Berkeley Sockets/iX Reference Manual

Printed in USA 02/94

Edition 3
E0294

Notice

Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Information Networks Division
19420 Homestead Road
Cupertino, CA 95014

Printing History

New editions are complete revisions of the manual. Updates, which are issued between editions, contain additional and replacement pages to be merged into the manual. The dates on the title page change only when a new edition or a new update is published.

Edition 1 . . . . . June 1992

Edition 2 . . . . . October 1992

Edition 3 . . . . . February 1994

Preface

MPE/iX, Multiprogramming Executive with Integrated POSIX, is the latest in a series of forward-compatible operating systems for the HP 3000 line of computers.

In HP documentation and in talking with HP 3000 users, you will encounter references to MPE XL, the direct predecessor of MPE/iX. MPE/iX is a superset of MPE XL. All programs written for MPE XL will run without change under MPE/iX. You can continue to use MPE XL system documentation, although it may not refer to features added to the operating system to support POSIX (for example, hierarchical directories).

Finally, you may encounter references to MPE V, which is the operating system for HP 3000s not based on PA-RISC architecture. MPE V software can be run on the PA-RISC (Series 900) HP 3000s in what is known as *compatibility mode*.

The *Berkeley Sockets/iX Reference Manual* is written for experienced application programmers:

- If you are an MPE/iX-based application programmer, you may want to develop new applications for the HP 3000 that can be used on other platforms as well.
- If you are a UNIX-based application programmer, you may want to port existing C-based applications to the HP 3000 hardware platform.

This manual will assist application programmers in porting applications from UNIX-based systems to MPE/iX-based systems.

The BSD sockets product was initially released on MPE/iX release 4.0. POSIX functions were added for MPE/iX release 4.5. Additional socket functionality has been added in release 5.0.

Contents

Tables

1

Introduction

References Used

Several items referenced here are HP-UX man pages, listed in the format "name(#)", as per the UNIX convention. The number given indicates the chapter of the HP-UX man pages where that given page can be found.

- af_ccitt(7F)
- byteorder(3N)
- creat(2)
- dup(2)
- exec(2)
- hosts(4)
- ifconfig(1M)
- inet(7F)
- networks(4)
- open(2)
- pipe(2)
- protocols(4)
- resolver(3N)
- services(4)
- signal(5)
- sigvector(2)
- tcp(7P)
- udp(7P)

Include Files Used

The following table shows the include files used for HP-UX and MPE/iX applications.

Table 1-1.
Include Files Used for HP-UX and MPE/iX Applications

HP-UX Name	MPE/iX Name
<sys/types.h>	types.h.sys
<sys/socket.h>	socket.h.sys
<sys/un.h>	un.h.sys
<sys/file.h>	file.h.sys
<sys/errno.h>	errno.h.sys
<sys/ioctl.h>	ioctl.h.sys
<netinet/in.h>	in.h.sys
<netinet/tcp.h>	tcp.h.sys
<unistd.h>	unistd.h.sys
<fcntl.h>	fcntl.h.sys
<time.h>	time.h.sys
<uio.h>	uio.h.sys
<netdb.h>	netdb.h.sys

The name service routines and BSD socket routines are stored in a native mode relocatable library file name `SOCKETRL.NET.SYS`. When linking your programs, you should include this file in the link list. For example,

```
link objfile,progfile;rl=socketrl.net.sys,libc.lib.sys
```

Ensure that you link with the POSIX library (`/lib/libc.a`) instead of `libc.lib.sys` for POSIX programs. Note that if you are using the POSIX library you must use file indirection as shown in the programming example in Chapter 5.

POSIX Function Support

The following POSIX functions are supported as of MPE/iX release 4.5:

* `close()`
* `dup()`
* `exec()`
* `fork()`
* `read()`
* `write()`

These functions are located in relocatable libraries available through the purchase of the MPE/iX Developer's Kit. For more information about these functions, refer to the *MPE/iX Developer's Kit Reference Manual, Volume 1*.

2

Socket System Calls

This section describes the socket system calls available on MPE/iX. Differences from UNIX 4.3 BSD and limitations are noted.

SOCKET

C Interface

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

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

Description

The `socket` call creates an endpoint for communication and returns a descriptor. The socket descriptor returned is used in all subsequent socket-related system calls.he *af* parameter specifies an address family to be used to interpret addresses in later operations that specify the socket. These address families are defined in the include file `<sys/socket.h>`. The only currently supported address families are as follows:

```
AF_INET  (DARPA Internet addresses)
AF_UNIX  (directory path names on a local node)
```

Note

If you do not have a supported networking link product installed on your system and you attempt to use the address family `AF_INET`, then the `EAFNOSUPPORT` error is returned.

The `AF_UNIX` address family can be used to create socket connections without requiring a networking link product to be installed.

The *type* parameter specifies the communication semantics for the socket. Currently defined types are as follows:

```
SOCK_STREAM
SOCK_DGRAM
```

A `SOCK_STREAM` type provides sequenced, reliable, two-way, connection-based byte streams. A `SOCK_DGRAM` socket supports datagrams, which are connection-less, unreliable messages of a fixed, typically small, maximum length.

The *protocol* parameter specifies a particular protocol to be used with the socket. The protocol number to use depends on the communication domain in which communication is to take place. (Refer to the chapter on name services routines.) *Protocol* can be supplied as zero, in which case the system chooses a protocol type to use, based on the socket type.

Sockets of type `SOCK_STREAM` are byte streams similar to UNIX pipes, except that they are full-duplex instead of half-duplex. 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` or `accept` call. Once connected, data can be transferred using `send` and `recv` calls or `read` and `write` calls. When a session has been completed, a `close` can be performed.

The communications protocol (TCP) used to implement SOCK_STREAM for AF_INET sockets, ensures that data is not lost or duplicated. If a peer has buffer space for data and the data cannot be successfully transmitted within a reasonable length of time, the connection is considered broken and the next recv call indicates an error with *errno* set to ETIMEDOUT. An end-of-file condition (zero bytes read) is returned if a process tries to read on a broken stream. To use the *errno* global variable, include the file <sys/errno.h>.

SOCK_DGRAM sockets allow sending of messages to correspondents named in sendto calls. It is also possible to receive messages at a SOCK_DGRAM socket with recvfrom. The sockets operation is controlled by socket level options set by the setsockopt system call. (Refer to getsockopt or setsockopt.) These options are defined in the file <sys/socket.h>.

Return Value

If the call is successful, a valid file descriptor referencing the socket is returned. If it fails, a -1 is returned, and an error code is stored in *errno*.

Errors

The following errors are returned by socket:

[EHOSTDOWN]	The networking subsystem has not been started.
[EAFNOSUPPORT]	The specified address family is not supported on this version of the system.
[ESOCKTNOSUPPORT]	The specified socket type is not supported in this address family.
[EPROTONOSUPPORT]	The specified protocol is not supported.
[EMFILE]	The per-process descriptor table is full.
[ENOBUFS]	No buffer space is available. The socket cannot be created.
[ENFILE]	The system's table of open files is temporarily full, and no more socket calls can be accepted.
[EPROTOTYPE]	The type of socket and protocol do not match.
[ETIMEDOUT]	The connection timed out.

MPE/iX Specific

Break mode is supported on MPE/iX. This is true of all Berkeley Sockets/iX system calls described in this section.

Author

UCB (University of California at Berkeley)

BIND

C Interface

AF_UNIX only:

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

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

AF_INET only:

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

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

Description

The `bind` system call assigns an address to an unbound socket. When a socket is created with `socket`, it exists in an address space (address family) but has no address assigned. The `bind` call causes the socket whose descriptor is *s* to become bound to the address specified in the socket address structure pointed to by *addr*. The *addrlen* parameter must specify the size of the address structure. Since the size of the socket address structure varies between socket address families (16 bytes for `AF_INET`; 110 bytes for `AF_UNIX`), the correct socket address structure should be used with each address family (`struct sockaddr_in` for `AF_INET`; `struct sockaddr_un` for `AF_UNIX`).

Here is the socket address structure for AF_INET, extracted from the IN.H.SYS file:

```
struct in_addr {
    union {
        struct { u_char s_b1,s_b2,s_b3,s_b4; } s_un_b;
        struct { u_short s_w1,s_w2; } S_un_w;
        u_long S_addr;
    } S_un;
#define s_addr  S_un.S_addr /* can be used for most tcp & ip code
*/
#define s_host  S_un.S_un_b.s_b2     /* host on imp  */
#define s_net   S_un.S_un_b.s_b1     /* network      */
#define s_imp   S_un.S_un_w.s_w2     /* imp          */
#define s_impno S_un.S_un_b.s_b4     /* imp #        */
#define s_lh    S_un.S_un_b.s_b3     /* logical host */
};

#define      INADDR_ANY        (u_long)0x00000000
#define      INADDR_THISHOST   (u_long)0x00000000

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

Here is the socket address structure for AF_UNIX, extracted from the UN.H.SYS file:

```
struct  sockaddr_un {
        short   sun_family;         /* AF_UNIX         */
        char    sun_path[108];      /* path name       */
};
```

Examples

Here is an example program to create and bind an AF_UNIX socket:

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

main ()
{
int s;
int af, type, protocol;
struct sockaddr_un addr;
int addrlen;

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

```
addr.sun_family = AF_UNIX;
strcpy (addr.sun_path,"tmp/socket");
addrlen = 110;

bind (s,addr, addrlen);

}
```

Return Value

If the `bind` is successful, a 0 value is returned. If it fails, -1 is returned, and an error code is stored in *errno*.

Errors

The following errors are returned by `bind`:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is not a socket.
[EADDRNOTAVAIL]	The specified address is bad or is not available from the local machine.
[EADDRINUSE]	The specified address is already in use.
[EINVAL]	The socket is already bound to an address, the socket has been shut down or *addrlen* is a bad value.
[EAFNOSUPPORT]	The requested address does not match the address family of this socket.
[EACCES]	The requested address is protected, and the current user has inadequate permission to access it. (This error can be returned by `AF_INET` only.)
[EFAULT]	The *addr* parameter is not a valid pointer.
[EOPNOTSUPP]	The socket whose descriptor is *s* is of a type that does not support address binding.
[ENOBUFS]	Insufficient buffer memory is available. The `bind` cannot complete.
[EDESTADDREQ]	No *addr* parameter was specified.
[ENETUNREACH]	The network is not reachable from this host.

MPE/iX Specific

On HP-UX, when binding an `AF_UNIX` socket to a path name (such as `/tmp/mysocket`), an open file having that name is created in the file system. When the bound socket is closed, that file still exists unless it is removed or unlinked. This does not occur on MPE/iX (that is, no file is created).

On HP-UX, you are allowed to specify a specific network while binding. MPE/iX does not. The IP address portion of *sockaddr* for `AF_INET` must be zero.

Author

UCB (University of California at Berkeley)

See Also

`connect`, `listen`, `socket`, `getsockname`, Name Service Routines

LISTEN

C Interface

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

Description

To accept connections, a socket is first created with `socket`, a queue for incoming connections is specified with `listen`, and then connections are accepted with `accept`. The `listen` call applies only to unconnected sockets of type `SOCK_STREAM`. Note that you cannot call `listen` after `accept` has been called. If the socket has not been bound to a local port before the `listen` is invoked, the system automatically binds a local port for the socket to listen on.

The listen queue is established for the socket specified by the *s* parameter, which is a socket descriptor.

The *backlog* parameter defines the maximum allowable length of the queue for pending connections. If a connection request arrives when the queue is full, the client receives an `ETIMEDOUT` error.

The *backlog* parameter is limited (silently) to be in the range of 1 to 128. If you specify any other value, the system automatically assigns the closest value within range.

Return Value

If the call is successful, 0 is returned. If the call fails, a -1 is returned, and an error code is stored in *errno*.

Errors

The following errors are returned by `listen`:

[EBADF]	The argument *s* is not a valid descriptor.
[EDESTADDRREQ]	No `bind` address was established.
[ENOTSOCK]	The argument *s* is not a socket.
[EOPNOTSUPP]	The socket is not of a type that supports the `listen` operation.
[ENOBUFS]	Series 300 only: No buffer space is available. The `listen` call cannot be started at this time.
[EINVAL]	The socket has been shut down or is already connected.

MPE/iX Specific The backlog limit on MPE/iX is 128 as opposed to the backlog limit of 20 on HP-UX. When an HP-UX socket has performed a `listen`, the incoming connection requests are completed as they are received (up to the *backlog* limit). When using MPE/iX, connections are completed by the call to `accept`.

Author UCB (University of California at Berkeley)

See Also `accept`, `connect`, `socket`

ACCEPT

C Interface

AF_UNIX only:

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

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

AF_INET only:

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

int ns;
ns=accept(s, addr, addrlen)
int s;
struct sockaddr_in *addr;
int *addrlen;
```

Description

This call is used with connection-based socket types, such as `SOCK_STREAM`. The argument *s* is a socket descriptor created with `socket`, bound to an address with `bind`, and listening for connections after a `listen`. If pending connections are present on the queue, the `accept` call extracts the first connection on the queue, creates a new socket, and allocates a new file descriptor, *ns*, for the socket. If no pending connections are present on the queue and non-blocking mode has been enabled using the O_NONBLOCK or O_NDELAY fcntl flags (refer to `fcntl` section), `accept` returns an error as described in the Errors section below. If no pending connections are present on the queue and non-blocking mode has not been enabled, `accept` blocks the caller until a connection is present. The accepted socket, *ns*, cannot be used to accept more connections. The original socket*s* remains open. It is possible to determine if a listening socket has pending connection requests ready for an `accept` call by using `select` for reading.

The argument *addr* should point to a local socket address structure. The `accept` call fills in this structure with the address of the connecting entity, as known to the underlying protocol. The format of the address depends upon the protocol and the address family of the socket *s*. The *addrlen* is a pointer to `int`; it should initially contain the size of the structure pointed to by *addr*. On return it

contains the actual length (in bytes) of the address returned. If the memory pointed to by *addr* is not large enough to contain the entire address, only the first *addrlen* bytes of the address are returned.

`AF_UNIX` only: The *addr* parameter to `accept()` is ignored.

Return Value

If the call is successful, a non-negative integer is returned, which is a descriptor for the accepted socket. If the call fails, a -1 is returned and an error code is stored in *errno*.

Errors

The following errors are returned by `accept`:

[EHOSTDOWN]	The networking subsystem has not been started or has been stopped.
[EBADF]	The file descriptor *s* is invalid.
[ENOTSOCK]	The argument *s* references a file, not a socket.
[EOPNOTSUPP]	The socket referenced by *s* is not of type `SOCK_STREAM`.
[EFAULT]	The *addr* parameter is not a valid pointer.
[EWOULDBLOCK]	Non-blocking I/O is enabled using `O_NDELAY`, and no connections are present to be accepted.
[EMFILE]	The maximum number of file descriptors for this process are already currently open.
[ENFILE]	The system's table of open files is full, and no more `accept` calls can be accepted, temporarily.
[ENOBUFS]	No buffer space is available. The `accept` cannot complete. The queued socket `connect` request is aborted.
[EINVAL]	The socket referenced by *s* is not currently a listen socket, or it has been shut down. A `listen` must be done before an `accept` is allowed. This is also returned if the length is less than zero.
[EINTR]	The call was interrupted by a signal before a valid connection arrived.
[EAGAIN]	Non-blocking I/O is enabled using `O_NONBLOCK`, and no connections are present to be accepted.

MPE/iX Specific Connections are completely established in the `accept` call. The *addr* returned from `accept` when a connection is made in loopback is the loopback address (127.0.0.1). On HP-UX, the local host's IP address would be returned in this case.

Author UCB (University of California at Berkeley)

See Also `bind`, `connect`, `listen`, `select`, `socket`

CONNECT

C Interface

AF_UNIX sockets only:

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

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

AF_INET sockets only:

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

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

Description

The parameter *s* is a socket descriptor. The *addr* parameter is a pointer to a socket address structure. This structure contains the address of a remote socket to which a connection is established. The *addrlen* parameter specifies the size of this address structure. Since the size of the socket address structure varies between socket address families (16 bytes for `AF_INET`; 110 bytes for `AF_UNIX`), the correct socket address structure should be used with each address family (`struct sockaddr_in` for `AF_INET`, `struct sockaddr_un` for `AF_UNIX`).

If the socket is of type `SOCK_STREAM`, then `connect` attempts to contact the remote host in order to make a connection between the remote socket (peer) and the local socket specified by *s*. The call normally blocks until the connection completes. If non-blocking mode has been enabled using the `O_NONBLOCK` or `O_NDELAY` `fcntl` flags and the connection cannot be completed immediately, then `connect` returns an error as described below. In these cases, the `select` call can be used on this socket to determine when the connection has completed by selecting it for writing.

The `O_NONBLOCK` and `O_NDELAY` flags are defined in `<fcntl.h>` and are explained in the `fcntl` section. If *s* is a `SOCK_STREAM` socket that is bound to the same local address as another `SOCK_STREAM` socket and *addr* is the same as the peer address of the other socket, `connect` returns `EADDRINUSE`.

If the `AF_INET` socket does not already have a local address bound to it (refer to the `bind` call), the `connect` call also binds the socket to a local address chosen by the system.

Stream sockets may successfully connect only once.

Return Value

If the call is successful, a 0 is returned. If it fails, a -1 is returned, and an error code is stored in *errno*.

Errors

The following errors are returned by `connect`:

[EBADF]	The argument *s* is not a valid file descriptor.
[ENOTSOCK]	The argument *s* is a file descriptor for a file, not a socket.
[EADDRNOTAVAIL]	The specified address is not available on this machine, or the socket is a `tcp` or `udp` socket and the zero port number is specified.
[EAFNOSUPPORT]	Addresses in the specified address family cannot be used with this socket. For datagram sockets, the peer address is no longer maintained by the system.
[EISCONN]	The socket is already connected.
[EINVAL]	The socket has already been shut down or has a `listen` active on it, or *addrlen* is a bad value.
[ETIMEDOUT]	Connection establishment timed out without establishing a connection. The *backlog* parameter may be full. (Refer to the `listen` call.)
[ECONNREFUSED]	The attempt to connect was forcefully rejected.
[ENETUNREACH]	The network is not reachable from this host.
[EADDRINUSE]	The address is already in use.
[EFAULT]	The *addr* parameter is not a valid pointer.
[EINPROGRESS]	Non-blocking I/O is enabled using `O_NONBLOCK` or `O_NDELAY`, and the connection cannot be completed immediately. This is not a failure.
[ENOSPC]	All available virtual circuits are in use.
[EOPNOTSUPP]	A `connect` attempt was made on a socket type that does not support this call.
[EINTR]	The call was interrupted by a signal before a valid connection arrived.

MPE/iX Specific The `connect` call is not supported for `SOCK_DGRAM` sockets on the current release.

Author UCB (University of California at Berkeley)

See Also `accept`, `select`, `socket`, `getsockname`, `fcntl`

SOCKETPAIR

C Interface

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

socketpair(d, type, protocol, sv)
int d, type, protocol;
int sv[2];
```

Description

The `socketpair` call creates an unnamed pair of connected sockets in the specified domain *d*, of the specified *type*, and using the optionally specified *protocol*. The descriptors used in referencing the new sockets are returned in *sv[0]* and *sv[1]*. The two sockets are indistinguishable.

This intrinsic is implemented for `AF_UNIX` sockets only.

Return Value

If the call is successful, a 0 is returned. If the call fails, a -1 is returned and an error code is stored in *errno*.

Errors

The following errors are returned by `socketpair`:

[EMFILE]	Too many descriptors are in use by this process.
[EAFNOSUPPORT]	The specified address family is not supported on this machine.
[EPROTONOSUPPORT]	The specified protocol is not supported on this machine.
[EOPNOSUPPORT]	The specified protocol does not support creation of socket pairs.
[EFAULT]	The address *sv* does not specify a valid part of the process address space.

See Also

`read`, `write`

SEND

C Interface

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

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

sendmsg(s, msg, flags)
int s;
struct msghdr msg[];
int flags;
```

AF_UNIX only:

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

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

AF_INET only:

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

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

Description

The `send`, `sendto`, and `sendmsg` calls are used to transmit a message to another socket.

The argument *s* is a socket descriptor that specifies the socket on which the message is sent. The *msg* parameter points to the buffer that contains the message.

If the socket uses connection-based communications (for example, a `SOCK_STREAM` socket), then these calls can be used only after the connection has been established. (Refer to `connect`.) In this case, any destination specified by the *to* parameter is ignored. For connection-less sockets (for example, `SOCK_DGRAM`), the *to* parameter must be used.

The address of the target is contained in a socket address structure pointed at by *to*, with the *tolen* parameter specifying the size of the structure. If the address specified in the argument is a broadcast address, the `SO_BROADCAST` option must be set for broadcasting to succeed.

If a `sendto` is attempted on a `SOCK_DGRAM` socket before any local address has been bound to it, the system automatically selects a local address to be used for the message (`AF_INET` only). In this case, there is no guarantee that the same local address is used for successive `sendto` requests on the same socket.

The length of the message is given by *len*. The length of data actually sent is returned. If the message is too long to pass atomically through the underlying protocol, the message is not transmitted, -1 is returned, and *errno* is set to `EMSGSIZE`. For `SOCK_DGRAM` and `SOCK_STREAM` sockets, this size is fixed by the implementation. (Refer to "MPE/iX Specific" section below.)

No indication of failure to deliver is implicit in a `send`, `sendto`, or `sendmsg`. Return values of -1 indicate some locally detected errors.

If no buffer space is available to hold the data to be transmitted, `send` blocks unless non-blocking mode is enabled. There are two ways to enable non-blocking mode: with the `O_NONBLOCK` `fcntl` flag and with the `O_NDELAY` `fcntl` flag.

If `O_NONBLOCK` is set using `fcntl` (defined in `<fcntl.h.sys>` and explained in the `fcntl` section), POSIX-style non-blocking I/O is enabled. In this case, the `send` request completes in one of three ways:

- If there is enough space available in the system to buffer all of the data, the `send` completes successfully, having written out all of the data and returns the number of bytes written.
- If there is not enough space in the buffer to write out the entire request, the `send` completes successfully, having written as much data as possible, and returns the number of bytes that it was able to write.

- If there is no space in the system to buffer any of the data, the `send` completes successfully, having written no data, and returns a -1 with *errno* set to `EAGAIN`.

  If `O_NDELAY` is set using `fcntl` (defined in `<fcntl.h.sys>` and explained in the `fcntl` section), non-blocking I/O is enabled. In this case, the `send` request completes in one of three ways:

- If there is enough space available in the system to buffer all of the data, the `send` completes successfully, having written out all of the data, and returns the number of bytes written.
- If there is not enough space in the buffer to write out the entire request, the `send` completes successfully, having written as much data as possible, and returns the number of bytes that it was able to write.
- If there is no space in the system to buffer any of the data, the `send` completes successfully, having written no data, and returns 0.

If the `O_NDELAY` and `O_NONBLOCK` flags are cleared using `fcntl`, non-blocking I/O is disabled. In this case, the `send` always executes completely (blocking as necessary) and returns the number of bytes written.

If the available buffer space is not large enough for the entire message, the `EWOULDBLOCK` error is returned.

To summarize, both behave the same if there is enough space to write all of the data or even some of the data. They differ in the third case, where there is not enough space to write any of the data.

The `select` call can be used to determine when it is possible to send more data.

The supported values for *flags* are zero. A `write()` call made to a socket behaves the same way as `send` with *flags* set to zero.

See `recv` for a description of the *msghdr* structure for `sendmsg`.

Return Value

If successful, the call returns the number of characters sent. If the call fails, a -1 is returned, and an error code is stored in *errno*.

Errors

The following errors are returned by `send`, `sendto`, or `sendmsg`:

[EACCES]	Process doing a `send` of a broadcast packet is not privileged.
[EBADF]	An invalid descriptor was specified.
[ENOTSOCK]	The argument *s* is not a socket.
[EFAULT]	The *msg* or *to* parameter is not a valid pointer.

[EMSGSIZE]	The socket requires that messages be sent atomically, and the size of the message to be sent made this impossible.
[EWOULDBLOCK]	The socket is in non-blocking mode, and the requested operation would block.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[EINVAL]	The *len* or *tolen* parameter contains a bad value.
[EDESTADDRREQ]	The *to* parameter needs to specify a destination address for the message. This is also given if the specified address contains unspecified fields. (Refer to the `inet` section.)
[ENOTCONN]	The `send` on a socket has not connected, or a `send` on a socket did not complete the connect sequence with its peer or is no longer connected to its peer.
[EAFNOSUPPORT]	Requested address does not match the address family of this socket.
[EPIPE]	An attempt was made to `send` on a socket that was connected, but the connection has been shut down either by the remote peer or by this side of the connection.
[EOPNOTSUPP]	The `MSG_OOB` flag was specified; it is not supported for `AF_UNIX` sockets.
[EINTR]	The call was interrupted by a signal before a valid connection arrived.

MPE/iX Specific

The maximum `udp` message size that can be sent on a `SOCK_DGRAM` socket is 30,000 bytes on MPE/iX, as opposed to 9,216 bytes on HP-UX. For `SOCK_STREAM`, there is also a maximum message size of 30,000 bytes on MPE/iX. For `AF_UNIX`, there is a maximum window size of 30,000 bytes.

The `sendto` call can be used only with `SOCK_DGRAM` sockets.

The `send` call can be used only with `SOCK_STREAM` sockets.

The `flags` parameter must be zero.

Author

UCB (University of California at Berkeley)

See Also

`getsockopt`, `recv`, `select`, `socket`

RECV

C Interface

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

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

recvmsg(s, msg, flags)
int s;
struct msghdr msg[];
int flags;
```

AF_UNIX only:

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

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

AF_INET only:

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

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

Description

The `recv`, `recvfrom`, and `recvmsg` calls are used to receive messages from a socket.

The argument *s* is a socket descriptor from which the message is received. The *buf* parameter is a pointer to the buffer into which the messages are placed. The *len* parameter is the maximum number of bytes that will fit into the buffer referenced by *buf*.

If the socket uses connection-based communications (for example, a `SOCK_STREAM` socket), then these calls can be used only after the connection has been established. (Refer to `connect`.) For connection-less sockets (for example, a `SOCK_DGRAM` socket), these calls can be used whether a connection has been established or not.

The `recvfrom` call operates the same as the `recv` call does, except that it is able to return the address of the socket from which the message was sent. If *from* is non-zero, the source address of the message is placed into the socket address structure pointed to by *from*. The *fromlen* parameter is a value-result parameter, initialized to the size of the structure associated with *from*, and modified on return to indicate the actual size of the address stored there. If the memory pointed to by *from* is not large enough to contain the entire address, only the first *fromlen* bytes of the address are returned.

The length of the message is the functional return.

For message-based sockets like `SOCK_DGRAM`, the entire message must be read in one operation. If a message is too long to fit in the supplied buffer, the excess bytes are discarded. For stream-based sockets like `SOCK_STREAM`, there is no concept of message boundaries. In this case, data is returned to the user as soon as it becomes available, and no data is discarded.

If no data is available to be received, `recv` waits for a message to arrive unless non-blocking mode is enabled. There are two ways to enable non-blocking mode: with the `O_NONBLOCK` `fcntl` flag, and with the `O_NDELAY` `fcntl` flag.

If `O_NONBLOCK` is set using `sfcntl` (defined in `<fcntl.h.sys>` and explained in the `fcntl` section), POSIX-style non-blocking I/O is enabled. In this case, the `recv` request completes in one of three ways:

- If there is enough data available to satisfy the entire request, `recv` completes successfully, having read all of the data, and returns the number of bytes read.
- If there is not enough data available to satisfy the entire request, `recv` completes successfully, having read as much data as possible, and returns the number of bytes that it was able to read.
- If there is no data available, `recv` completes successfully, having read no data, and returns a -1 with *errno* set to `EAGAIN`.

If `O_NDELAY` is set using `sfcntl` (defined in `<fcntl.h.sys>` and explained in the `fcntl` section), non-blocking I/O is enabled. In this case, the `recv` request completes in one of three ways:

- If there is enough data available to satisfy the entire request, `recv` completes successfully, having read all of the data, and returns the number of bytes read.
- If there is not enough data available to satisfy the entire request, `recv` completes successfully, having read as much data as possible, and returns the number of bytes that it was able to read.
- If there is no data available, `recv` completes successfully, having read no data, and returns a 0.

If `O_NONBLOCK` or `O_NDELAY` is cleared using `sfcntl`, the corresponding style of non-blocking I/O, if previously enabled, is disabled. In this case, `recv` always executes completely (blocking as necessary) and returns the number of bytes read.

To summarize, both behave the same if there is enough data available to satisfy the entire request or even part of the request. They differ only in the third case, where there is no data available.

The `select` call can be used to determine when more data arrives by selecting the socket for reading.

The *flags* parameter can be set to `MSG_PEEK` or zero. If it is set to `MSG_PEEK`, any data returned to the user is treated as if it had not been read. The next `recv` rereads the same data. The value is as follows:

```
#define  MSG_PEEK  0x2  /* peek at incoming message */
```

A `read` call behaves the same way as a `recv` call with *flags* set to zero.

The `recv|msg` call uses a *msghdr* structure to minimize the number of directly supplied parameters. This structure has the following form, as defined in `<sys/socket.h>`:

```
struct msghdr {
       caddr_t msg_name;         /* optional address            */
       int     msg_namelen;      /* size of address             */
       struct  iov *msg_iov;     /* scatter/gather array        */
       int     msg_iovlen;       /* # elements in msg_iov       */
       caddr_t msg_accrights;    /* access rights sent/received */
       int     msg_accrightslen;};
```

Here, *msg_name* and *msg_namelen* specify the destination address if the socket is unconnected, and *msg_name* can be given as a null pointer if no names are desired or required. The *msg_iov* and *msg_iovlen* describe the scatter gather locations, as described in `read`. Access rights to be sent with the message are specified in *msg_accrights*, which has *msg_accrightslen*.

Return Value

If the call is successful, it returns the number of bytes received. If the call fails, a -1 is returned, and an error code is stored in *errno*. A zero is returned if the socket is blocking and the transport connection to the remote node fails.

Errors

The following errors are returned by `recv`, `recvfrom`, or `recvmsg`:

[EBADF]	The argument *s* is an invalid descriptor.
[ENOTSOCK]	The argument *s* is not a socket.
[EWOULDBLOCK]	The socket is marked non-blocking, and the receive operation would block.
[EFAULT]	The *buf*, *from*, or *fromlen* parameters are not valid pointers.
[ETIMEDOUT]	The connection timed out during connection establishment, or due to a transmission timeout on an active connection.
[ENOTCONN]	Receive on a `SOCK_STREAM` socket that is not yet connected.
[EINVAL]	The *len* parameter is bad or there is no data available on a receive of out-of-band data.
[EINTR]	The call was interrupted by a signal before a valid connection arrived.
[EOPNOTSUPP]	A `receive` was attempted on a SOCK_DGRAM socket that has not been bound. A `bind` should be done before the `receive`.
[ENOBUFS]	Insufficient buffer memory is available.

MPE/iX Specific

For `AF_UNIX`, `recvfrom()` is supported; however, the *from* and *fromlen* parameters are ignored (that is, it works just like `recv()`).

The `recv` call can only be done on `SOCK_STREAM` sockets.

The `recvfrom` call can only be done on `SOCK_DGRAM` sockets.

The `MSG_OOB` flag, used for processing out-of-band data, is not supported on MPE/iX.

Author

UCB (University of California at Berkeley)

See Also

`read`, `select`, `send`, `socket`

SHUTDOWN

C Interface

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

Description

The `shutdown` system call is used to shut down a socket. The *s* parameter is the socket descriptor of the socket to be shut down. In the case of a full-duplex connection, `shutdown` can be used to either partially or fully shut down the socket, depending on the value of *how*.

If *how=0*, the socket can still send data, but it cannot receive data. Remaining data can be sent and new data can be sent; however, all further `recv()` calls return an end-of-file condition.

If *how*=1, further sends by the user will return an EPIPE error. A SIGPIPE signal will be sent to the user unless the user has used `ioctl()` to ignore the signal. Note that data already queued by a previous `send` call will still be sent.

If *how*=2, a socket cannot send remaining or new data or receive data. This is the same as doing a shutdown of 0 and a shutdown of 1 simultaneously.

Once the socket has been shut down for receives, all further `recv` calls return an end-of-file condition.

A `shutdown` on a connection-less socket, such as `SOCK_DGRAM`, only marks the socket unable to do further sends or receives, depending on *how*. Once this type of socket has been disabled for both sending and receiving data, it becomes fully shut down.

For `SOCK_STREAM` sockets, if *how* is 1 or 2, the connection begins a graceful disconnect. The disconnection is complete when both sides of the connection have done a `shutdown` with *how* equal to 1 or 2. Once the connection has been completely terminated, the socket becomes fully shut down.

Note the difference between the `close` and `shutdown` calls. `Close` makes the socket descriptor invalid while `shutdown` is used to partially or fully shutdown the I/O on the socket. The user can call `close` after a shutdown to make the socket descriptor unusable. The SO_LINGER option does not have any meaning for the `shutdown` call, but does for the `close` call. (Refer to `setsockopt`.)

Return Value

If the call is successful, a 0 is returned. If it fails, a -1 is returned, and an error code is stored in *errno*.

Errors

The following errors are returned by `shutdown`:

[EBADF] The argument*s* is not a valid descriptor.

[ENOTSOCK] The argument*s* is a file, not a socket.

[EINVAL] The specified socket is not connected.

Author

UCB (University of California at Berkeley)

See Also

`close`, `connect`, `socket`

GETSOCKOPT, SETSOCKOPT

C Interface

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

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

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

Description

`getsockopt` and `setsockopt` manipulate options associated with a socket. The socket is identified by the socket descriptor *s*. Options can exist at multiple protocol levels. Options are described below under the appropriate option.

When manipulating socket options, the level at which the option resides (*level*) and the name of the option (*optname*) must be specified. To manipulate options at the "socket" level, *level* is specified as `SOL_SOCKET`.

There are two kinds of options: boolean and non-boolean. Boolean options are either set or not set and also can use *optval* and *optlen* (see below) to pass information. Non-boolean options always use *optval* and *optlen* to pass information.

To determine whether or not a boolean option is set, the return value of `getsockopt` must be examined. If the option is set, `getsockopt` returns without error. If the boolean option is not set, `getsockopt` returns -1 and *errno* is set to `ENOPROTOOPT`.

For `setsockopt`, the parameters *optval* and *optlen* are used to pass option information from the calling process to the system. *optval* is the address of a location in memory that contains the option information to be passed to the system. *optlen* is an integer that specifies the size in bytes of the option information.

For `getsockopt`, *optval* and *optlen* are used to pass option information from the system to the calling process. *optval* is the address of a location in memory that contains the option information to be passed to the calling process, or (char *) NULL if the option information is not of interest and not to be passed to the calling process. *optlen* is an address of an integer initially used to specify

the maximum number of bytes of option information to be passed. If *optval* is not (char *) NULL, *optlen* is set on return to the actual number of bytes of option information passed. If the `getsockopt` call fails, no option information is passed.

optname and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The include file `<sys/socket.h>` contains definitions for "socket" level options. Options at other protocol levels vary in format and name.

The "socket" level options defined in the include file `<sys/socket.h>` are explained below.

SOL_SOCKETS

When the socket level is `SOL_SOCKETS`, the following options are available:

SO_DEBUG	(Boolean option) No functionality; included only for compatibility.
SO_KEEPALIVE	(Boolean option; `AF_INET SOCK_STREAM` sockets only) Keeps otherwise idle connected sockets active by forcing transmissions every 600 seconds for up to four retransmissions without a response. The length of time and number of retransmissions are configurable in NMMGR. Default: Off
SO_LINGER	(Boolean option; `AF_INET SOCK_STREAM` sockets only) Lingers on close if data is present. For `SO_LINGER`, *optval* points to a struct linger, defined in `<sys/socket.h>`. The linger structure contains an integer boolean flag to toggle behavior on/off and an integer linger value. Default: Off
SO_BROADCAST	(Boolean option; `AF_INET SOCK_DGRAM` sockets only) Toggles permission to transmit broadcast messages. Default: Off
SO_ERROR	Returns to the caller the last error stored in the socket record. This error variable is then cleared in the socket record.
SO_TYPE	Returns the socket type. This option is typically used by a process that inherits a socket when it is started.

`SO_LINGER` controls the actions taken when unsent messages are queued on a SOCK_STREAM socket and a `close` is performed. If `SO_LINGER` is toggled on with a non-zero linger interval, the system

blocks the process on the `close` attempt until it is able to transmit the data or until it decides it is unable to deliver the information. If `SO_LINGER` is toggled on with a linger interval of zero, the connection is immediately terminated on the `close` of the socket, and any unsent data queued on the connection is lost. If `SO_LINGER` is toggled off (default upon socket creation) and a `close` is issued, the call returns immediately. The system still gracefully brings down the connection by transmitting any queued data, if possible. `SO_LINGER` can be toggled on/off at any time during the life of an established connection. Toggling `SO_LINGER` does not affect the action of `shutdown`.

The SO_BROADCAST option requests permission to send internet broadcast datagrams on the socket.

The ip level option defined in the include file `<netinet/in.h>` is explained below.

IPPROTO_IP

When the socket level is `IPPROTO_IP`, the following option is available:

IP_OPTIONS	(`AF_INET SOCK_DGRAM`) Allows you to set and retrieve direct IP header information.

IPPROTO_UDP

No options are defined for this level.

The tcp level option defined in the include file `<netinet/tcp.h>` is described below.

IPPROTO_TCP

When the socket level is `IPPROTO_TCP`, the following option is available:

TCP_MAXSEG	(`AF_INET SOCK_STREAM`) Returns the maximum segment size in use for the socket. It defaults the size to 536 until the socket is connected. The size is negotiated during connection establishment.

Return Value

If the call is successful, 0 is returned. If it fails, -1 is returned and an error code is stored in *errno*.

Errors

The call to `getsockopt` or `setsockopt` fails if:

[EBADF]	The argument *s* is not a valid descriptor.
[EOPNOTSUPP]	The option is not supported by the protocol in use by the socket.
[ENOBUFS]	No buffer space is available.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOPROTOOPT]	In `getsockopt`, the requested option is currently not set.

[EINVAL]	The option is unknown at the socket level or the socket has been shut down.
[EFAULT]	The *optval* or, in the case of `getsockopt`, *optlen* parameters are not valid pointers.
[ESOCKTNOSUPPORT]	The socket is a NetIPC socket.

Author UCB (University of California at Berkeley)

See Also `socket, getprotoent`

GETPEERNAME

C Interface

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

getpeername(s, addr, addrlen)
int s;
struct sockaddr_in *addr;
int *addrlen;
```

Description

S is a socket descriptor. The `getpeername` system call returns the address of the peer socket connected to the socket indicated by *s*. The *addr* parameter points to a socket address structure in which this address is returned. The *addrlen* parameter points to an object of the type *int*, which should be initialized to indicate the size of the address structure. On return, the *addrlen* parameter contains the actual size of the address returned (in bytes). If *addr* does not point to enough space to contain the whole address of the peer, only the first *addrlen* bytes of the address are returned.

This call is supported for `AF_INET` only.

Return Value

If the call is successful, a 0 is returned. If the call fails, a -1 is returned, and an error code is stored in *errno*.

Errors

The following errors are returned by `getpeername`:

[EBADF]	The argument *s* is not a valid file descriptor.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOTCONN]	The socket is not connected.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[EFAULT]	The *addr* or *addrlen* parameters are not valid pointers.
[EINVAL]	The socket has been shut down.
[EOPNOT-SUPP]	The operation is not supported for `AF_UNIX` sockets.

Author

UCB (University of California at Berkeley)

See Also

`bind`, `socket`, `getsockname`

GETSOCKNAME

C Interface

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

getsockname(s, addr, addrlen)
int s;
struct sockaddr_in *addr;
int *addrlen;
```

Description

The argument *s* is a socket descriptor. The `getsockname` system call returns the address of the socket indicated by *s*. The *addr* parameter points to a socket address structure in which this address is returned. The *addrlen* parameter points to an *int* parameter that should be initialized to indicate the size of the address structure. On return, the *addrlen* parameter contains the actual size of the address returned (in bytes). If *addr* does not point to enough space to contain the whole address of the socket, only the first *addrlen* bytes of the address are returned.

This call is supported for `AF_INET` only.

Return Value

If the call is successful, a 0 is returned. If the call fails, a -1 is returned, and the error code is stored in *errno*.

Errors

The following errors are returned by `getsockname`:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[EFAULT]	The *addr* or *addrlen* parameters are not valid pointers.
[EINVAL]	The socket has been shut down.
[EOPNOT-SUPP]	The operation is not supported for `AF_UNIX` sockets.

Author

UCB (University of California at Berkeley)

See Also

`bind`, `socket`, `getpeername`

GETHOSTNAME

C Interface

```
#include <unistd.h>

int gethostname(hostname, size);
char *hostname;
size_t size;
```

Description

The `gethostname` system call returns the standard host name for the current processor as set by `sethostname`. The *size* parameter specifies the length of the hostname array. The hostname is null-terminated unless insufficient space is provided.

Return Value

If the call is successful, a 0 is returned. If it fails, a -1 is returned, and an error code is stored in *errno*.

Errors

The following error is returned by `gethostname`:

[EFAULT] The hostname points to an illegal address. The reliable detection of this error is implementation dependent.

MPE/iX Specific

This call returns the node name configured in the local domain name field in NMMGR.

Author

UCB (University of California at Berkeley)

Signals and Sockets

What is a Signal?

A signal is the notification to a process of an event. This event can be either external or internal. A signal can be generated by either the operating system or a process (even the same process that is receiving it). Other than which signal is delivered, no information is delivered with the signal. A process cannot determine what process generated the signal, how it was generated or which file descriptor, if any, for the process the signal is related to. When a signal is sent to a process and the process is made aware of that signal, it is said that that signal has been raised.

There are several methods for generating a signal. However, for the purposes of this discussion, how signals are generated is largely immaterial. Suffice it to say that most signals related to sockets are generated by software conditions.

There are many system calls relating to the handling of signals. Please refer to the POSIX manuals for a more complete description.

For the purposes of illustration, we will use the `signal()` system call, which is the simplest of the system calls dealing with sockets, to describe signal handling.

There are several default actions possible that may be performed upon receipt of a signal. Some of them are: ignore the signal, terminate the process, suspend (stop) the process, and resume (continue) the process. Of the signals generated by sockets, only the ignore and terminate defaults are available. Which of these actions is the default for signal reception depends on the signal.

When `signal()` is called, there are three methods for handling the signal. They are SIG_DFL, SIG_IGN or a user specified signal handler. SIG_DFL sets the signal handler to the default signal handler. This is useful if the signal handler was changed and the process should be returned to an *initial* state. SIG_IGN sets the signal handler to ignore this signal.

The third choice can be the most useful. It is used to set up a user specifiable function to be the signal handler. Whenever that signal is raised, the user specified function is invoked. This is generally referred to as catching a signal. A typical example of this type of handler (for a signal that is not indicative of an error state) is to either set or increment a flag indicating that a signal was received.

Sockets and Incoming Signals

When a signal is sent to a process while performing a sockets function, several things may occur. This depends on whether the socket function is defined as a *slow* function. A slow *function* is a function that can block indefinitely. For sockets, these functions are `read`, `write`, `recv`, `send`, `recvfrom`, `recvmsg`, `sendmsg`, and `accept`. All other sockets functions are *fast*.

Fast functions are not interrupted by a signal. Instead, the signal is raised when these socket functions exit.

Slow functions are interrupted by a signal if they are blocked waiting for IO (if they are processing IO, they are not interrupted). They are interrupted in the middle of processing by the raising of a signal. They stop what processing they are doing and return the error `EINTR`. They do not complete the IO that was initiated. The user program must re-initiate any desired IO explicitly.

Signals Generated By Sockets

There are two signals that can be generated by actions on a socket. They are SIGPIPE and SIGIO. A SIGPIPE is generated when a send operation is attempted on a *broken* socket. One way of *breaking* a socket is to do a shutdown(,2) on a socket. The default action is to terminate the process. The target of the signal is the process attempting the send.

The other signal is SIGIO. SIGIO is somewhat more complex than SIGPIPE. First, a call to `ioctl()` to request the enabling FIOASYNC is required to enable generation of this signal. Second, another call to `ioctl()` is required to request SIOCSPGRP, which

sets the target process group. A target process group is either a process (specified by a positive process id) or a process group (specified by a negative process id). A SIGIO signal is generated whenever new IO can complete on a socket. Examples of when a SIGIO signal is generated are when new data arrives at the socket, when data can again be sent on the socket, when the socket is either partially or completely shutdown or when a listen socket has a connection request posted on it.

Using Signals With Sockets

There are several issues to using sockets that can be resolved through the use of signals. One is the fact that there are no timeouts on sockets. A similar functionality can be accomplished with the use of the SIGALRM signal and the `alarm()` system call. `Alarm()` sets up a SIGALRM signal to be received a certain specified number of seconds in the future. If the socket call is not completed by the time the alarm goes off, it is interrupted. The alarm is reset by another call to `alarm()` with a time of 0 seconds.

Another issue that may arise if the use of signals to indicate when IO can complete. Use of the SIGIO signal can be used to indicate when IO can complete. After enabling the SIGIO signal on all of the desired descriptors, `pause()` is called to wait for a signal to arrive. Then, either using a polling method or `select()`, it can be determined which socket has IO ready to complete. This avoids repeated calls to no-blocking IO when there is no IO present.

MPE/iX Specific

Due to POSIX/iX design considerations, the `read()` and `write()` system calls may not be interrupted by a signal while they are running system code (see the *POSIX.1/iX Reference Manual*).

`Select()` does not support signals in this release. Its non-support takes the following form: when blocked in `select()`, any signal that arrives, even those we are set to ignore, interrupts `select()` and causes it to return EINTR. Even ordinary signals that default to ignore, such as SIGCHLD (the signal generated when a child process terminates), cause `select()` to be interrupted. Care must be taken to account for this non-support.

The SIGURG signal is not supported in this release. This is the signal that is sent when urgent data is received on a socket.

None of the socket functions should be called from within a signal handler.

3

File System Intrinsics

Unix 4.3 BSD provides an interface to make IPC similar to file I/O. You can open, read, write, or close a file that is type socket.

CLOSE, SCLOSE

Note 

This routine is called `sclose` for non-POSIX users, and is called `close` for POSIX users.

C Interface

```
int close (s)
int s;

int sclose (s)
int s;
```

Description

Closes the socket descriptor indicated by *s*. If this file descriptor is the last reference to the socket, then it is deactivated. For example, on the last close of a socket, associated naming information and queued data are discarded.

The SO_LINGER option of `setsockopt` can be used to determine what happens to data queued at the time of the `close`.

For more information about `close` or any other POSIX function, refer to the *MPE/iX Developer's Kit Reference Manual, Volume 1*.

Return Value

If the call completes successfully, a value of 0 is returned. If the call is unsuccessful, a value of -1 is returned and *errno* is set to indicate the error.

Errors

The following error is returned by `close`:

[EBADF] The argument *s* is not a valid descriptor.

See Also

`getsockopt`, `setsockopt`

DUP

C Interface

```
int dup (s)
int s;
```

Description

This function returns a socket descriptor that refers to the same socket as specified by *s*. Both descriptors can be used to reference the socket. The new socket descriptor is the lowest numbered available socket descriptor.

For more information about `dup` or any other POSIX function, refer to the *MPE/iX Developer's Kit Reference Manual, Volume 1*.

Return Value

If the call is successful, a valid file descriptor referencing the socket is returned. If it fails a -1 is returned, and an error code is stored in *errno*.

Errors

[ENOTSOCK] The argument *s* is not a valid socket descriptor.

[ENOMEM] The process has no more descriptors available.

See Also

`socket`

FCNTL, SFCNTL

Note

Only `sfcntl` is currently available.

C Interface

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

int sfcntl (s, cmd, arg)
int s, cmd, arg;
```

Description

The `sfcntl` routine provides control over open sockets. The *s* parameter is an open socket.

The following are possible values of the *arg* parameter. They are also referred to as file status flags.

O_NDELAY	Non-blocking I/O.
O_NONBLOCK	POSIX-style non-blocking I/O.

The following are possible values of the *cmd* argument:

F_GETFL	Get file status flags described above.
F_SETFL	Set `O_NDELAY` and `O_NONBLOCK` depending upon the value of *arg*. It is not possible to set both `O_NDELAY` and `O_NONBLOCK`. Note: To set a socket to use blocking I/O (after previously setting it to use non-blocking I/O), the *arg* parameter should be 0.

Return Value

Upon successful completion, the value returned depends on *cmd* as follows:

F_GETFL	Value of file status flags and access modes.
F_SETFL	Value other than -1.
	Otherwise, a value of -1 is returned, and *errno* is set to indicate the error.

Errors

The following errors are returned by `sfcntl`:

[EBADF]	The *s* parameter is not a valid open file descriptor.
[EINVAL]	The *cmd* parameter is not a valid command.
[EINVAL]	The *cmd* parameter is `F_SETFL`, and both `O_NONBLOCK` and `O_NDELAY` are specified.

MPE/iX Specific The `sfcntl` call is used instead of `fcntl` on MPE/iX.

Author The `fcntl` intrinsic was developed by Hewlett-Packard, AT&T, and the University of California, Berkeley.

See Also `close`, `read`, `write`

IOCTL

C Interface

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

int ioctl (s, request, arg)
int s;
int request;
void *arg;
```

Description

The `ioctl` function provides an interface for setting different characteristics for a socket, and retrieving information on a socket.

The parameter *s* is a socket descriptor. The *request* parameter specifies which command to perform on the socket. The commands are defined in `<sys/ioctl.h>`. The different commands that are available are described below.

Note: Any data structure referenced by *arg* must **not** contain any pointers.

FIONREAD

Returns the number of bytes immediately readable from the socket in the long integer whose address is *arg*.

For SOCK_STREAM sockets, the number of bytes currently readable from this socket is returned in the integer with the address *arg*. For SOCK_DGRAM sockets, the number of bytes currently readable, plus the size of the `sockaddr` structure (defined in `<sys/socket.h>`), is returned in the integer with the address *arg*.

FIOSNBIO

Enables or disables non-blocking I/O for the socket. If the integer whose address is *arg* is non-zero, then non-blocking I/O is enabled; that is, subsequent reads and writes to the device file are handled in a non-blocking manner (see below). If the integer whose address is *arg* is 0, then non-blocking I/O is disabled.

For reads, non-blocking I/O prevents all read requests to that socket from blocking, whether the requests succeed or fail. Such read requests complete in one of three ways:

- If there is enough data available to satisfy the entire request, the read completes successfully, having read all of the data, and returns the number of bytes read;

- If there is not enough data available to satisfy the entire request, the read completes successfully, having read as much data as possible, and returns the number of bytes it was able to read;
- If there is no data available, the read fails and *errno* is set to EWOULDBLOCK.

For writes, non-blocking I/O prevents all write requests to that socket from blocking, whether the requests succeed or fail. Such a write request completes in one of three ways:

- If there is enough space available in the system to buffer all the data, the write completes successfully having written out all of the data, and returns the number of bytes written;
- If there is not enough space in the buffer to write out the entire request, the write completes successfully, having written as much data as possible, and returns the number of bytes it was able to write;
- If there is no space in the buffer, the write fails and *errno* is set to EWOULDBLOCK.

To prohibit non-blocking I/O from interfering with the O_NDELAY flag (see `fcntl` section), the functionality of O_NDELAY always supercedes the functionality of non-blocking I/O. This means that if O_NDELAY is set, the transport performs read requests in accordance with the definition of O_NDELAY. When O_NDELAY is not set, the definition of non-blocking I/O applies.

When a socket is created, non-blocking I/O is disabled.

Blocking mode is the default. See `accept`, `connect`, `recv`, and `send` for an explanation of how non-blocking mode is used.

FIOGNBIO — Gets the status of non-blocking I/O. If non-blocking I/O is enabled, then the integer whose address is *arg* is set to 1. If non-blocking I/O is disabled, then the integer whose address is *arg* is set to 0.

SIOCATMARK — For SOCK_STREAM TCP sockets, upon return the integer with the address *arg* is non-zero if urgent data has arrived. For

sockets other than SOCK_STREAM TCP sockets, on return the integer with the address *arg* is always zero.

SIOCSPGRP — This request sets the process group or process ID associated with the socket to be the value of the integer with the address *arg*. A process group or process ID associated with the socket in this manner is signaled when the state of the socket changes: SIGIO is delivered if the socket is asynchronous, as described in FIOASYNC below. If the value of the integer with the address *arg* is positive, the signal is sent to the process whose process ID matches the value specified. If the value is negative, the signal is sent to all the processes that have a process group equal to the absolute value of the value specified. If the value is zero, no signal is sent to any process. It is necessary to issue this request with a non-zero integer value to enable the signal delivery mechanism described above; the default for the process group or process ID value is zero.

SIOCGPGRP — This request returns the process group or process ID associated with the socket in the integer with the address *arg*. If the value of the integer with the address *arg* is positive, then the value returned corresponds to a process ID. If the value is negative, then the value corresponds to all processes that have a process group equal to the absolute value of that value.

FIOASYNC — If the integer whose address is *arg* is non-zero, this request sets the state of the socket as asynchronous. Otherwise, the socket is put into synchronous mode (the default). Asynchronous mode enables the delivery of the SIGIO signal when a) new data arrives, or b) for connection-oriented protocols, whenever additional outgoing buffer space becomes available, or when the connection is established or broken. The process group or process ID associated with the socket must be non-zero in order for SIGIO signals to be sent; the signal is delivered according to the semantics of SIOCGPGRP described above.

Since both the `fcntl` O_NONBLOCK and O_NDELAY flags and `ioctl` FIOSNBIO requests are supported, some clarification on how these features interact is necessary. If the O_NONBLOCK

or O_NDELAY flag has been set, `recv` and `send` requests behave accordingly, regardless of any FIOSNBIO requests. If neither the O_NONBLOCK flag nor the O_NDELAY flag has been set, FIOSNBIO requests control of the behavior of `recv` and `send`.

Return Value

If the call is successful, a 0 is returned. If an error has occurred, a value of -1 is returned and *errno* is set to indicate the error.

Errors

`ioctl` fails if one or more of the following are true: IOC_OUT is ignored if an error occurs.

[EBADF]	The argument *s* is not a valid open file descriptor.
[EFAULT]	The system detected a NULL address while attempting to use the *arg* parameter passed by the caller.
[ENOTTY]	The request is not appropriate to the selected device.
[EINVAL]	The request parameter of the *arg* parameter is invalid, or a socket type that is not supported was specified.
[EINTR]	The `ioctl` call was interrupted by a signal.
[EPERM]	Typically, this error indicates that an `ioctl` request was attempted that is forbidden in some way to the calling process.

MPE/iX Specific

Programs using `ioctl()` must be linked with the POSIX C library.

Author

`ioctl` was developed by AT&T and HP.

See Also

fcntl, getsockopt, socket

READ

C Interface

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

Description

The `read` function is similar to the `recv` call except there is no flags parameter. It behaves the same way as a `recv` call with *flags* set to zero.

For more information about read or any other POSIX function, refer to the *MPE/iX Developer's Kit Reference Manual, Volume 1*.

Return Value

If the call is successful, it returns the number of bytes received. If the call fails, a -1 is returned, and an error code is stored in *errno*. A zero is returned if the socket is blocking and the transport connection to the remote node fails.

See Also

`recv`

SELECT

C Interface

```
#include <time.h>

int select(nfds, readfds, writefds, exceptfds,timeout)
int nfds, *readfds, *writefds, *exceptfds;
struc timeval *timeout;
```

Description

The `select` intrinsic examines the file descriptors specified by the bit masks *readfds*, *writefds*, and *exceptfds*. The bits from 0 through *nfds*-1 are examined. File descriptor f is represented by the bit $1 < f$ in the masks. More formally, a file descriptor is represented by the following:

```
fds[(f / BITS_PER_INT)] & (1 << (f % BITS_PER_INT))
```

When `select` completes successfully, it returns the three bit masks modified as follows: For each file descriptor less than *nfds*, the corresponding bit in each mask is set if the bit was set upon entry and the file descriptor is ready for reading or writing, or has an exceptional condition pending.

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` waits until an event causes one of the masks to be returned with a valid (non-zero) value. To poll, the *timeout* argument should be non-zero, pointing to a zero valued *timeval* structure. Specific implementations may place limitations on the maximum *timeout* interval supported.

Any or all of *readfds*, *writefds*, and *exceptfds* may be given as 0 if no descriptors are of interest. If all of the masks are given as 0 and *timeout* is not a zero pointer, `select` blocks for the time specified, or until interrupted by a signal. If all of the masks are given as 0 and *timeout* is a zero pointer, `select` blocks until interrupted by a signal.

Ordinary files always select true whenever selecting on reads, writes, and/or exceptions.

Examples

Example 1

The following call to `select` checks if any of four sockets are ready for reading. The `select` intrinsic times out after 5 seconds if no sockets are ready for reading:

Note

The code for opening the sockets or reading from the sockets is not shown in this example and this example must be modified if the calling process has more than 32 file descriptors open.

```
#define MASK(f)     (1 << (f))
#define NSDS 4int sd[NSDS];
int sdmask[NSDS];
int readmask = 0;
int readfds;
int nfound, i;
struct timeval timeout;

     /* First open each socket for reading and put the  */
     /* file descriptors into array sd[NSDS].  The code */
     /* for opening the sockets is not shown here.      */

     for (i=0; i < NSDS; i++) {
       sdmask[i] = MASK(sd[i]);
       readmask |= sdmask[i];
     }
     timeout.tv_sec  = 5;
     timeout.tv_usec = 0;
     readfds = readmask;

     /* select on NSDS+3 file descriptors if stdin, stdout */
     /* and stderr are also open                           */

     if ((nfound = select (NSDS+3, &readfds, 0, 0, &timeout)) == -1)
       perror ("select failed");
     else
       if (nfound == 0)
         printf ("select timed out \n");
       else
         for (i=0; i < NSDS; i++)
           if (sdmask[i] & readfds)
             /* Read from sd[i].  The code for reading */
             /* is not shown here.                     */
           else
             printf ("sd[%d] is not ready for reading \n",i);
```

Example 2

The following programming example shows how `select` can be used to wait on multiple sockets.

```
/* This program is an example of how select can be
used on multiple sockets */
/* on MPE/iX.                              */

/* Compile with SOCKET_SOURCE and POSIX_SOURCE defined. */
/* Link with socketrl and libcinit.          */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/in.h>
```

```
#include <sys/errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <time.h>

#define TRUE 0
#define FALSE 1

main ()
{
  int maxfds;
  int sock, sock2, peer1;
  int struc_len;
  int fret;
  int done = FALSE;
  char data[256];
  char data_to_send = 'D';
  char *datptr;
  int dlen;
  int readfds;
  int writefds;
  struc timeval timeout;
  struc sockaddr_in sockaddr;

  sock = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP;

  peer1 = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);

  sockaddr.sin_family = AF_INET;
  sockaddr.sin_addr.s_addr = INADDR_THISHOST;
  sockaddr.sin_port = 4444;

  struc_len = 8;
  if (bind (sock, (struc sockaddr *) &sockaddr,
        sizeof sockaddr) << 0 ) {
    printf ("Bind failed\n");};
  listen (sock, 10);

  sfcntl (peer1, F_SETFL, O_NONBLOCK);

/* connect() returns with EINPROGRESS. */
   fret = connect (peer1, (struct sockaddr *) &sockaddr,
             struc_len);

   sock2 = accept (sock,
             (struc sockaddr *) &sockaddr,
             &struc_len);

/* Call recv to complete the connection */
   recv (peer1, 0, 0, 0);
```

```
/* sock2 and peer1 are now connected */
   daptr = data;
   done = FALSE;
   while (done == FALSE) {

/* This code example shows how to use select() to wait on multiple
*/
/* sockets. Note that first you have to find the maximum descriptor
*/
/* number being used. The appropriate bit(s) must be set for the */
/* select masks and then they must be checked after the call. In */
/* this example, sock2 is waiting to receive data from peer1.  */
    if (peer1 << sock2) {
       maxfds = sock2;
       }
    else
       maxfds = peer1;

/* set a 5 second timer for the call to select(). */
    timeout.tv_sec = 5;
    timeout.tv_usec = 0;

    writefds = (1 << peer1);
    readfds = (1 << peer1) + (1 << sock2);

    fret = select (maxfds + 1, &readfds, &writefds, 0, &timeout);

    if (fret << = 0) {
       printf ("error1\n");
       done = TRUE;
       }
    else
       {
       if ((readfds && (1 << sock2)) ! = 0) {
          dlen = 100;
          fret = recv (sock2, datptr, dlen, 0);
       printf ("received %d bytes.\n", fret);
       if (data[0]! = data_to_send) {
          printf ("error2\n");
          };
       done = TRUE;
       }
    else
       {
       if ((writefds && (1 << peer1))! = 0) {
          dlen = 1;
          data[0] = data_to_send;
          fret = send (peer1, datptr, dlen, 0);
          printf ("sent %d bytes.\n", fret);
          }
       else
```

```
            {
            printf ("error3\n");
            }
         }
      };/* end else */
   };/* end while */
}
```

Return Value

The `select` intrinsic returns the number of descriptors contained in the bit masks. If an error occurs, -1 is returned and an error code is stored in *errno*. If the time limit expires, then `select` returns 0, and all of the masks are cleared.

Errors

The `select` intrinsic returns the following errors:

[EBADF]	One or more of the bit masks specified an invalid descriptor.
[EFAULT]	One or more of the pointers was invalid.
[EINVAL]	An invalid *timeval* was passed for timeout.
[EINVAL]	The value of *nfds* is less than zero.

Note

The file descriptor masks are always modified on return, even if the call returns as the result of a timeout.

Author

The `select` intrinsic was developed by Hewlett-Packard and the University of California, Berkeley.

WRITE

C Interface

```
int write (s, msg, len);
int s;
char *msg;
int len
```

Description

The `write` function is similar to the `send` call except there is no *flags* parameter. It behaves the same way as a `send` call with *flags* set to zero.

For more information about `write` or any other POSIX function, refer to the *MPE/iX Developer's Kit Reference Manual, Volume 1*.

Return Value

If successful, the call returns the number of characters sent. If the call fails, a -1 is returned, and an error code is stored in *errno*.

MPE/iX Specific

The `write` call is supported as of MPE/iX release 4.5.

See Also

`send`

4

Name Service Routines

This section describes several library routines that can best be described as name service routines, since they return information based on names, addresses, and numbers. Each subsection describes a set of five related intrinsics, as indicated below:

GETHOSTENT	GETNETENT	GETPROTOENT	GETSERVENT
gethostent	getnetent	getprotoent	getservent
sethostent	setnetent	setprotoent	setservent
gethostbyname	getnetbyname	getprotobyname	getservbyname
gethostbyaddr	getnetbyaddr	getprotobynumber	getservbyport
endhostent	endnetent	endprotoent	endservent

These routines are stored in a native mode relocatable library file and are used for `AF_INET` only. The relocatable library file name is `SOCKETRL.NET.SYS`. When linking your programs, you should include this file in the link list. For example,

```
link objfile,progfile;rl=socketrl.net.sys,libc.lib.sys
```

Ensure that you link with the POSIX library (`/lib/libc.a`) instead of `libc.lib.sys` for POSIX programs. Note that if you are using the POSIX library you must use file indirection as shown in the programming example in Chapter 5.

INET_ADDR, INET_NETWORK, INET_NTOA

C Interface

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

unsigned long inet_addr(cp);
  const char *cp;

unsigned long inet_network(cp);
  const char *cp;

char *inet_ntoa(in);
  struc in_addr in;
```

Description

The routines `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 internet network numbers, respectively. Their return values can be assigned to a struct in_addr (defined in /usr/include/netinet/in.h) as in the following example:

```
struct in_addr addr;
char *cp;

addr.s_addr = inet_addr(cp);
```

`inet_ntoa` takes an interet address and returns an ASCII string representing the address in "dot" (.) notation.

All internet addresses are returned in network order (bytes ordered from left to right). All network numbers and local address parts are returned as machine-format integer values. Bytes in HP-UX systems are ordered from left to right.

Internet Addresses: 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 in 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 in 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 (i.e., a leading 0x or 0X implies hexadecimal; a leading 0 implies octal; otherwise, the number is interpreted as decimal).

Return Value

`inet_addr` and `inet_network` return -1 for malformed requests.

Warnings

The string returned by `inet_ntoa` resides in a static memory area.

Author

UCB (University of California at Berkeley)

See Also

`gethostent, getnetent`

GETHOSTENT

C Interface

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

extern int h_errno;

struct hostent *gethostent()

struct hostent *gethostbyname(name)
char *name;

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

sethostent(stayopen)
int stayopen;

endhostent()
```

Description

The `gethostent`, `gethostbyname`, and `gethostbyaddr` subroutines return a pointer to a structure defined as follows in `netdb.h`:

```
struct  hostent {
char   *h_name;         /* official name of host            */
char   **h_aliases;     /* alias list                       */
int    h_addrtype;      /* address type                     */
int    h_length;        /* length of address                */
char   **h_addr_list;   /* list of addresses from name server */
};
#define h_addr h_addr_list[0]   /* address for backward */
                                /* compatibility        */
```

The members of this structure are as follows:

h_name	Official name of the host.
h_aliases	A null-terminated array of alternate names for the host.
h_addrtype	The type of address being returned; currently always `AF_INET`.
h_length	The length, in bytes, of the address.
h_addr_list	A null-terminated array of network addresses for the host.

h_addr — The first address in h_addr_list; this is for compatibility with previous HP-UX implementations, where a `struct` hostent contains only one network address per host.

If the local system is configured to use the name server, then:

- The `gethostent` subroutine always returns a null pointer.
- The `sethostent` subroutine, if the stayopen flag is non-zero, requests the use of a connected stream socket for queries to the name server. The connection is retained after each call to `gethostbyname` or `gethostbyaddr`.
- The `endhostent` subroutine closes the stream socket connection.

The `gethostbyname` and `gethostbyaddr` subroutines each retrieve host information from the name server through the resolver. Names are matched in a case-insensitive manner; for example, `berkeley.edu`, `Berkeley.EDU`, and `BERKELEY.EDU` would all match the entry for `berkeley.edu`.

The resolver reads the configuration file `RESLVCNF.NET.SYS` to get the default domain name and the Internet address of the initial hosts running the name server. If the environment variable `LOCALDOMAIN` is set by the user, that name is used as the default domain (overriding any other default). If the name server Internet addresses are not listed in the configuration file, the resolver aborts and the hosts file is tried (see below). If there are errors in the configuration file, they are silently ignored.

If the local system is not using the name server, then:

- The `gethostent` subroutine reads the next line of `HOSTS.NET.SYS`, opening the file if necessary.
- The `sethostent` subroutine opens and rewinds the file. If the stayopen flag is non-zero, the host database is not closed after each call to `gethostent` (either directly or indirectly through one of the other `gethost` calls).
- The `endhostent` subroutine closes the file.
- The `gethostbyname` subroutine sequentially searches from the beginning of the file until a host name (among either the official names or the aliases) matching its parameter *name* is found, or until EOF is encountered. Names are matched in a case-insensitive manner; for example, `berkeley.edu`, `Berkeley.EDU`, and `BERKELEY.EDU` would all match the entry for `berkeley.edu`.
- The `gethostbyaddr` subroutine sequentially searches from the beginning of the file until an Internet address matching its parameter *addr* is found, or until EOF is encountered.

In calls to `gethostbyaddr`, the parameter *addr* must point to an internet address in network order (refer to the `inet` section) and the *addr* parameter must be 4-byte aligned, or an escape is generated.

The parameter *len* must be the number of bytes in an Internet address, that is, *sizeof* (`struct` *in_addr*). The parameter *type* must be the constant `AF_INET`.

Return Value

If successful, `gethostbyname`, `gethostbyaddr`, and `gethostent` return a pointer to the requested *hostent* struct. The `gethostbyname` and `gethostbyaddr` subroutines return NULL if their *host* or *addr* parameters, respectively, cannot be found in the database. If `hosts.net.sys` is being used, they also return NULL if they are unable to open `hosts.net.sys`. The `gethostbyaddr` subroutine also returns NULL if either its *addr* or *len* parameter is invalid. The `gethostent` subroutine always returns NULL if the name server is being used.

If the name server is being used and `gethostbyname` or `gethostbyaddr` returns a NULL pointer, the external integer *h_errno* contains one of the following values:

[HOST_NOT_FOUND]	No such host 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 time later may succeed.
[NO_RECOVERY]	This is a non-recoverable error.
[NO_ADDRESS]	The requested name is valid but has no IP address; this is not a temporary error. This means that another type of request to the name server results in an answer.

If the name server is not being used, the value of *h_errno* may not be meaningful.

Restrictions

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.

MPE/iX Specific

The names of the hosts file and resolver configuration file on MPE/iX are `HOSTS.NET.SYS` and `RESLVCNF.NET.SYS`, as opposed to `/etc/hosts` and `/etc/resolv.conf` on HP-UX.

Author

UCB (University of California at Berkeley)

Files

`HOSTS.NET.SYS`, `RESLVCNF.NET.SYS`

See Also

`resolver`, `hosts`

GETNETENT

C Interface

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

struct netent *getnetent()

struct netent *getnetbyname(name)
char *name;

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

setnetent(stayopen)
int stayopen;

endnetent()
```

Description

The `getnetent`, `getnetbyname`, and `getnetbyaddr` subroutines each return a pointer to an object with the following structure. This structure contains fields found in the network protocol database, `/etc/networks`.

```
struct    netent {
    char *n_name;       /* official name of net */
    char **n_aliases;   /* alias list           */
    int  n_addrtype;    /* net number type      */
    long n_net;         /* net number           */
};
```

The members of this structure are as follows:

n_name — The official name of the network.

n_aliases — A null-terminated list of alternate names for the network.

n_addrtype — The type of the network number returned, always `AF_INET`.

n_net — The network number. Network numbers are returned in machine byte order.

The `getnetent` subroutine reads the next line of the file, opening the file if necessary.

The `setnetent` subroutine opens and rewinds the file. If the *stayopen* flag is non-zero, the network database is not closed after each call to `getnetent` (either directly, or indirectly through one of the other `getnet` calls).

The `endnetent` subroutine closes the file.

The `getnetbyname` subroutine sequentially searches from the beginning of the file until a network name (among either the official names or the aliases) matching its parameter *name* is found, or until EOF is encountered.

The `getnetbyaddr` subroutine sequentially searches from the beginning of the file until a network number matching its parameter *net* is found, or until EOF is encountered. The parameter *net* must be in network order. The parameter *type* must be the constant `AF_INET`.

Network numbers are supplied in host order. (Refer to the `inet` section.)

Restrictions

All information is contained in a static area, so it must be copied if it is to be saved. Only Internet network numbers are currently understood.

Return Value

The `getnetent`, `getnetbyname`, and `getnetbyaddr` subroutines return a null pointer (0) on EOF or when they are unable to open `NETWORKS.NET.SYS`. The `getnetbyaddr` subroutine also returns a null pointer if its parameter *type* is invalid.

MPE/iX Specific

The name of the networks file on MPE/iX is `NETWORKS.NET.SYS`, as opposed to `/etc/networks` on HP-UX.

Author

UCB (University of California at Berkeley)

Files

`NETWORKS.NET.SYS`

See Also

`networks`

GETPROTOENT

C Interface

```
#include <netdb.h>

struct protoent *getprotoent()

struct protoent *getprotobyname(name)
char *name;

struct protoent *getprotobynumber(proto)
int proto;

setprotoent(stayopen)
int stayopen;

endprotoent()
```

Description

The `getprotoent`, `getprotobyname`, and `getprotobynumber` subroutines each return a pointer to an object with the following structure. This structure contains fields found in the network protocol database, `/etc/protocols`.

```
struct    protoent {
     char *p_name;        /* official name of protocol */
     char **p_aliases;    /* alias list                */
     long p_proto;        /* protocol number           */
};
```

The members of this structure are as follows:

p_name — The official name of the protocol.

p_aliases — A null-terminated list of alternate names for the protocol.

p_proto — The protocol number.

The `getprotoent` subroutine reads the next line of the file, opening the file if necessary.

The `setprotoent` subroutine opens and rewinds the file. If the *stayopen* flag is non-zero, the protocol database is not closed after each call to `getprotoent` (either directly or indirectly through one of the other `getproto` calls).

The `endprotoent` subroutine closes the file.

The `getprotobyname` and `getprotobynumber` subroutines sequentially search from the beginning of the file until a protocol name matching the parameter *name* or a protocol number matching the parameter *proto* is found, or until EOF is encountered.

Restrictions All information is contained in a static area, so it must be copied if it is to be saved. Only the Internet protocols are currently understood.

Return Value The `getprotoent`, `getprotobyname`, and `getprotobynumber` subroutines return a null pointer (0) on EOF or when they are unable to open `PROTOCOL.NET.SYS`.

MPE/iX Specific The name of the protocols file on MPE/iX is `PROTOCOL.NET.SYS`, as opposed to `/etc/protocols` on HP-UX.

Author UCB (University of California at Berkeley)

Files `PROTOCOL.NET.SYS`

See Also `protocols`

GETSERVENT

C Interface

```
#include <netdb.h>

struct servent *getservent()

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

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

setservent(stayopen)
int stayopen;

endservent()
```

Description

The `getservent`, `getservbyname`, and `getservbyport` subroutines each return a pointer to an object with the following structure containing the broken-out fields of a line in the network services database, `/etc/services`.

```
struct    servent {
     char *s_name;     /* official name of service */
     char **s_aliases; /* alias list               */
     long s_port;      /* port service resides at  */
     char *s_proto;    /* protocol to use          */
};
```

The members of this structure are as follows:

s_name — The official name of the service.

s_aliases — A null-terminated list of alternate names for the service.

s_port — The port number at which the service resides. Port numbers are returned in network byte order.

s_proto — The name of the protocol to use when contacting the service.

The `getservent` subroutine reads the next line of the file, opening the file if necessary.

The `setservent` subroutine opens and rewinds the file. If the *stayopen* flag is non-zero, the services database is not closed after each call to `getservent` (either directly or indirectly through one of the other `getserv` calls).

The `endservent` subroutine closes the file.

The `getservbyname` and `getservbyport` subroutines sequentially search from the beginning of the file until a service name (among either the official names or the aliases) matching the parameter *name* or a port number matching the parameter *port* is found, or until EOF is encountered. If a non-NULL protocol name is also supplied (for example, `tcp` or `udp`), searches must also match the protocol.

Restrictions

All information is contained in a static area, so it must be copied if it is to be saved.

Return Value

The `getservent`, `getservbyname`, and `getservbyport` subroutines return a null pointer (0) on EOF or when they are unable to open `SERVICES.NET.SYS`.

MPE/iX Specific

The name of the services file on MPE/iX is `SERVICES.NET.SYS`, as opposed to `/etc/services` on HP-UX.

Author

UCB (University of California at Berkeley)

Files

`SERVICES.NET.SYS`

See Also

`services`

5

Programming Example

This section contains an example source program, describes how it was compiled and linked, and lists the output generated from it.

Source Program

The following program was broken into steps to show how a connection is established between two sockets.

1. Establish the connection in loopback, using a single process:

```
#include </stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <sys/in.h>
#include <sys/errno.h>
#include <fcntl.h>
#include <unistd.h>

main()
{
   int sock;
   int peer1;
   int struc_len;
   int fret;
   int sock2;
   struct sockaddr_in sockaddr;
```

2. Create the two sockets:

```
sock=socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (sock == -1)
   print ("Error creating socket.\n");

peer1 = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (peer1 == -1)
   printf ("Error creating socket.\n");
```

3. Create the socket structure to bind the socket. The IP address used is 0 (loopback), and the SAP is set to an arbitrary constant (4444). This structure is used for both the `bind` and the `connect`:

```
sockaddr.sin_family = AF_INET;
sockaddr.sin_addr.s_addr = INADDR_THISHOST;
```

```
sockaddr.sin_port = 4444;

struc_len = 16;

if (bind (sock, (struct sockaddr *) &sockaddr,
         sizeof (sockaddr)) < 0 )
   printf ("Bind failed\n);
listen (sock,10);
```

4. Set the socket initiating the connection to be nonblocking. This allows the `connect` to return to the user without blocking and waiting for `accept`:

Note

The `sfcntl` function should be used until the `fcntl` is provided by the operating system.

```
sfcntl (peer1, F_SETFL, O_NONBLOCK);

fret = connect (peer1, (struct sockaddr *) &sockaddr, struc_len);
if (fret == -1)
   printf ("Connect failed with error %d\n",errno);
else
   printf ("Connect succeeded\n");

sock2 = accept (sock, (struct sockaddr *) &sockaddr, &struc_len);
if (sock2 == -1)
   printf ("Accept failed\n");
else
   printf ("Accept succeeded\n");
```

5. Call `recv` to complete the connection:

```
recv (peer1, 0,0,0);
}   /* end main */
```

Compiling

Some MPE/iX include files expect certain variables to be defined. For example, `<types.h.sys>` expects `SOCKET_SOURCE` to be defined. Defines are declared in a C program by using `#define`; however, instead of modifying source code, a define can be declared at runtime.

The following example shows how to compile the program:

```
:ccxl sourcepg,obj,listing;info="-Aa &
-D_SOCKET_SOURCE -D_POSIX_SOURCE"
```

Linking

The following example shows how to link the program:

```
:link from=obj;rl=^rllist;to=prog
```

The following `RLLIST` file was used in linking the above program:

```
socketrl.net.sys
libcinit.lib.sys
```

Note: Ensure that you link with the POSIX library (`/lib/libc.a`) instead of `libcinit.lib.sys` for POSIX programs. If you are using `fork()`, you need to link with PH capabilities.

Output

When the above program was run, it had the following output:

```
Connect failed with error 245
Accept succeeded
```

Note that the `connect` failed with error 245. This corresponds to `EINPROGRESS`, indicating that the `connect` could not complete immediately because an `accept` had not yet been issued. If the connection was not using the nonblocking mode, then the `connect` would have blocked.