket-level Options (Continued)

Socket Option	Description
SO_KEEPAIVE	<p>The keepalive option enables the periodic probing of the remote connection. This option specifies the type of keepalive to use. It also turns off the keepalive option. Should the connected peer fail to respond to the probe, the connection is considered broken and the process using the socket is notified via errno ETIMEDOUT. These int values are supported:</p> <ul style="list-style-type: none">0 - Turn off keepalive1 - Use keepalive with no data, and do not abort the session if no response2 - Use keepalive with no data, and abort the session if no response3 - Use keepalive with data. <p>See the notes following this section for a discussion of keepalive. Their use is discouraged and is provided only for those applications that are incapable of detecting idle sessions.</p>
SO_LINGER	<p>Controls the action taken when unsent messages are queued on socket and a close() is performed. If the socket promises reliable delivery of data and SO_LINGER is set to a value other than 0, the system blocks the process on the close() attempt until it is able to transmit the data or until the number of seconds specified by the SO_LINGER option expire or until the connection has timed out. If SO_LINGER is set and the interval is 0, the socket is closed once the system has scheduled that an orderly release be performed.</p>
SO_MAXRCVBYTCNT	<p>Returns the maximum allowable setting for the number of bytes of buffering allowed for the receive simplex of the socket. The SO_RCVBYTCNT option cannot be set to a value larger than that returned by this option.</p>
SO_MAXRCVREQCNT	<p>Returns the maximum allowable setting for the number of outstanding receive requests. The SO_RCVREQCNT option cannot be set to a value larger than that returned by this option.</p>
SO_MAXSNDBYTCNT	<p>Returns the maximum allowable setting for the number of bytes of buffering allowed for the transmit simplex of the socket. The SO_SNDBYTCNT option cannot be set to a value larger than that returned by this option.</p>
SO_MAXSNDRREQCNT	<p>Returns the maximum allowable setting for the number of outstanding transmit requests. The SO_SNDRREQCNT option cannot be set to a value larger than that returned by this option.</p>
SOO_OBINLINE	<p>This option is not currently supported by this release of the socket library. It is implemented on UNIX, and its purpose is to request that out-of-band data be placed in the normal data input queue as received by protocols that support out-of-band data; it is then be accessible with recv() or read() functions without the MSG_OOB flag.</p>
SO_OPTIONS	<p>Returns the current setting of all socket level options as contained in the socket's control block. This option is specific to this implementation and is not portable to other socket libraries. This option was designed for debugging and is not an option to be used by the average user. The socket options are defined in the header file sockvar.h.</p>
SO_RCVBUF	<p>Adjusts the normal buffer size allocated for input. The buffer size can be increased for high-volume connections to improve throughput or can be decreased to limit the possible backlog of incoming data. The system places an absolute limit on this value. This implementation of sockets provides for this option for backward compatibility, but also allows for buffer options that are more specific to the underlying API and therefore provide a better method of controlling a socket's buffering characteristics. These options are SO_RCVBYTCNT and SO_RCVREQCNT. All buffering options can be set only once by the socket user and must be done before any data is sent or received on the socket.</p>

Table 3-4 Socket-level Options (Continued)

Socket Option	Description
SO_RCVBYTCNT	Adjusts the number of bytes allocated to the receive circular buffer for a socket. This option is specific to this implementation of sockets and therefore is not portable to other socket libraries. This option can be set only once successfully, and if the receive byte count of the circular buffer is changed, it must be done prior to sending or receiving any data on the socket.
SO_RCVREQCNT	Adjusts the number of receive requests that can be active within the socket library at a given time. This option is specific to this implementation of sockets and therefore is not portable to other socket libraries. This option can be set only once successfully, and if the receive request count is changed, it must be done prior to sending or receiving any data on the socket.
SO_READFRAG	This option lets a user of a datagram type socket that preserves message boundaries read a datagram a piece at a time. Traditionally with UNIX sockets, if a user issues a read request for 100 bytes and the datagram being read consists of 120 bytes, the socket returns the first 100 bytes to the caller and then flushes the remaining 20 bytes. This default method of operation by this implementation of sockets can be overridden with this option. This option does not allow parsing of pieces of a single datagram into a single user buffer. When this option is used, the user must determine the boundaries of datagrams. This option is specific to this implementation and is not portable to other socket implementations.
SO_REUSEADDR	This option indicates that the rules used in validating addresses supplied in a bind() function should allow reuse of local addresses.
SO_SENDALL	This option guarantees that any form of send request (send(), sendto(), sendmsg(), write(), or writev()) that is done in a blocking mode transmits all the data specified by the user. Traditionally BSD sockets would send as many bytes as the current buffering allocation allowed and then return to the user with a count of the actual number of bytes transmitted. If the user requested that a write of 200 bytes be done, but there currently was only buffering space for 150 bytes, the socket would queue 150 bytes for transmission and return a count of 150 to the caller to indicate that 50 bytes could not be transmitted due to lack of buffer space. This implementation of sockets acts identically to a UNIX socket under the same scenario under the default setting of the socket options. However, if this option is turned ON in Cisco IOS for S/390 sockets, the socket blocks the user until all of the data has been queued for transmission or some type of error occurs. This option is specific to this implementation.
SO_SNDBUF	Adjusts the normal buffer size allocated for output. The buffer size can be increased for high-volume connections to improve throughput or can be decreased to limit the possible backlog of outgoing data. The system places an absolute limit on this value. This implementation of sockets provides this option for backward compatibility. It also allows for buffer options that are more specific to the underlying API and therefore provides a better method of controlling a socket's buffering characteristics. These options are SO_SNDBYTCNT and SO_SNDREQCNT. All buffering options can be set only once by the socket user and must be done before any data is sent or received on the socket.
SO_SNDBYTCNT	Adjusts the number of bytes allocated to the send circular buffer for a socket. This option is specific to this implementation of sockets and is not portable to other socket libraries. This option can be set only once successfully, and if the send byte count of the circular buffer is changed, it must be done prior to sending or receiving any data on the socket.

Table 3-4 Socket-level Options (Continued)

Socket Option	Description
SO_SNDREQCNT	Adjusts the number of send requests that can be active within the socket library at a given time. This option is specific to this implementation of sockets and is not portable to other socket libraries. This option can be set only once successfully, and if the send request count is changed, it must be done prior to sending or receiving any data on the socket.
SO_STATE	Returns the current socket state as contained in the socket's control block. This option is specific to this implementation and is not portable to other socket libraries. This option was designed for debugging and is not an option the average user should use. The socket states are defined in the header file <code>sockvar.h</code> .
SO_SUBSTATE	Returns the current setting of the socket's substate as contained in the socket's control block. This option is specific to this implementation and is not portable to other socket libraries. This option is for debugging and is not for use by the average user. The socket substates are defined in the header file <code>sockvar.h</code> .
SO_TYPE	Returns the type of the socket, such as <code>SOCK_STREAM</code> ; it is useful for servers that inherit sockets on startup.
SO_USELOOPBACK	Requests that the loopback interface is used rather than a real physical interface. These options are recognized at the TCP level (<code>IPPROTO_TCP</code>):
TCP_MAXSEG	This option is not supported by this implementation of sockets at the present release. On UNIX this option lets the user of a <code>SOCK_STREAM</code> socket declare the value of the maximum segment size for TCP to use when negotiating this value with its remote endpoint.
TCP_NODELAY	Ensures that TCP type sockets (<code>SOCK_STREAM</code>) send data as soon as possible and do not wait for more data or a given amount of time to enhance the packetizing algorithm. This option is similar to the BSD UNIX socket option.

Return Value

If `getsockopt()` is successful, a value of 0 is returned. A return value of -1 indicates an error, and the error code stored in the global integer `errno` indicates the nature of the error.

Error Codes

The `getsockopt()` function returns these error codes (from `<errno.h>`) in the global integer `errno`:

EBADF	The <code>s</code> argument is not a valid descriptor.
EFAULT	The pointer to the value buffer points to inaccessible memory.
EFAULT	The pointer to the value buffer length points to inaccessible memory.
EINVAL	The size of the value buffer does not equal the size of the option. Most options require an integer length buffer or, in the case of <code>SO_LINGER</code> , the buffer must be the size of the linger structure.
EINVAL	The option buffer size is greater than the maximum allowed by the API.
EINVAL	The option is not supported at the level requested.
EINVAL	No options can be read from the protocol layers.

Implementation Notes

These options are recognized at the socket level on BSD UNIX systems, but are not supported by the API. If any of these options are referenced, an error is generated:

- SO_SNDLOWAT Sets the send low water buffering mark.
- SO_RCVLOWAT Sets the receive low water buffering mark.
- SO_SNDTIMEO Sets the send timeout value. This option is not currently implemented in UNIX.
- SO_RCVTIMEO Sets the receive timeout value. This option is not currently implemented in UNIX.

See Also

ioctl(), setsockopt(), socket(), fcntl()

getstabsize()

get socket table size

Synopsis

```
#include <socket.h>
#include <uio.h>
int getstabsize()
```

Description

The `getstabsize()` function provides a method by which the socket library user can determine the maximum number of sockets that can be used at any given time.

The `getstabsize()` function performs the same function as the `gettablesize` function call on UNIX. It varies somewhat in that it only returns the maximum number of sockets that can be used at a given time and not the combined total of socket and file descriptors.

Implementation Notes

See the function description for the differences.

gettimeofday()

get the date and time

Synopsis

```
#include <sys/time.h>
int gettimeofday ( tp )
struct time_t *tp;
```

Description

The `gettimeofday()` function gets the system's notion of the current time. The current time is expressed in elapsed seconds and microseconds since a particular time. The time chosen depends on `#define USESASCTIME`. If this is defined, `time_t` will be returned as a double and is relative to 00:00 January 1, 1900. If the above define is not used, `time_t` is returned as an int relative to 00:00 January 1, 1970.

Implementation Notes

Consult the SAS/C documentation for more information on the `time_t` structure.

htonl()

convert long values from host to network byte order

Synopsis

```
#include <inet.h>
unsigned long htonl ( hostlong );
unsigned long hostlong;
```

Description

The `htonl()` function is provided to maintain compatibility with application programs ported from other systems that byte-swap memory. These systems store 32-bit quantities in right-to-left byte order instead of the usual left-to-right byte order assumed by network protocols.

`htonl()` converts 32-bit quantities from host byte order to network byte order. This function was originally provided for UNIX systems running on VAX processors that byte-swap memory. On machines such as IBM mainframes where network and host byte order are the same, this function is defined as a null macro in the include file `<inet.h>`.

This function is most often used in conjunction with Internet addresses and port numbers as returned by `gethostbyname()` and `getservbyname()`.

Implementation Notes

The `htonl()` function is implemented as a null macro and performs no operation on the function argument.

See Also

`gethostbyname()`, `getservbyname()`

htons()

convert short values from host to network byte order

Synopsis

```
#include <inet.h>
unsigned short htons ( hostshort );
unsigned short hostshort;
```

Description

The `htons()` function is provided to maintain compatibility with application programs ported from other systems that byte-swap memory. These systems store 16-bit quantities in right-to-left byte order instead of the usual left-to-right byte order assumed by network protocols.

`htons()` converts 16-bit quantities from host byte order to network byte order. This function was originally provided for UNIX systems running on VAX processors that byte-swap memory. On machines such as IBM mainframes, where network and host byte order are the same, this function is defined as a null macro in the include file `<inet.h>`.

This function is most often used in conjunction with Internet addresses and port numbers as returned by `gethostbyname()` and `getservbyname()`.

Implementation Notes

The `htons()` function is implemented as a null macro and performs no operation on the function argument.

See Also

`gethostbyname()`, `getservbyname()`

inet()

internet address manipulation routines

Synopsis

```
#include <socket.h>
#include <uio.h>
#include <inet.h>
unsigned long inet_addr ( cp )
char *cp;
unsigned long inet_network ( cp )
char *cp;
char *inet_ntoa ( in )
struct in_addr in;
unsigned long inet_makeaddr ( net, lna )
int net, lna;
int inet_lnaof ( in )
struct in_addr in;
int inet_netof ( in )
struct in_addr in;
```

Description

The `inet` functions are a set of routines that construct Internet addresses or break Internet addresses down into their component parts. Routines that convert between the binary and ASCII (“dot” notation) form of Internet addresses are included.

The functions `inet_addr()` and `inet_network()` each interpret character strings representing numbers expressed in the Internet standard dot (“.”) notation, returning numbers suitable for use as Internet addresses and network numbers, respectively. The function `inet_ntoa()` takes an Internet address and returns an ASCII string representing the address in dot notation. The function `inet_makeaddr()` takes an Internet network number and a local network address (host number) and constructs an Internet address from it. The functions `inet_netof()` and `inet_lnaof()` break apart Internet host addresses, returning the network number and local network address part, respectively.

All Internet addresses are returned in network byte order. All network numbers and local address parts are returned as integer values in host byte order. On machines such as IBM mainframes, network and host byte order are the same.

Values specified using the Internet standard dot notation take one of these forms:

a.b.c.d

a.b.c

a.b

- When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address.
- When a three-part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right-most two bytes of the network address. This makes the three-part address format convenient for specifying Class B network addresses as 128.net.host.
- When a two-part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right-most three bytes of the network address. This makes the two-part address format convenient for specifying Class A network addresses as net.host.
- When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as parts in a dot notation can be decimal, octal, or hexadecimal, as specified in the C language (in other words, a leading 0x or 0X implies hexadecimal; otherwise, a leading 0 implies octal; if there is no leading 0, the number is interpreted as decimal).

Return Value

The `inet_addr()` function returns an Internet address if successful, or a value of -1 if the request was unsuccessful.

The `inet_network()` function returns a network number if successful, or a value of -1 if the request was malformed.

The `inet_ntoa()` function returns a pointer to an ASCII string giving an Internet address in dot notation, `inet_makeaddr()` returns an Internet address, and `inet_netof()` and `inet_lnaof()` each return an integer value.

Implementation Notes

The function `inet_makeaddr()` returns a structure of type `in_addr` with the UNIX function. In this implementation, it returns a value of type unsigned long.

See Also

`gethostbyname()`, `getnetbyname()`

inet_aton()

convert ASCII string to network address

Synopsis

```
#include <types.h>
#include <socket.h>
#include <in.h>
#include <inet.h>
int inet_aton( cp , in)
const char    *cp;
struct in_addr    *in;
```

Description

The `inet_aton()` function is provided to convert an ASCII representation of an Internet Address to its Network Internet Address.

`inet_aton()` interprets a character string representing numbers expressed in the Internet standard ‘.’ notation, returning a number suitable for use as an Internet address.

Implementation Notes

Values specified using the ‘.’ (dot) notation take one of the following forms:

a.b.c.d

a.b.c

a.b

a

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address. When a three part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right-most two bytes of the network address. This makes the three part address format convenient for specifying Class B network addresses as “128.net.host”. When a two part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right-most three bytes of the network address. This makes the two part address format convenient for specifying Class A network addresses as “net.host”. When only one part is given, the value is stored directly in the network address without any byte rearrangement. All numbers supplied as “parts” in a ‘.’ notation may be decimal, octal, or hexadecimal, as specified in the C language (that is, a leading 0x or 0X implies hexadecimal; a leading 0 implies octal; otherwise, the number is interpreted as decimal).

See Also

`inet_ntoa()`

inet_ntoa()

convert network internet address to ASCII representation

Synopsis

```
#include <types.h>
#include <socket.h>
#include <in.h>
#include <inet.h>
char *inet_ntoa ( in )
const struct in_addr in;
```

Description

The `inet_ntoa()` function is provided to convert an Internet network address to its ASCII “d.d.d.d” representation.

`inet_ntoa()` returns a pointer to a string in the base 256 notation "d.d.d.d". Internet addresses are returned in network order (bytes ordered from left to right).

Implementation Notes

`inet_ntoa()` points to a buffer which is overwritten on each call.

See Also

`inet_aton()`

ioctl()

control I/O

Synopsis

```
#include <socket.h>
#include <uio.h>
int ioctl ( s, request, argp )
int s;
unsigned long request;
char *argp;
```

Description

The `ioctl()` function manipulates I/O controls associated with a socket. In particular, the sensing of out-of-band data marks and the enabling of non-blocking I/O can be controlled.

The `ioctl()` function performs a variety of control functions on a socket, indicated by `s`. An `ioctl` request has encoded in it whether the argument pointed to by `argp` is an input parameter supplied by the caller or an output parameter returned by the `ioctl()` function. `request` also encodes in bytes the length of the argument.

Permissible values for `request` are defined in `<socket.h>`. The current implementation of `ioctl()` supports these requests:

Table 3-5 **ioctl Requests**

Request	Definition
FIOASYNC	Used to enable the usage of the user-added signals of the socket library. Asynchronous events happening at the local endpoint initiate the triggering of a signal. SIGIO, SIGURG and SIGPIPE are all supported by the implementation.
FIONBIO	Used to set and clear non-blocking I/O mode. When a socket is in non-blocking I/O mode, any function that would otherwise block for flow control or synchronization completes immediately with an error. A return value of -1 indicates the error, and the error code EWOULDBLOCK stored in the external integer <code>errno</code> indicates the blocking condition. The function should be reissued later.
FIONREAD	Returns the status of the received data flag. Unlike UNIX, which actually returns the number of data bytes in the receive buffer, a return value of 1 indicates that there is data to be read. A value of 0 indicates an absence of data to be read.
SIOCADDRT	Add a single routing table entry Note: not allowed for user application programs.
SIOCATMARK	Indicates whether or not the current read pointer points to the location at which out-of-band data was received. Whenever out-of-band data is received, the location in the input stream is marked. The boolean value returned by this request indicates when the read pointer has reached the mark.
SIOCDELRT	Delete a single routing table entry. Note: not allowed for user application programs.
SIOCGIFADDR	Get interface address. The argument for this type of request is defined by <code>ifreq</code> , which is found in <code>if.h</code> .
SIOCGIFBRDADDR	Get broadcast address. The argument for this type of request is defined by <code>ifreq</code> , which is found in <code>if.h</code> .

Table 3-5 ioctl Requests (Continued)

Request	Definition
SIOCGIFCONF	Get interface configuration list. This request takes an ifconf structure as a value-result parameter. The ifc_len field should be initially set to the size of the buffer pointed to by ifc_buf. On return, it will contain the length, in bytes of the configuration list. The argument for this type of request is defined by ifreq, which is found in if.h.
SIOCGIFDSTADDR	Get point-to-point address for the interface. The argument for this type of request is defined by ifreq, which is found in if.h.
SIOCGIFFLAGS	Get the interface flags. The argument for this type of request is defined by ifreq, which is found in if.h.
SIOCGIFHWADDR	Get the hardware address. The argument for this type of request is defined by ifreq, which is found in if.h.
SIOCGIFMETRIC	Get the metric associated with the interface. The argument for this type of request is defined by ifreq, which is found in if.h.
SIOCGIFMTU	Get the maximum transmission unit size for interface. The argument for this type of request is defined by ifreq, which is found in if.h.
SIOCGIFNETMASK	Get the network address mask. The argument for this type of request is defined by ifreq, which is found in if.h.
SIOCGIFNUM	Get the number of interfaces. This request returns an integer which is the number of interface descriptions (struct ifreq, found in if.h) that will be returned by the SIOCGIFCONF ioctl; that is, it gives an indication of how large ifc_len has to be.
SIOCSIFMETRIC	Sets the network interface routing metric. The argument for this type of request is defined by ifreq, which is found in if.h.

Return Value

If ioctl() is successful, a value of 0 is returned. A return value of -1 indicates an error, and the error code stored in the global integer errno indicates the nature of the error.

Error Codes

The ioctl() function returns these error codes (from <errno.h>) in the global integer errno:

- EBADF The s argument is not a valid descriptor.
- EFAULT The pointer argp is invalid.
- EINVAL Request or argp is invalid or not supported by this implementation of ioctl().

Implementation Notes

The ioctl() function as implemented on UNIX-based systems is used to manipulate controls that tend to be specific to a given implementation of a protocol and often do not translate directly to other environments. As such, most of the request values defined for UNIX have no meaning in the MVS environment or are not readily portable.

Note the difference of FIONREAD as previously described.

See Also

accept(), connect(), recv(), recvfrom(), recvmsg(), send(), sendmsg(), sendto(), write()

listen()

listen for connections on a socket

Synopsis

```
#include <socket.h>
#include <uio.h>
int listen ( s, backlog )
int s, backlog;
```

Description

The `listen()` function indicates the application program is ready to accept connection requests arriving at a socket of type `SOCK_STREAM`. The connection request is queued (if possible) until accepted with an `accept()` function.

To accept connections, a socket is first created with `socket()`, a readiness to accept incoming connections, and a queue limit for incoming connections, which are specified with `listen()`, and then the connections are accepted with `accept()`. The `listen()` function applies only to sockets of type `SOCK_STREAM` or `SOCK_ASSOC`.

The `backlog` parameter defines the maximum number of pending connections that may be queued. If a connection request arrives with the queue full, the client can receive an error with an indication of `ECONNREFUSED`, or, if the underlying protocol supports retransmission, the request can be ignored so that retries can succeed.

Return Value

If `listen()` is successful, a value of 0 is returned. A return value of -1 indicates an error, and the error code stored in the global integer `errno` indicates the nature of the error.

Error Codes

The `listen()` function returns these error codes (from `<errno.h>`) in the global integer `errno`:

<code>EADDRINUSE</code>	The specified address is not available. Another socket is currently using this endpoint to listen.
<code>EALREADY</code>	The socket is currently listening.
<code>EBADF</code>	The argument <code>s</code> is not a valid descriptor.
<code>EINVAL</code>	The <code>backlog</code> variable passed by the user is less than or equal to zero (0).
<code>EINVAL</code>	The socket is not in the bound state.
<code>EOPNOTSUPP</code>	The socket is not of a type that supports the operation <code>listen()</code> .

Implementation Notes

The backlog can be limited to a value smaller than the current maximum of five supported by most BSD UNIX implementations. If the underlying protocol cannot support the value specified, a smaller value is substituted. backlog only limits the number of connection requests that can be queued simultaneously and not the total number of connections that can be accepted. A listen count of less than or equal to zero is invalid (EINVAL).

This implementation lets the user program listen on SOCK_ASSOC sockets and accept the incoming connection requests.

See Also

accept(), connect(), socket()

mvselect()

synchronous I/O multiplexing with optional ECB list

Synopsis

```
#include <socket.h>
#include <uio.h>

int mvselect ( nfd, readfds, writefds, exceptfds, timeout, ecblst, ecblstp )

int nfd;
fd_set *readfds, *writefds, *exceptfds;
struct timeval *timeout;
int ecblst;
unsigned long **ecblstp;
FD_SET ( fd, &fdset )
FD_CLR ( fd, &fdset )
FD_ISSET ( fd, &fdset )
FD_ZERO ( &fdset )
int fd;
fd_set fdset;
```

Description

The mvselect() function is used to synchronize processing of several sockets operating in non-blocking mode and other system events related to an optional ECB list. Sockets that are ready for reading or writing, or that have a pending exceptional condition can be selected, as well as the posting of any ECBs in the ECB list. If no sockets are ready for processing or no ECBs are posted, the mvselect() function can block indefinitely or wait for a specified period of time (which may be zero) and then return.

mvselect() examines the I/O descriptor sets whose addresses are passed in readfds, writefds, and exceptfds to see if some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. The first nfd descriptors are checked in each set (the descriptors from 0 through nfd-1 in the descriptor sets are examined). Also, the ECB list is checked to see which have been posted. Only ecblst number of ECBs are checked. On return, mvselect() replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. The total number of ready descriptors in all the sets and ECBs posted are returned.

Sockets use MVS STIMER services. Socket applications may use up to fifteen STIMER calls per task control block (TCB). STIMER CANCEL ID=ALL must not be used by socket applications.

Note If you invoke mvselect() before any sockets are opened, an error code of EBADF will be returned because the internal user work area has not been initialized. A workaround for this would be to open a “throwaway” socket prior to calling mvselect().

The descriptor sets are stored as bit fields in arrays of integers. These macros are provided for manipulating such descriptor sets:

- | | |
|---------------------------------------|--|
| <code>FD_ZERO(&fdset)</code> | Initializes a descriptor set <code>fdset</code> to the null set. |
| <code>FD_SET(fd, &fdset)</code> | Includes a particular descriptor <code>fd</code> in <code>fdset</code> . |
| <code>FD_CLR(fd, &fdset)</code> | Removes <code>fd</code> from <code>fdset</code> . |
| <code>FD_ISSET(fd, &fdset)</code> | Is non-zero if <code>fd</code> is a member of <code>fdset</code> , zero otherwise. |

The behavior of these macros is undefined if a descriptor value is less than zero or greater than or equal to `FD_SETSIZE`, which normally is at least equal to the maximum number of descriptors supported by the system.

If `timeout` is a non-zero pointer, it specifies a maximum interval to wait for the selection to complete. If `timeout` is a zero pointer, `mvselect` blocks indefinitely. To affect a poll, the `timeout` argument should be non-zero, pointing to a zero-valued `timeval` structure. The `mvselect` function uses the OS timer. If this is undesirable, the `timeout` variable should be a zero pointer and one of the ECBs in the list should be posted by the application-specific `timeout` routine.

Any of `readfds`, `writfds`, and `exceptfds` can be given as zero pointers if no descriptors are of interest.

Note For release 3.1 and higher, note that the `select()` and `mvselect()` functions have changed slightly for SAS/C users. SAS/C allows signals to be raised. `select()` may complete if the signal is raised without the conditions on the `select()` parameters having been met. This new feature allows you to use signal functionality, issue a blocking receive, and do a send from a signal handler.

Return Value

If successful, `mvselect()` returns the sum of the number of ready descriptors that are contained in the descriptor sets and the posted ECBs, or 0 if the time limit expires. Otherwise, the value -1 is returned, and the error code stored in the global integer `errno` indicates the nature of the error. If `mvselect()` returns with an error, including one due to an interrupted call, the descriptor sets is unmodified.

Error Codes

The `mvselect()` function returns these error codes (from `<errno.h>`) in the global integer `errno`:

- | | |
|---------------------|--|
| <code>EBADF</code> | One of the descriptor sets specified an invalid descriptor. |
| <code>EINVAL</code> | The specified time limit is invalid. One of its components is negative or too large. |
| <code>EINVAL</code> | The count of socket descriptors to check is less than or equal to 0. |
| <code>EINVAL</code> | No valid socket descriptors were referenced by the three sets of masks passed by the caller. |
| <code>ENOMEM</code> | Could not allocate space to build ECB list. |

Implementation Notes

The implementation of `mvselect()` provided with the API supports only MVS ECBs and descriptors associated with sockets.

See Also

`accept()`, `connect()`, `read()`, `write()`, `recv()`, `send()`, `recvfrom()`, `sendto()`, `recvmsg()`, `sendmsg()`, `select()`

openold()

open a socket passed from another task

Synopsis

```
#include <socket.h>
#include <uio.h>
int openold(token)
unsigned long token;
```

Description

The openold() function is used to receive the ownership of a socket from another task within the same address space.

openold() creates a new socket descriptor using a token passed from another task within the same address space. Initially, the other task had created a socket using either socket() or accept(). The other task then called closepass() and passed the token returned by closepass() via application-dependent IPC to the receiving task or subtask.

The other task then calls close() and the receiving task calls openold(). When openold() completes, the receiving task now owns the socket and the other task can no longer reference it.

The sequence of events is shown below:

Main Task	Subtask
1. Call closepass(fd).	
2. Send IPC to subtask passing token returned by closepass	
3. Call close(fd). On return from close, main task can no longer reference this fd.	
	4. Receive token from main task via application-dependent IPC.
	5. Call openold(token).
	6. Subtask may now use fd returned by openold to access network.
	7. Subtask is through with socket.
	8. Call close(fd) to remove socket.

Return Value

On successful completion, openold() returns a new socket descriptor. Otherwise, the value -1 is returned, and the error code stored in the global integer errno indicates the nature of the error.

Error Codes

The `openold()` function returns these error codes (from `<errno.h>`) in the global integer `errno`:

<code>EACCESS</code>	Permission to create a socket of the specified type and/or protocol is denied.
<code>ECONFIG</code>	Socket configuration has an error, or a user session with the underlying API cannot be opened.
<code>EINVAL</code>	Token is invalid.
<code>EMFILE</code>	The system file table is full.
<code>EMFILE</code>	A new socket cannot be opened due to an API resource shortage or user endpoint allocation limit.
<code>ENOBUFS</code>	Insufficient buffer space is available. The socket cannot be created until sufficient resources are freed.
<code>EPROTONOSUPPORT</code>	The specified protocol is not supported within this domain.

Implementation Notes

The implementation of this function is provided to ease the development of server-oriented socket applications using the socket library.

See Also

`accept()`, `close()`, `closepass()`, `socket()`

ntohl()

convert long values from network to host byte order

Synopsis

```
#include <inet.h>
unsigned long ntohl ( netlong );
unsigned long netlong;
```

Description

The `ntohl()` function is provided to maintain compatibility with application programs ported from other systems that byte-swap memory. These systems store 32-bit quantities in right-to-left byte order instead of the usual left-to-right byte order assumed by network protocols.

`ntohl()` converts 32-bit quantities from network byte order to host byte order. This function was originally provided for UNIX systems running on VAX processors that byte-swap memory. On machines such as IBM mainframes where network and host byte order are the same, this function is defined as a null macro in the include file `<inet.h>`.

This function is most often used in conjunction with Internet addresses and port numbers as returned by `gethostbyname()` and `getservbyname()`.

Implementation Notes

The `ntohl()` function is implemented as a null macro and performs no operation on the function argument.

See Also

`gethostbyname()`, `getservbyname()`

ntohs()

convert short values from network to host byte order

Synopsis

```
#include <inet.h>
unsigned short ntohs ( netshort );
unsigned short netshort;
```

Description

The ntohs() function is provided to maintain compatibility with application programs ported from other systems that byte-swap memory. These systems store 16-bit quantities in right-to-left byte order instead of the usual left-to-right byte order assumed by network protocols.

ntohs() converts 16-bit quantities from network byte order to host byte order. This function was originally provided for UNIX systems running on VAX processors that byte-swap memory. On machines such as IBM mainframes where network and host byte order are the same, this function is defined as a null macro in the include file <inet.h>.

This function is most often used in conjunction with Internet addresses and port numbers as returned by gethostbyname() and getservbyname().

Implementation Notes

The ntohs() function is implemented as a null macro and performs no operation on the function argument.

See Also

gethostbyname(), getservbyname()

openlog()

initialize the log file

Synopsis

```
#include <syslog.h>
#include <socket.h>
#include <inet.h>
void openlog ( ident, logopt, facility )
char *ident;
int logopt;
int facility;
```

Description

The openlog() function is provided to initialize the log file.

If special processing is needed, the openlog() function initializes the log file. The parameter ident is a string that is prepended to every message. logopt is a bit field indicating logging options. Values for logopt are:

Table 3-6 logopt Values

logopt Values	Definition
LOG_PID	Log the process ID with each message. This is useful for identify in specific processes.
LOG_CONS	Write messages to the system console if they cannot be sent to syslog.
LOG_NDELAY	Open the connection to syslog() immediately. Normally the open is delayed until the first message is logged. This is useful for programs that need to manage the order in which file descriptors are allocated.
LOG_NOWAIT	Do not wait for processes that have been forked to log messages onto the console. This option should be used by processes that enable notification of child termination using SIGCHLD since syslog() may otherwise block waiting for a process whose exit status has already been collected. The facility parameter encodes a default facility to be assigned to all messages that do not have an explicit facility already encoded:
LOG_KERN	Messages generated by the kernel. These cannot be generated by any user processes.
LOG_USER	Messages generated by random user processes. This is the default facility identifier if none is specified.
LOG_MAIL	The mail system.
LOG_DAEMON	System daemons, such as ftpd(1M).
LOG_AUTH	The authorization system: login(1), su(1M), getty(1M), etc.
LOG_LPR	The line printer spooling system: lpr(1B), lpc(1B), etc.
LOG_NEWS	Reserved for the USENET network news system.
LOG_UUCP	Reserved for the UUCP system; it does not currently use syslog.
LOG_CRON	The cron/at facility; crontab(1), at(1), cron(1M), etc.
LOG_LOCAL0	Reserved for local use.
LOG_LOCAL1	Reserved for local use.
LOG_LOCAL2	Reserved for local use.

Table 3-6 **logopt Values (Continued)**

logopt Values	Definition
LOG_LOCAL3	Reserved for local use.
LOG_LOCAL4	Reserved for local use.
LOG_LOCAL5	Reserved for local use.
LOG_LOCAL6	Reserved for local use.
LOG_LOCAL7	Reserved for local use.

See Also

syslog(), closelog(), vsyslog()

read()

read input

Synopsis

```
#include <socket.h>
#include <uio.h>
int read ( s, buf, nbytes )
int s;
char *buf;
int nbytes;
```

Description

The read() function is used to input data from a socket. It operates in the same manner as reading data from a file. In a normal UNIX environment, where socket and file I/O are integrated, the read() function can be called with either a socket or file descriptor.

read() attempts to read nbytes of data from the socket referenced by the descriptor s into the buffer pointed to by buf. On successful completion, read() returns the number of bytes actually read and placed in the buffer.

Return Value

If successful, read() returns the number of bytes read. A value of 0 is returned if an end-of-file condition exists, indicating no more read() functions should be issued to this socket. Otherwise, the value -1 is returned, and the error code stored in the global integer errno indicates the nature of the error.

Error Codes

The read() function returns these error codes (from <errno.h>) in the global integer errno:

EBADF	An invalid descriptor was specified.
ECONNABORTED	The incoming connection request was aborted by the remote endpoint.
ECONNREFUSED	The remote endpoint refused to continue the connection.
ECONNRESET	The remote endpoint reset the connection request.
EDESTUNREACH	Remote destination is now unreachable.
EFAULT	The buffer passed by the user points to inaccessible memory.
EFAULT	The pointer to the storage where the socket library is to place the name of the remote endpoint passed by the user points to inaccessible memory.
EFAULT	The pointer to the length of the name buffer points to inaccessible storage.
EHOSTUNREACH	Remote host is now unreachable.
EINVAL	The number of bytes in the receive buffer is less than zero (0).

EINVAL	The number of bytes requested on the receive request is greater than the current maximum buffer space allocated or greater than the maximum receive request allowed to this type of endpoint.
EINVAL	A connectionless socket does not yet have a name bound to it.
EINVAL	A connectionless socket is being used and the length of the name passed by the user is zero (0).
ENETDOWN	Local network interface is down.
ENOBUFS	The system was unable to allocate an internal buffer. The operation might succeed when buffers become available.
ENOBUFS	The socket library cannot allocate the necessary buffer space within the API.
ENOTCONN	The socket being used requires a connection before data can be transferred, and no such connection currently exists.
ETIMEDOUT	The connection request by the remote endpoint timed out.
EWOULDBLOCK	The socket is marked non-blocking and the requested operation would block.

See Also

recv(), recvfrom(), recvmsg(), select(), socket(), write()

readv()

read vectored input

Synopsis

```
#include <socket.h>
#include <uio.h>
int readv ( s, iov, iovcnt )
int s;
struct iovec *iov;
int iovcnt;
```

Description

The readv() function is used to input data from a socket in scatter mode when the input is to be noncontiguous. It operates in the same manner as reading data from a file. In a normal UNIX environment, where socket and file I/O are integrated, the readv() function can be called with either a socket or file descriptor.

The readv() function performs the same action as read(), but scatters the input data into the iovcnt buffers specified by the members of the iov array:

```
iov[0], iov[1], ..., iov[iovcnt-1]
```

The iovec structure is defined in this way:

```
struct iovec
{
    caddr_t    iov_base;
    int        iov_len;
};
```

Each iovec entry specifies the base address and length of an area in memory where data should be placed. The readv() function always fills an area completely before proceeding to the next iov entry. On successful completion, readv() returns the total number of bytes read.

Return Value

If successful, readv() returns the total number of bytes read. A value of 0 is returned if an end-of-file condition exists, indicating no more readv() functions should be issued to this socket. Otherwise, the value -1 is returned, and the error code stored in the global integer errno indicates the nature of the error.

Error Codes

The readv() function returns these error codes (from <errno.h>) in the global integer errno:

- | | |
|--------------|---|
| EBADF | An invalid descriptor was specified. |
| ECONNABORTED | The incoming connection request was aborted by the remote endpoint. |
| ECONNREFUSED | The remote endpoint refused to continue the connection. |
| ECONNRESET | The remote endpoint reset the connection request. |
| EDESTUNREACH | The remote destination is now unreachable. |

EFAULT	The buffer passed by the user points to inaccessible memory.
EFAULT	The pointer to the storage where the socket library is to place the name of the remote endpoint passed by the user points to inaccessible memory.
EFAULT	The pointer to the length of the name buffer points to inaccessible storage.
EFAULT	The pointer to the iovec structure points to inaccessible memory.
EHOSTUNREACH	Remote host is now unreachable.
EINVAL	The number of bytes in the receive buffer is less than zero (0).
EINVAL	The number of bytes requested on the receive request is greater than the current maximum buffer space allocated or greater than the maximum receive request allowed to this type of endpoint.
EINVAL	A connectionless socket does not yet have a name bound to it.
EINVAL	A connectionless socket is being used and the length of the name passed by the user is zero (0).
EINVAL	The count of iovec elements in the array is greater than 16 or less than or equal to zero (0).
ENETDOWN	Local network interface is down.
ENOBUFS	The system was unable to allocate an internal buffer. The operation might succeed when buffers become available.
ENOBUFS	The socket library cannot allocate the necessary buffer space within the API.
ENOTCONN	The socket being used requires a connection before data can be transferred, and no such connection currently exists.
ETIMEDOUT	The connection request by the remote endpoint timed out.
EWOULDBLOCK	The socket is marked non-blocking and the requested operation would block.

Implementation Notes

The maximum number of vectored I/O structures (struct iovec) in an array per call is sixteen (MSG_IOVLEN).

See Also

recv(), recvfrom(), recvmsg(), select(), socket(), writev()

perror()

system error messages

Synopsis

```
#include <socket.h>
#include <uio.h>
perror ( s )
char *s;
```

Description

The perror() function produces a short error message on the standard error file describing the last error encountered during a call to the socket library for a C program.

perror() prints a system-specific error message generated as a result of a failed call to the socket library. The argument string s is printed, followed by a colon, the system error message, and a new line. Most usefully, the argument string is the name of the program that incurred the error. The errno is taken from the external variable errno that is set when errors occur but not cleared when non-erroneous calls are made.

To simplify variant formatting of messages, the vector of message strings, sock_errlist is provided; errno is used as an index in this table to get the message string without the new line.

Implementation Notes

This implementation of perror() function front ends the C library perror() function. The list of socket errors is stored in an array of character pointers in the variable sock_errlist. Instead of using sys_nerr variable, the maximum number of error codes defined by the constant ESMAX is used.

recv()

receive a message on a socket

Synopsis

```
#include <socket.h>
#include <uio.h>
int recv ( s, buf, len, flags )
int s;
char *buf;
int len, flags;
```

Description

The `recv()` function is used to receive incoming data that has been queued for a socket. This function is normally used to receive a reliable, ordered stream of data bytes on a socket of type `SOCK_STREAM`, but can also be used to receive datagrams on a socket of type `SOCK_ASSOC` if an association was formed with a `connect()` function.

The `recv()` function is normally used only on a connected socket (read `connect()` for more information), while `recvfrom()` and `recvmsg()` can be used to receive data on a socket whether it is in a connected state or not.

The address of the buffer into which the message is to be received is given by `buf`, and its size is given by `len`. The length of the message is returned. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from (read `socket()` for more information).

If no messages are available at the socket, `recv()` waits for a message to arrive, unless the socket is non-blocking (read `ioctl()` for more information) in which case a value of `-1` is returned with the external variable `errno` set to `EWOULDBLOCK`. The `select()` function can be used to determine when more data arrives.

The `flags` argument to `recv()` can be set to `MSG_OOB` (from `<socket.h>`) to receive out-of-band data, but only if out-of-band data is supported by the underlying protocol. Otherwise, `flags` should be set to `0`.

Return Value

The `recv()` function returns the number of bytes received if successful. A value of `0` is returned if an end-of-file condition exits, indicating no more `recv()` functions should be issued to this socket. Otherwise, the value `-1` is returned, and the error code stored in the global integer `errno` indicates the nature of the error.

Error Codes

The `recv()` function returns these error codes (from `<errno.h>`) in the global integer `errno`:

<code>EBADF</code>	An invalid descriptor was specified.
<code>ECONNABORTED</code>	The incoming connection request was aborted by the remote endpoint.
<code>ECONNREFUSED</code>	The remote endpoint refused to continue the connection.
<code>ECONNRESET</code>	The remote endpoint reset the connection request.

EDESTUNREACH	The remote destination is now unreachable.
EFAULT	The buffer passed by the user points to inaccessible memory.
EFAULT	The pointer to the storage where the socket library is to place the name of the remote endpoint passed by the user points to inaccessible memory.
EFAULT	The pointer to the length of the name buffer points to inaccessible storage.
EHOSTUNREACH	The remote host is now unreachable.
EINVAL	The number of bytes in the receive buffer is less than zero (0).
EINVAL	The number of bytes requested on the receive request is greater than the current maximum buffer space allocated or greater than the maximum receive request allowed to this type of endpoint.
EINVAL	A connectionless socket does not yet have a name bound to it.
EINVAL	A connectionless socket is being used and the length of the name passed by the user is zero (0).
ENETDOWN	The local network interface is down.
ENOBUFS	The system was unable to allocate an internal buffer. The operation may succeed when buffers become available.
ENOBUFS	The socket library cannot allocate the necessary buffer space within the API.
ENOTCONN	The socket being used requires a connection before data can be transferred, and no such connection currently exists.
ETIMEDOUT	The connection request by the remote endpoint timed out.
EWOULDBLOCK	The socket is marked non-blocking and the requested operation would block.

Implementation Notes

The peek option is not supported for `recv()`, and if `flags` is set to `MSG_PEEK`, an error (`EFAULT`) is generated. The option to receive out-of-band data (`MSG_OOB`) is not supported. The maximum number of vectored I/O structures (`struct iovec`) in an array per call is 16 (`MSG_IOVLEN`).

See Also

`recvfrom()`, `recvmsg()`, `read()`, `send()`, `select()`, `getsockopt()`, `socket()`

recvfrom()

receive a datagram on a socket

Synopsis

```
#include <socket.h>
#include <uio.h>
int recvfrom ( s, buf, len, flags, from, fromlen )
int s;
char *buf;
int len, flags;
struct sockaddr *from;
int *fromlen;
```

Description

The `recvfrom()` function is used to receive incoming data that has been queued for a socket. This function normally is used to receive datagrams on a socket of type `SOCK_ASSOC`, but can also be used to receive a reliable, ordered stream of data bytes on a connected socket of type `SOCK_STREAM`.

The `recvfrom()` function normally is used to receive datagrams from a socket, indicated by `s`. If `from` is non-zero, the source address of the datagram is filled in. The `fromlen` parameter is a value-result parameter, initialized to the size of the buffer associated with `from` and modified on return to indicate the actual size of the address stored there.

The address of a buffer into which the datagram is to be received is given by `buf`, and its size is given by `len`. The length of the datagram is returned. If a datagram is too long to fit in the supplied buffer, excess bytes might be discarded depending on the type of socket the datagram is received from (`read socket()` for more information).

If no datagrams are available at the socket, `recvfrom()` waits for a datagram to arrive, unless the socket is non-blocking (`read ioctl()` for more information), in which case a value of 1 is returned with the external variable `errno` set to `EWOULDBLOCK`. The `select()` function can be used to determine when more data arrives.

The `flags` argument to `recvfrom()` can be set to `MSG_OOB` (from `<socket.h>`) to receive out-of-band data, but only if out-of-band data is supported by the underlying protocol. Otherwise, `flags` should be set to 0.

Return Value

The `recvfrom()` function returns the number of bytes received if successful. A value of 0 is returned if an end-of-file condition exists, indicating no more `recvfrom()` functions should be issued to this socket. Otherwise, the value -1 is returned, and the error code stored in the global integer `errno` indicates the nature of the error.

Error Codes

The `recvfrom()` function returns these error codes (from `<errno.h>`) in the global integer `errno`:

`EBADF` An invalid descriptor was specified.

`ECONNABORTED` The incoming connection request was aborted by the remote endpoint.

ECONNREFUSED	The remote endpoint refused to continue the connection.
ECONNRESET	The remote endpoint reset the connection request.
EDESTUNREACH	The remote destination is now unreachable.
EFAULT	The buffer passed by the user points to inaccessible memory.
EFAULT	The pointer to the storage where the socket library is to place the name of the remote endpoint passed by the user points to inaccessible memory.
EFAULT	The pointer to the length of the name buffer points to inaccessible storage.
EHOSTUNREACH	The remote host is now unreachable.
EINVAL	The number of bytes in the receive buffer is less than zero (0).
EINVAL	The number of bytes requested on the receive request is greater than the current maximum buffer space allocated or greater than the maximum receive request allowed to this type of endpoint.
EINVAL	A connectionless socket does not yet have a name bound to it.
EINVAL	A connectionless socket is being used and the length of the name passed by the user is zero (0).
ENETDOWN	The local network interface is down.
ENOBUFS	The system was unable to allocate an internal buffer. The operation might succeed when buffers become available.
ENOBUFS	The socket library cannot allocate the necessary buffer space within the API.
ENOTCONN	The socket being used requires a connection before data may be transferred, and no such connection currently exists.
ETIMEDOUT	The connection request by the remote endpoint timed out.
EWOULDBLOCK	The socket is marked non-blocking and the requested operation would block.

Implementation Notes

The peek option is not supported for `recvfrom()`, and if `flags` is set to `MSG_PEEK`, it is ignored. The option to receive out-of-band data (`MSG_OOB`) is not supported and is also ignored. The maximum number of vectored I/O structures (`struct iovec`) in an array per call is `16(MSG_IOVLEN)`.

See Also

`recv()`, `recvmsg()`, `read()`, `send()`, `select()`, `getsockopt()`, `socket()`

recvmsg()

receive a message on a socket

Synopsis

```
#include <socket.h>
#include <uio.h>
int recvmsg ( s, msg, flags )
int s;
struct msghdr msg [ ];
int flags;
```

Description

The `recvmsg()` function is used to receive incoming data that has been queued for a connected or unconnected socket `s`. Data is received in “scatter” mode and placed into noncontiguous buffers.

The argument `msg` is the pointer of a structure, `msghdr`, used to minimize the number of directly supplied parameters. This structure has this form, as defined in `<socket.h>`:

```
struct msghdr
{
    caddr_t          msg_name;
    int             msg_namelen;
    struct iovec     *msg_iov;
    int             msg_iovlen;
    caddr_t          msg_accrightrights;
    int             msg_accrightrightslen;
};
```

Here `msg_name` and `msg_namelen` specify the source address if the socket is unconnected; `msg_name` can be given as a null pointer if the source address is not desired or required. The `msg_iov` and `msg_iovlen` describe the “scatter” locations, as described in `read()`. A buffer to receive any access rights sent along with the message is specified in `msg_accrightrights`, which has length `msg_accrightrightslen`. Access rights are currently limited to file descriptors, with each occupying the size of an `int`.

The length of the message is returned. If a message is too long to fit in the supplied buffer, excess bytes might be discarded depending on the type of socket the message is received from (`read socket()` for more information).

If no messages are available at the socket, `recvmsg()` waits for a message to arrive, unless the socket is non-blocking (`read ioctl()` for more information), in which case a value of 1 is returned with the external variable `errno` set to `EWOULDBLOCK`. The `select()` function can be used to determine when more data arrives.

The `flags` argument to `recvmsg()` can be set to `MSG_OOB` (from `<socket.h>`) to receive out-of-band data, but only if out-of-band data is supported by the underlying protocol. Otherwise, `flags` should be set to 0.

Return Value

The `recvmsg()` function returns the number of bytes received if successful. A value of 0 is returned if an end-of-file condition exits, indicating no more `recvmsg()` functions should be issued to this socket. Otherwise, the value -1 is returned, and the error code stored in the global integer `errno` indicates the nature of the error.

Error Codes

The recvmsg() function returns these error codes (from <errno.h>) in the global integer errno:

EBADF	An invalid descriptor was specified.
ECONNABORTED	The incoming connection request was aborted by the remote endpoint.
ECONNREFUSED	The remote endpoint refused to continue the connection.
ECONNRESET	The remote endpoint reset the connection request.
EDESTUNREACH	Remote destination is now unreachable.
EFAULT	The buffer passed by the user points to inaccessible memory.
EFAULT	The pointer to the storage where the socket library is to place the name of the remote endpoint passed by the user points to inaccessible memory.
EFAULT	The pointer to the length of the name buffer points to inaccessible storage.
EFAULT	The msghdr pointer points to inaccessible memory.
EFAULT	The iovec structure pointer within the msghdr structure points to inaccessible memory.
EHOSTUNREACH	Remote host is now unreachable.
EINVAL	The number of bytes in the receive buffer is less than zero (0).
EINVAL	The number of bytes requested on the receive request is greater than the current maximum buffer space allocated or greater than the maximum receive request allowed to this type of endpoint.
EINVAL	A connectionless socket does not yet have a name bound to it.
EINVAL	A connectionless socket is being used and the length of the name passed by the user is zero (0).
EINVAL	The number of iovec structures specified by the msghdr is less than or equal to 0.
EMSGSIZE	The number of iovec structures specified by the msghdr structure is greater than 16.
ENETDOWN	Local network interface is down.
ENOBUFS	The system was unable to allocate an internal buffer. The operation might succeed when buffers become available.
ENOBUFS	The socket library cannot allocate the necessary buffer space within the API.
ENOTCONN	The socket being used requires a connection before data can be transferred, and no such connection currently exists.
ETIMEDOUT	The connection request by the remote endpoint timed out.
EWOULDBLOCK	The socket is marked non-blocking and the requested operation would block.

Implementation Notes

The peek option is not supported for `recvmsg()`, and if `flags` is set to `MSG_PEEK`, it is ignored. The option to receive out-of-band data (`MSG_OOB`) is not supported and is ignored. The maximum number of vectored I/O structures (`struct iovec`) in an array per call is 16 (`MSG_IOVLEN`).

See Also

`recv()`, `recvfrom()`, `read()`, `send()`, `select()`, `getsockopt()`, `socket()`

select()

synchronous I/O multiplexing

Synopsis

```
#include <socket.h>
#include <uio.h>
int select ( nfd, readfds, writefds, exceptfds, timeout )
int nfd;
fd_set *readfds, *writefds, *exceptfds;
struct timeval *timeout;
FD_SET ( fd, &fdset )
FD_CLR ( fd, &fdset )
FD_ISSET ( fd, &fdset )
FD_ZERO ( &fdset )
int fd;
fd_set fdset;
```

Description

The select() function is used to synchronize processing of several sockets operating in non-blocking mode. Sockets that are ready for reading, ready for writing, or have a pending exceptional condition can be selected. If no sockets are ready for processing, the select() function can block indefinitely or wait for a specified period of time (which may be zero) and then return.

select() examines the I/O descriptor sets whose addresses are passed in readfds, writefds, and exceptfds to see if some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. The first nfd descriptors are checked in each set (in other words, the descriptors from 0 through nfd-1 in the descriptor sets are examined). On return, select() replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. The total number of ready descriptors in all the sets is returned.

The descriptor sets are stored as bit fields in arrays of integers. These macros are provided for manipulating such descriptor sets:

- FD_ZERO (&fdset) Initializes a descriptor set fdset to the null set.
- FD_SET (fd, &fdset) Includes a particular descriptor fd in fdset.
- FD_CLR (fd, &fdset) Removes fd from fdset.
- FD_ISSET (fd, &fdset) Is non-zero if fd is a member of fdset, zero otherwise.

The behavior of these macros is undefined if a descriptor value is less than zero or greater than or equal to FD_SETSIZE, which normally is at least equal to the maximum number of descriptors supported by the system.

If timeout is a non-zero pointer, it specifies a maximum interval to wait for the selection to complete. If timeout is a zero pointer, the select blocks indefinitely. To affect a poll, the timeout argument should be non-zero, pointing to a zero-valued timeval structure.

Any of readfds, writefds, and exceptfds can be given as zero pointers if no descriptors are of interest.

Sockets use MVS STIMER services. Socket applications may use up to fifteen STIMER calls per task control block (TCB). STIMER CANCEL ID=ALL must not be used by socket applications.

Note For release 3.1 and higher, note that the `select()` and `mvselect()` functions have changed slightly for SAS/C users. SAS/C allows signals to be raised. `select()` may complete if the signal is raised without the conditions on the `select()` parameters having been met. This new feature allows you to use signal functionality, issue a blocking receive, and do a send from a signal handler.

Return Value

If successful, `select()` returns the number of ready descriptors that are contained in the descriptor sets, or 0 if the time limit expires. Otherwise, the value -1 is returned, and the error code stored in the global integer `errno` indicates the nature of the error. If `select()` returns with an error, including one due to an interrupted call, the descriptor sets are unmodified.

Error Codes

The `select()` function returns these error codes (from `<errno.h>`) in the global integer `errno`:

EBADF	One of the descriptor sets specified an invalid descriptor.
EINVAL	The specified time limit is invalid. One of its components is negative or too large.
EINVAL	The count of socket descriptors to check is less than or equal to 0.
EINVAL	No valid socket descriptors were referenced by the three sets of masks passed by the caller.

Implementation Notes

The implementation of `select()` provided with the API supports only descriptors associated with sockets.

See Also

`accept()`, `connect()`, `read()`, `write()`, `recv()`, `send()`, `recvfrom()`, `sendto()`, `recvmsg()`, `sendmsg()`, `mvselect()`

send()

send a message on a socket

Synopsis

```
#include <socket.h>
#include <uio.h>
int send ( s, msg, len, flags )
int s;
char *msg;
int len, flags;
```

Description

The send() function is used to send outgoing data on a connected socket s. This function is normally used to send a reliable, ordered stream of data bytes on a socket of type SOCK_STREAM, but can also be used to send datagrams on a socket of type SOCK_ASSOC, if an association has been formed with a connect() function.

The send() function normally is used only on a connected socket (read connect() for more information), while sendto() and sendmsg() can be used to send data on a socket whether it is connected or not.

The location of the message is given by msg, and its size is given by len. The number of bytes sent is returned. If the message is too long to pass automatically through the underlying protocol, the error EMSGSIZE is returned, and the message is not transmitted.

If no buffer space is available at the socket to hold the message to be transmitted, send() normally blocks. However, if the socket has been placed in non-blocking I/O mode (read ioctl() for more information), a value of -1 is returned with the external variable errno set to EMSGSIZE, and the message is not transmitted. The select() function can be used to determine when it is possible to send more data.

The flags argument to send() may be set to MSG_OOB (from <socket.h>) to send out-of-band data, if the underlying protocol supports this notion. Otherwise, flags should be set to 0.

Return Value

If successful, send() returns the number of bytes sent. Otherwise, the value -1 is returned, and the error code stored in the global integer errno indicates the nature of the error.

Error Codes

The send() function returns these error codes (from <errno.h>) in the global integer errno:

- | | |
|---------------|--|
| EADDRNOTAVAIL | A connectionless socket had a send request issued to it, but the user passed a name of the remote endpoint that was invalid. Either the remote port is zero (0) or the remote address is INADDR_ANY. |
| EAFNOSUPPORT | The name of the remote endpoint to send the data to specified a domain other than AF_INET. |
| EBADF | An invalid descriptor was specified. |
| ECONNABORTED | The incoming connection request was aborted by the remote endpoint. |

ECONNREFUSED	The remote endpoint refused to continue the connection.
ECONNRESET	The remote endpoint reset the connection request.
EDESTADDRREQ	A connectionless socket is being used and no name of the remote endpoint has been passed by the user.
EDESTUNREACH	Remote destination is now unreachable.
EFAULT	The buffer passed by the user points to inaccessible memory.
EFAULT	The pointer to the name of the remote endpoint passed by the user points to inaccessible memory.
EHOSTUNREACH	Remote host is now unreachable.
EINVAL	The number of bytes to transmit is less than or equal to zero (0).
EINVAL	A connectionless socket does not yet have a name bound to it.
EINVAL	A connectionless socket is being used and the length of the name passed by the user is zero (0).
EINVAL	A send request was issued to a socket that is operating in connectionless mode, but the user did not pass a name of the remote endpoint to which to send the data.
EISCONN	A socket associated with a remote endpoint has been issued a send request in which the user specified a remote endpoint name.
EMSGSIZE	The socket requires that the message be sent automatically, and the size of the message to be sent made this impossible.
ENETDOWN	Local network interface is down.
ENOBUFS	The system was unable to allocate an internal buffer. The operation might succeed when buffers become available.
ENOBUFS	The output queue for a network interface was full. This generally indicates that the interface has stopped sending, but it can be caused by transient congestion.
ENOTCONN	The socket being used requires a connection before data can be transferred, and no such connection currently exists.
EOPNOTSUPP	The user tried to send urgent data (MSG_OOB) on a socket that does not support this concept.
EPIPE	An attempt was made to send to a socket that is not connected to or associated with a peer socket.
ETIMEDOUT	The connection request by the remote endpoint timed out.
EWOULDBLOCK	The socket is marked non-blocking and the requested operation would block.

Implementation Notes

The send() function does not support the MSG_DONTROUTE option and should not be set in flags. The option MSG_MORE lets the user program inform sockets of the fact that more data is about to be written to the socket. This may be used by the transport provider to influence its packetizing algorithm. The maximum number of vectored I/O structures (struct iovec) in an array per call is 16 (MSG_IOVLEN).

See Also

recv(), sendto(), sendmsg(), select(), getsockopt(), socket(), write()

sendmsg()

send a message on a socket

Synopsis

```
#include <socket.h>
#include <uio.h>
int sendmsg ( s, msg, flags )
int s;
struct msghdr msg [ ];
int flags;
```

Description

The `sendmsg()` function is used to send outgoing data on a connected or unconnected socket `s`. Data is sent in gather mode from a list of noncontiguous buffers.

The argument `msg` is a pointer to a structure, `msghdr`, used to minimize the number of directly supplied parameters. This structure has this form, as defined in `<socket.h>`:

```
struct msghdr
{
    caddr_t      msg_name;
    int         msg_namelen;
    struct iovec *msg_iov;
    int         msg_iovlen;
    caddr_t      msg_accrightrights;
    int         msg_accrightrightslen;
};
```

Here `msg_name` and `msg_namelen` specify the destination address if the socket is unconnected. `msg_name` can be given as a null pointer if the destination address is not required. The `msg_iov` and `msg_iovlen` describe the gather locations, as described in `writev()`. A buffer containing any access rights to send with the message is specified in `msg_accrightrights`, which has length `msg_accrightrightslen`. Access rights are currently limited to file descriptors, each occupying the size of an `int`.

The number of bytes sent is returned. If the message is too long to pass automatically through the underlying protocol, the error `EMSGSIZE` is returned, and the message is not transmitted.

If no message space is available at the socket to hold the message to be transmitted, `sendmsg()` normally blocks. However, if the socket has been placed in non-blocking I/O mode (`read ioctl()` for more information), a value of `-1` is returned with the external variable `errno` set to `EMSGSIZE`, and the message is not transmitted. The `select()` function can be used to determine when it is possible to send more data.

The `flags` argument to `sendmsg()` can be set to `MSG_OOB` (from `<socket.h>`) to send out-of-band data if the underlying protocol supports this notion. Otherwise, `flags` should be set to `0`.

Return Value

If successful, `sendmsg()` returns the number of bytes sent. Otherwise, the value `-1` is returned, and the error code stored in the global integer `errno` indicates the nature of the error.

Error Codes

The sendmsg() function returns these error codes (from <errno.h>) in the global integer errno:

EADDRNOTAVAIL	A connectionless socket had a send request issued to it, but the user passed a name of the remote endpoint that was invalid. Either the remote port is zero (0) or the remote address is INADDR_ANY.
EAFNOSUPPORT	The name of the remote endpoint to send the data to specified a domain other than AF_INET.
EBADF	An invalid descriptor was specified.
ECONNABORTED	The incoming connection request was aborted by the remote endpoint.
ECONNREFUSED	The remote endpoint refused to continue the connection.
ECONNRESET	The remote endpoint reset the connection request.
EDESTADDRREQ	A connectionless socket is being used and no name of the remote endpoint has been passed by the user.
EDESTUNREACH	Remote destination is now unreachable.
EFAULT	The buffer passed by the user points to inaccessible memory.
EFAULT	The pointer to the name of the remote endpoint passed by the user points to inaccessible memory.
EFAULT	The pointer to the msghdr structure points to inaccessible memory.
EFAULT	The msghdr has an iovec pointer that points to inaccessible memory.
EHOSTUNREACH	Remote host is now unreachable.
EINVAL	The number of bytes to transmit is less than or equal to zero (0).
EINVAL	A connectionless socket does not yet have a name bound to it.
EINVAL	A connectionless socket is being used and the length of the name passed by the user is zero (0).
EINVAL	A send request was issued to a socket that is operating in connectionless mode, but the user did not pass a name of the remote endpoint to which to send the data.
EINVAL	The msghdr structure specifies an array of less than or equal to 0 iovec elements.
EISCONN	A socket that is associated with a remote endpoint has been issued a send request in which the user specified a remote endpoint name.
EMSGSIZE	The socket requires that the message be sent automatically, and the size of the message to be sent made this impossible.
EMSGSIZE	The msghdr structure specifies an array of iovec elements of less than or equal to 0.
ENETDOWN	Local network interface is down.

ENOBUFS	The system was unable to allocate an internal buffer. The operation might succeed when buffers become available.
ENOBUFS	The output queue for a network interface was full. This generally indicates that the interface has stopped sending, but can be caused by transient congestion.
ENOTCONN	The socket being used requires a connection before data can be transferred, and no such connection currently exists.
EOPNOTSUPP	The user tried to send urgent data (MSG_OOB) on a socket that does not support this concept.
EPIPE	An attempt was made to send to a socket that is not connected to or associated with a peer socket.
ETIMEDOUT	The connection request by the remote endpoint timed out.
EWouldBlock	The socket is marked non-blocking and the requested operation would block.

Implementation Notes

The `sendmsg()` function does not support the `MSG_DONTROUTE` option and should not be set in flags. The option `MSG_MORE` lets the user program inform sockets of the fact that there is more data about to be written to the socket. This can be used by the transport provider to influence its packetizing algorithm. The maximum number of vectored I/O structures (`struct iovec`) in an array per call is 16 (`MSG_IOVLEN`).

See Also

`recv()`, `sendto()`, `sendmsg()`, `select()`, `getsockopt()`, `socket()`, `write()`

sendto()

send a datagram on a socket

Synopsis

```
#include <socket.h>
#include <uio.h>
int sendto ( s, msg, len, flags, to, tolen )
int s;
char *msg;
int len, flags;
struct sockaddr *to;
int tolen;
```

Description

The `sendto()` function is used to send outgoing data on a connected or unconnected socket. This function normally is used to send datagrams on a socket of type `SOCK_DGRAM`, but can also be used to send a reliable, ordered stream of data bytes on a connected socket of type `SOCK_STREAM`.

The `sendto()` function normally is used to transmit a datagram from a socket that is unconnected. The socket is indicated by `s`. The destination address is given by `to`, and its length is given by `tolen`. If the socket is connected or associated with a destination (read `connect()` for more information), `to` and `tolen` can be set to 0.

The location of the datagram is given by `msg`, and its size is given by `len`. The number of bytes sent is returned. If the datagram is too long to pass automatically through the underlying protocol, the error `EMSGSIZE` is returned, and the datagram is not transmitted.

If no buffer space is available at the socket to hold the datagram to be transmitted, `sendto()` normally blocks. However, if the socket has been placed in non-blocking I/O mode (read `ioctl()` for more information), a value of -1 is returned with the external variable `errno` set to `EMSGSIZE`, and the datagram is not transmitted. The `select()` function can be used to determine when it is possible to send more data.

The `flags` argument to `sendto()` can be set to `MSG_OOB` (from `<socket.h>`) to send out-of-band data, if the underlying protocol supports this notion. Otherwise, `flags` should be set to 0.

Return Value

If successful, `sendto()` returns the number of bytes sent. Otherwise, the value -1 is returned, and the error code stored in the global integer `errno` indicates the nature of the error.

Error Codes

The `sendto()` function returns these error codes (from `<errno.h>`) in the global integer `errno`:

- | | |
|----------------------------|--|
| <code>EADDRNOTAVAIL</code> | A connectionless socket had a send request issued to it, but the user passed a name of the remote endpoint that was invalid. Either the remote port is zero (0) or the remote address is <code>INADDR_ANY</code> . |
| <code>EAFNOSUPPORT</code> | The name of the remote endpoint to send the data to specified a domain other than <code>AF_INET</code> . |
| <code>EBADF</code> | An invalid descriptor was specified. |

ECONNABORTED	The incoming connection request was aborted by the remote endpoint.
ECONNREFUSED	The remote endpoint refused to continue the connection.
ECONNRESET	The remote endpoint reset the connection request.
EDESTADDRREQ	A connectionless socket is being used and no name of the remote endpoint has been passed by the user.
EDESTUNREACH	The remote destination is now unreachable.
EFAULT	The buffer passed by the user points to inaccessible memory.
EFAULT	The pointer to the name of the remote endpoint passed by the user points to inaccessible memory.
EHOSTUNREACH	The remote host is not unreachable.
EINVAL	The number of bytes to transmit is less than or equal to zero (0).
EINVAL	A connectionless socket does not yet have a name bound to it.
EINVAL	A connectionless socket is being used and the length of the name passed by the user is zero (0).
EINVAL	A send request was issued to a socket that is operating in connectionless mode, but the user did not pass a name of the remote endpoint to which to send the data.
EISCONN	A socket that is associated with a remote endpoint has been issued a send request in which the user specified a remote endpoint name.
EMSGSIZE	The socket requires that the message be sent automatically, and the size of the message to be sent made this impossible.
ENETDOWN	The local network interface is down.
ENOBUFS	The system was unable to allocate an internal buffer. The operation might succeed when buffers become available.
ENOBUFS	The output queue for a network interface was full. This generally indicates that the interface has stopped sending, but can be caused by transient congestion.
ENOTCONN	The socket being used requires a connection before data can be transferred, and no such connection currently exists.
EOPNOTSUPP	The user tried to send urgent data (MSG_OOB) on a socket that does not support this concept.
EPIPE	An attempt was made to send to a socket that is not connected to or associated with a peer socket.
ETIMEDOUT	The connection request by the remote endpoint timed out.
EWouldBLOCK	The socket is marked non-blocking and the requested operation would block.

Implementation Notes

The `sendto()` function does not support the `MSG_DONTROUTE` option and should not be set in flags. The option `MSG_MORE` lets the user program inform sockets of the fact that there is more data about to be written to the socket. This can be used by the transport provider to influence its packetizing algorithm. The maximum number of vectored I/O structures (`struct iovec`) in an array per call is 16 (`MSG_IOVLEN`).

See Also

`recv()`, `send()`, `sendmsg()`, `select()`, `getsockopt()`, `socket()`, `write()`

setlogmask()

set log priority mask

Synopsis

```
#include <syslog.h>
int setlogmask ( maskpri )
int maskpri;
```

Description

The setlogmask() function sets the log priority mask to maskpri and returns the previous mask. Calls to syslog() with a priority not set in maskpri are rejected. The mask for an individual priority pri is calculated by the macro LOG_MASK (pri); the mask for all priorities up to and including toppri is given by the macro LOG_UPTO (toppri). The default allows all priorities to be logged.

See Also

closelog(), openlog(), syslog(), vsyslog()

setsockopt()

set options on a socket

Synopsis

```
#include <socket.h>
#include <uio.h>
#include <inet.h>
#include <tcp.h>
int setsockopt ( s, level, optname, optval, optlen )
int s, level, optname;
char *optval;
int optlen;
```

Description

The `setsockopt()` function is used to manipulate options associated with a socket. Options always exist at the socket level and can also exist at layers within the underlying protocols. Options are retrieved with the `getsockopt()` function. Options can exist at multiple protocol levels; they are always present at the uppermost socket level.

When manipulating socket options, the level at which the option resides and the name of the option must be specified. To manipulate options at the socket level, `level` is specified as `SOL_SOCKET`. To manipulate options at any other level the protocol number of the appropriate protocol controlling the option is supplied.

Example

To indicate that a TCP protocol option is to be changed, `level` should be set to the protocol number of TCP. Read `getprotobyname()` for more information.

The parameters `optval` and `optlen` identify a buffer that contains the option value. The `optlen` parameter is the length of the option value in bytes.

The `optname` parameter and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The include file `<socket.h>` contains definitions for socket level options. Options at other protocol levels vary in format and name; consult the appropriate appendix for additional information.

Most socket-level options require an `int` parameter for `optval`. For boolean options, a non-zero value indicates the option is to be enabled, and a zero value indicates the option is to be disabled. `SO_LINGER` uses a `struct linger` parameter, defined in `<socket.h>`, that specifies the desired state of the option and the linger interval.

These options are recognized at the socket level. Except as noted, each can be set with `setsockopt()` and examined with `getsockopt()`:

Table 3-7 Socket-level Options

Option	Definition
SO_BROADCAST	Requests permission to send broadcast datagrams on the socket.
SO_DEBUG	Enables debugging in the socket modules. This option is implemented slightly differently than UNIX in that on UNIX it allows for debugging at the underlying protocol modules.

Table 3-7 Socket-level Options (Continued)

Option	Definition
SO_DONTROUTE	Indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.
SO_KEEPALIVE	Enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken and processes using the socket are notified via an errno ETIMEDOUT.
SO_LINGER	Controls the action taken when unsent messages are queued on socket and a close() is performed. If the socket promises reliable delivery of data and SO_LINGER is set to a value other than 0, the system blocks the process on the close() attempt until it is able to transmit the data or until the number of seconds specified by the SO_LINGER option expire or until the connection has timed out. If SO_LINGER is set and the interval is 0, the socket is closed once the system has scheduled that an orderly release be performed.
SO_OOBINLINE	This option is not currently supported by this release of the socket library. It is implemented on UNIX, and its purpose is to request that out-of-band data be placed in the normal data input queue as received by protocols that support out-of-band data; it is then accessible with recv() or read() functions without the MSG_OOB flag.
SO_RCVBUF	Adjusts the normal buffer size allocated for input. The buffer size can be increased for high-volume connections to improve throughput or can be decreased to limit the possible backlog of incoming data. The system places an absolute limit on this value. This implementation of sockets provides for this option for backward compatibility, but also allows for buffer options that are more specific to the underlying API and therefore provide a better method of controlling a socket's buffering characteristics. These options are SO_RCVBYTCNT and SO_RCVREQCNT. All buffering options can be set only once by the socket user and must be done before any data is sent or received on the socket.
SO_RCVBYTCNT	Adjusts the number of bytes allocated to the receive circular buffer for a socket. This option is specific to this implementation of sockets and therefore is not portable to other socket libraries. This option can be set only once successfully, and if the receive byte count of the circular buffer is changed, it must be done prior to sending or receiving any data on the socket.
SO_RCVLOWAT	Adjusts the size of the receive low water mark.
SO_RCVREQCNT	Adjusts the number of receive requests that can be active within the socket library at a given time. This option is specific to this implementation of sockets and therefore is not portable to other socket libraries. This option can be set only once successfully, and if the receive request count is changed, it must be done prior to sending or receiving any data on the socket.
SO_READFRAG	This option lets a user of a datagram type socket that preserves message boundaries read a datagram a piece at a time. Traditionally, with UNIX sockets, if a user issues a read request for 100 bytes and the datagram being read consists of 120 bytes, the socket returns the first 100 bytes to the caller and then flushes the remaining 20 bytes. This default method of operation by this implementation of sockets can be overridden by using this option. This option does not allow for the parsing of pieces of a single datagram into a single user buffer. When this option is used, the user must determine the boundaries of datagrams. This option is specific to this implementation and is not portable to other socket implementations.
SO_REUSEADDR	Indicates that the rules used in validating addresses supplied in a bind() function should allow reuse of local addresses.

Table 3-7 Socket-level Options (Continued)

Option	Definition
SO_SENDALL	This option guarantees that any form of send request (send(), sendto(), sendmsg(), write(), or writev()) that is done in a blocking mode transmits all the data specified by the user. Traditionally, BSD sockets send as many bytes as the current buffering allocation allowed for and then return to the user with a count of the actual number of bytes transmitted. If the user requested that a write of 200 bytes be done but there currently was only buffering space for 150 bytes, the socket queues 150 bytes for transmission and returns a count of 150 to the caller to indicate that 50 bytes could not be transmitted due to lack of buffer space. This implementation of sockets acts identically to a UNIX socket under the same scenario under the default setting of the socket options. However, if this option is turned ON in Cisco IOS for S/390 sockets, the socket blocks the user until all of the data has been queued for transmission or some type of error occurs. This option is specific to this implementation.
SO_SNDBUF	Adjusts the normal buffer size allocated for output. The buffer size can be increased for high-volume connections to improve throughput or can be decreased to limit the possible backlog of outgoing data. The system places an absolute limit on this value. This implementation of sockets provides this option for backward compatibility. It also allows for buffer options that are more specific to the underlying API and therefore provides a better method of controlling a socket's buffering characteristics. These options are SO_SNDBYTCNT and SO_SNDREQCNT. All buffering options can be set only once by the socket user and must be done before any data is sent or received on the socket.
SO_SNDBYTCNT	Adjusts the number of bytes allocated to the send circular buffer for a socket. This option is specific to this implementation of sockets and therefore is not portable to other socket libraries. This option can be set only once successfully, and if the send byte count of the circular buffer is changed, it must be done prior to sending or receiving any data on the socket.
SO_SNDLOWAT	Adjusts the size of the send low water mark.
SO_SNDREQCNT	Adjusts the number of send requests that can be active within the socket library at a given time. This option is specific to this implementation of sockets and therefore is not portable to other socket libraries. This option can be set only once successfully, and if the send request count is changed, it must be done prior to sending or receiving any data on the socket.
SO_USELOOPBACK	Requests that the loopback interface is used rather than a real physical interface.

The following are TCP level options:

TCP_NODELAY	Ensures that TCP type sockets (SOCK_STREAM) send the data as soon as possible and do not wait for more data or a given amount of time to enhance the packetizing algorithm. This option is similar to the BSD UNIX socket option.
TCP_MAXSEG	This option is not supported by this implementation of sockets at the present release. On UNIX, this option lets the user of a SOCK_STREAM socket declare the value of the maximum segment size for TCP to use when negotiating this value with its remote endpoint.

The following option is at the UDP level.

UDP_CHECKSUM	Sets whether UDP checksum computation is to be performed.
--------------	---

The following options are at the IP level

IP_HDRINCL	Specifies that the application include the IP header in data for the SEND option. Applicable for RAW sockets only.
IP_OPTIONS	Sets specific options in the IP header
IP_TOS	Sets the type-of-service field in IP header of outgoing packets.
IP_TTL	Sets the time-to-live field in IP header of outgoing packets.

Return Value

If setsockopt() is successful, a value of 0 is returned. A return value of -1 indicates an error, and the error code stored in the global integer errno indicates the nature of the error.

Error Codes

The setsockopt() function returns these error codes (from <errno.h>) in the global integer errno:

EBADF	The s argument is not a valid descriptor.
EFAULT	The pointer to the value buffer points to inaccessible memory.
EFAULT	The pointer to the value buffer length points to inaccessible memory.
EINVAL	The size of the value buffer does not equal the size of the option. Most options require an integer length buffer, or in the case of SO_LINGER the buffer must be the size of the linger structure.
EINVAL	The size of the option buffer is greater than the maximum allowed by the API.
EINVAL	The option is not supported at the level requested.
EINVAL	No options can be read from the protocol layers.

Implementation Notes

These options are recognized at the socket level on BSD UNIX systems and for OpenEdition Converged sockets, but are not supported by the API. If any of these options are referenced, an error is generated:

SO_RCVLOWAT	Sets the receive low water buffering mark.
SO_RCVTIMEO	Sets the receive timeout value. This option is not currently implemented in UNIX.
SO_SNDLOWAT	Sets the send low water buffering mark.
SO_SNDTIMEO	Sets the send timeout value. This option is not currently implemented in UNIX.

The effect of setting the supported socket level options can differ from that which occurs in a UNIX environment.

Example

The SO_DEBUG option enables the API debugging facilities, but the output produced by those facilities can differ from that produced on a UNIX system.

See Also

getsockopt(), ioctl(), socket()

shutdown()

shut down part of a full-duplex connection

Synopsis

```
#include <socket.h>
#include <uio.h>
int shutdown ( s, how )
int s, how;
```

Description

The shutdown() function is used to gracefully shut down a socket. The input path can be shut down while continuing to send data, the output path can be shut down while continuing to receive data, or the socket can be shut down in both directions at once. Data queued for transmission is not lost.

The shutdown() function causes all or part of a full-duplex connection on the socket associated with s to be shut down.

If how is 0, further receives are disallowed. If how is 1, further sends are disallowed. If how is 2, further sends and receives are disallowed.

Return Value

If shutdown() is successful, a value of 0 is returned. A return value of -1 indicates an error, and the error code stored in the global integer errno indicates the nature of the error.

Error Codes

The shutdown() function returns these error codes (from <errno.h>) in the global integer errno:

EBADF	The s argument is not a valid descriptor.
ECONNABORTED	The connection was aborted by a local action of the API.
ECONNREFUSED	The remote endpoint refused to continue the connection.
ECONNRESET	The remote endpoint reset the connection request.
ETIMEDOUT	The connection timed out.

See Also

connect(), socket()

socket()

create an endpoint for communication

Synopsis

```
#include <socket.h>
#include <uio.h>
int socket ( domain, type, protocol )
int domain, type, protocol;
```

Description

The `socket()` function creates an endpoint in a communications domain. The endpoint is called a socket. When the socket is created, a protocol is selected and a descriptor is returned to represent the socket. The socket descriptor is used in all subsequent functions referencing the socket. Only sockets in the Internet domain using TCP or UDP protocol are supported by this implementation.

The domain argument specifies a communications domain within which communication takes place; this selects the protocol family to use. The protocol family generally is the same as the address family for the addresses supplied in later operations on the socket. These families are defined in the include file `<socket.h>`.

This is the only protocol family currently recognized by this implementation:

```
PF_INET (Internet protocols)
```

The socket has the indicated type, which specifies the semantics of communication. These are the currently defined types:

Table 3-8 **Socket Types**

SOCK_STREAM	Provides sequenced, reliable, two-way connection-based byte streams. An out-of-band data transmission mechanism can be supported.
SOCK_DGRAM	Supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length).
SOCK_ASSOC	Supports datagrams (associations, unreliable messages of a fixed (typically small) maximum length).
SOCK_RAW	Supports datagrams (connectionless, unreliable messages of a fixed maximum length) with the complete IP header included in each datagram.

The protocol specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. However, it is possible that many protocols can exist, in which case a particular protocol must be specified in this manner. The protocol number to use is specific to the communication domain in which communication is to take place.

Sockets of type `SOCK_STREAM` are full-duplex byte streams. A stream socket must be in a connected state before any data can be sent or received on it. A connection to another socket is created with a `connect()` function. Once connected, data can be transferred using `read()` and `write()`

functions or some variant of the `send()` and `recv()` functions. When a session has been completed, a `close()` or `shutdown()` can be performed. Out-of-band data can also be transmitted as described in `send()` and received as described in `recv()`.

The communications protocols used to implement a `SOCK_STREAM` ensure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, the connection is considered broken and subsequent functions indicate an error with -1 returns and with `ETIMEDOUT` as the specific code in the global variable `errno`.

`SOCK_ASSOC` sockets allow for the association of one user to that of a remote user. Once an association is made using the connection oriented requests of `connect()` for the client and `listen()` and `accept()` for the server, datagrams can be transferred back and forth in similar fashion as `SOCK_STREAM`, although without the guarantees of delivery.

`SOCK_DGRAM` sockets allow sending of datagrams to correspondents named in `sendto()` functions. Datagrams generally are received with `recvfrom()`, which returns the next datagram with its return address.

`SOCK_RAW` sockets are similar to `SOCK_DGRAM` in that they use `sendto()` and `recvfrom()` calls, but each datagram must include the complete IP header. The `connect()` call may be used with sockets of type `SOCK_RAW` to fix the destination. If `connect()` is used, the `read()` (or `recv()`) and `write()` (or `send()`) calls may be used.

The operation of sockets is controlled by socket level options. These options are defined in the file `<socket.h>`. The `setsockopt()` and `getsockopt()` functions are used to set and get options, respectively.

An `atexit()` call at socket initialization ensures the close of sockets at program termination.

If you compile with the IBM/C compiler and use the explicit `dlfree()` of Cisco IOS for S/390 modules, your application must disable the `atexit()` call before the `socket()` with the following statement:

```
s0skcfg.exitfunc = NULL;      /* nullify atexit() call */
```

Also, you'll need to include:

```
#pragma map(s0skcf, "S0SKCF") /* for IBM prelinker*/
```

For `linkedit`, you'll need to specify:

```
INCLUDE CIROBJ(S0SKCF)
```

Note With the `atexit()` disabled, the programmer must remember to `close()` each socket.

Return Value

If successful, `socket()` returns a socket descriptor to be used in all subsequent functions referencing the socket. Otherwise, the value -1 is returned, and the error code stored in the global integer `errno` indicates the nature of the error.

Error Codes

The `socket()` function returns these error codes (from `<errno.h>`) in the global integer `errno`:

<code>EACCESS</code>	Permission to create a socket of the specified type and/or protocol is denied.
<code>ECONFIG</code>	Socket configuration has an error or a user session with the underlying API cannot be opened.
<code>EMFILE</code>	The system file table is full.
<code>EMFILE</code>	A new socket cannot be opened due to an API resource shortage or user endpoint allocation limit.
<code>ENOBUFS</code>	Insufficient buffer space is available. The socket cannot be created until sufficient resources are freed.
<code>EPROTONOSUPPORT</code>	The specified protocol is not supported within this domain.
<code>EPROTOTYPE</code>	The protocol type is not supported within this domain.

Implementation Notes

The protocol families (domains) `PF_UNIX`, `PF_NS`, and `PF_IMPLINK`, and the socket types `SOCK_RAW`, `SOCK_RDM`, and `SOCK_SEQPACKET` are not supported by this implementation.

`SOCK_DGRAM` provides the functionality of `SOCK_DGRAM` in the UNIX environment. `SOCK_DGRAM` also allows for the operation of a server endpoint in much the same fashion as `SOCK_STREAM` sockets.

See Also

`accept()`, `bind()`, `connect()`, `fcntl()`, `getsockname()`, `getsockopt()`, `ioctl()`, `listen()`, `read()`, `recv()`, `select()`, `send()`, `shutdown()`, `write()`, `closepass()`, `openold()`

sstat()

socket/file status

Synopsis

```
#include <socket.h>
#include <uio.h>
int sstat ( s, buf )
int s;
struct stat *buf;
```

Description

The sstat() obtains information about a file specified by the file descriptor argument. In the UNIX world this information consists of protection levels, owner ID, group ID, device type, size of the file in bytes, last access time, last modify time, file last status change time, optimal block size, number of blocks allocated, and other detailed information. The sstat() function in the socket library returns only the blocksize that equates to the buffer limits of the socket

Error Codes

The socket() function returns these error codes (from <errno.h>) in the global integer errno:.

EBADF The s argument is not a valid descriptor.

EFAULT buf points to an invalid address.

Implementation Notes

This function takes the place of the fstat function of UNIX.

strerror()

get error message string

Synopsis

```
#include <string.h>

char *strerror ( errno )
int errno;
```

Description

The strerror() function returns a pointer to the errno string associated with errno. strerror() uses the same set of error messages as perror().

See Also

perror()

syslog()

pass message to system log

Synopsis

```
#include <syslog.h>
#include <socket.h>
int syslog ( priority, message, /* parameters... */ )
int priority;
char *message;
```

Description

The syslog() function passes a message to the system log. The message is tagged with a priority of priority. The message looks like a printf string except that %m is replaced by the current error message (collected from errno). A trailing NEWLINE is added if needed.

Priorities are encoded as a facility and a level. The facility describes the part of the system generating the message. The level is selected from the bitwise inclusive OR of zero or more of the following flags, defined in the header <syslog.h>.

LOG_EMERG	A panic condition. This is normally broadcast to all users.
LOG_ALERT	A condition that should be corrected immediately, such as a corrupted system database.
LOG_CRIT	Critical conditions, such as hard device errors.
LOG_ERR	Errors.
LOG_WARNING	Warning messages.
LOG_NOTICE	Conditions that are not error conditions, but that may require special handling.
LOG_INFO	Informational messages.
LOG_DEBUG	Messages that contain information normally of use only when debugging a program.

See Also

openlog(), closelog(), vsyslog()

vsyslog()

log message with va_arg argument list

Synopsis

```
#include <syslog.h>
#include <stdarg.h>
int vsyslog ( priority, message, ap )
int priority;
const char *message;
va_list ap;
```

Description

The vsyslog() logs arguments to the log using variable length argument lists.

vsyslog() is the same as syslog() except that instead of being called with a variable number of arguments, it is called with an argument list as defined by va_arg. va_arg is included in the SAS include file stdarg.h and is one of a set of macros used to process a list of arguments. For more information on va_arg, consult your SAS/C compiler documentation.

See Also

syslog(), openlog(), closelog()

write()

write output

Synopsis

```
#include <socket.h>
#include <uio.h>
int write ( s, buf, nbytes )
int s;
char *buf;
int nbytes;
```

Description

The write() function is used to output data from a socket. It operates in the same manner as writing data to a file. In a normal UNIX environment where socket and file I/O are integrated, the write() function can be called with either a socket or file descriptor.

write() attempts to write nbytes of data to the socket referenced by the descriptor s from the buffer pointed to by buf. When using non-blocking I/O on sockets that are subject to flow control, write() might write fewer bytes than requested; the return value must be noted, and the remainder of the operation should be retried when possible.

Return Value

On successful completion, write() returns the number of bytes actually written. Otherwise, the value -1 is returned, and the error code stored in the global integer errno indicates the nature of the error.

Error Codes

The write() function returns these error codes (from <errno.h>) in the global integer errno:

- | | |
|---------------|--|
| EADDRNOTAVAIL | A connectionless socket had a send request issued to it, but the user passed a name of the remote endpoint that was invalid. Either the remote port is zero (0) or the remote address is INADDR_ANY. |
| EAFNOSUPPORT | The name of the remote endpoint to send the data to specified a domain other than AF_INET. |
| EBADF | An invalid descriptor was specified. |
| ECONNABORTED | The incoming connection request was aborted by the remote endpoint. |
| ECONNREFUSED | The remote endpoint refused to continue the connection. |
| ECONNRESET | The remote endpoint reset the connection request. |
| EDESTADDRREQ | A connectionless socket is being used and no name of the remote endpoint has been passed by the user. |
| EDESTUNREACH | Remote destination is now unreachable. |
| EFAULT | The buffer passed by the user points to inaccessible memory. |

EFAULT	The pointer to the name of the remote endpoint passed by the user points to inaccessible memory.
EHOSTUNREACH	Remote host is now unreachable.
EINVAL	The number of bytes to transmit is less than or equal to zero (0).
EINVAL	A connectionless socket does not yet have a name bound to it.
EINVAL	A connectionless socket is being used and the length of the name passed by the user is zero (0).
EINVAL	A send request was issued to a socket that is operating in connectionless mode, but the user did not pass a name of the remote endpoint to which to send the data.
EISCONN	A socket that is associated with a remote endpoint has been issued a send request in which the user specified a remote endpoint name.
EMSGSIZE	The socket requires that the message be sent automatically, and the size of the message to be sent made this impossible.
ENETDOWN	Local network interface is down.
ENOBUFS	The system was unable to allocate an internal buffer. The operation might succeed when buffers become available.
ENOBUFS	The output queue for a network interface was full. This generally indicates that the interface has stopped sending, but may be caused by transient congestion.
ENOTCONN	The socket being used requires a connection before data can be transferred, and no such connection currently exists.
EOPNOTSUPP	The user tried to send urgent data (MSG_OOB) on a socket that does not support this concept.
EPIPE	An attempt was made to send to a socket that is not connected to or associated with a peer socket.
ETIMEDOUT	The connection request by the remote endpoint timed out.
EWOULDBLOCK	The socket is marked non-blocking and the requested operation would block.

See Also

send(), sendfrom(), sendmsg(), select(), socket(), read()

writev()

write vectored output

Synopsis

```
#include <socket.h>
#include <uio.h>
int writev ( s, iov, iovcnt )
int s;
struct iovec *iov;
int iovcnt;
```

Description

The writev() function is used to output data from a socket in gather mode when the data are not contiguous. It operates in the same manner as writing data to a file. In a normal UNIX environment where socket and file I/O are integrated, the writev() function can be called with either a socket or file descriptor.

The writev() function performs the same action as write(), but gathers the output data from the iovcnt buffers specified by the members of the iov array:

```
iov [ 0 ], iov [ 1 ], ..., iov [ iovcnt-1 ]
```

The iovec structure is defined in this way:

```
struct iovec {
    caddr_t iov_base;
    int iov_len; };
```

Each iovec entry specifies the base address and length of an area in memory from which data should be written. The writev() function always writes a complete area before proceeding to the next iov entry. When using non-blocking I/O on sockets that are subject to flow control, writev() might write fewer bytes than requested; the return value must be noted, and the remainder of the operation should be retried when possible.

Return Value

On successful completion, writev() returns the total number of bytes actually written. Otherwise, the value -1 is returned, and the error code stored in the global integer errno indicates the nature of the error.

Error Codes

The writev() function returns these error codes (from <errno.h>) in the global integer errno:

- EADDRNOTAVAIL** A connectionless socket had a send request issued to it, but the user passed a name of the remote endpoint that was invalid. Either the remote port is zero (0) or the remote address is INADDR_ANY.
- EAFNOSUPPORT** The name of the remote endpoint to send the data to specified a domain other than AF_INET.
- EBADF** An invalid descriptor was specified.
- ECONNABORTED** The incoming connection request was aborted by the remote endpoint.

ECONNREFUSED	The remote endpoint refused to continue the connection.
ECONNRESET	The remote endpoint reset the connection request.
EDESTADDRREQ	A connectionless socket is being used and no name of the remote endpoint has been passed by the user.
EDESTUNREACH	Remote destination is now unreachable.
EFAULT	The buffer passed by the user points to inaccessible memory.
EFAULT	The pointer to the name of the remote endpoint passed by the user points to inaccessible memory.
EFAULT	The pointer to the iovec array points to inaccessible memory.
EHOSTUNREACH	Remote host is now unreachable.
EINVAL	The number of bytes to transmit is less than or equal to zero (0).
EINVAL	A connectionless socket does not yet have a name bound to it.
EINVAL	A connectionless socket is being used and the length of the name passed by the user is zero (0).
EINVAL	A send request was issued to a socket that is operating in connectionless mode, but the user did not pass a name of the remote endpoint to which to send the data.
EINVAL	The count of iovec elements in the array is greater than 16 or less than or equal to 0.
EISCONN	A socket associated with a remote endpoint has been issued a send request in which the user specified a remote endpoint name.
EMSGSIZE	The socket requires that the message be sent automatically, and the size of the message to be sent made this impossible.
ENETDOWN	Local network interface is down.
ENOBUFS	The system was unable to allocate an internal buffer. The operation might succeed when buffers become available.
ENOBUFS	The output queue for a network interface was full. This generally indicates that the interface has stopped sending, but may be caused by transient congestion.
ENOTCONN	The socket being used requires a connection before data can be transferred, and no such connection currently exists.
EOPNOTSUPP	The user tried to send urgent data (MSG_OOB) on a socket that does not support this concept.
EPIPE	An attempt was made to send to a socket that is not connected to or associated with a peer socket.
ETIMEDOUT	The connection request by the remote endpoint timed out.
EWOULDBLOCK	The socket is marked non-blocking and the requested operation would block.

Implementation Notes

The maximum number of vectored I/O structures (struct iovec) in an array per call is 16 (MSG_IOVLEN).

See Also

send(), sendfrom(), sendmsg(), select(), socket(), readv()

writev()
