NAME

socket − create an endpoint for communication

SYNOPSIS

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

int socket(int *domain***, int** *type***, int** *protocol***);**

DESCRIPTION

Socket creates an endpoint for communication and returns a descriptor.

The *domain* parameter specifies a communication domain; this selects the protocol family which will be used for communication. These families are defined in **<sys/socket.h>**. The currently understood formats include:

Name	Purpose	Man page
PF_UNIX,PF_LOCAL	Local communication	**unix**(7)
PF_INET	IPv4 Internet protocols	**ip**(7)
PF_INET6	IPv6 Internet protocols	
PF_IPX	IPX − Novell protocols	
PF_NETLINK	Kernel user interface device	**netlink**(7)
PF_X25	ITU-T X.25 / ISO-8208 protocol	**x25**(7)
PF_AX25	Amateur radio AX.25 protocol	
PF_ATMPVC	Access to raw ATM PVCs	
PF_APPLETALK	Appletalk	**ddp**(7)
PF_PACKET	Low level packet interface	**packet**(7)

The socket has the indicated *type*, which specifies the communication semantics. Currently defined types are:

SOCK_STREAM
Provides sequenced, reliable, two-way, connection-based byte streams. An out-of-band data transmission mechanism may be supported.

SOCK_DGRAM
Supports datagrams (connectionless, unreliable messages of a fixed maximum length).

SOCK_SEQPACKET
Provides a sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length; a consumer is required to read an entire packet with each read system call.

SOCK_RAW
Provides raw network protocol access.

SOCK_RDM
Provides a reliable datagram layer that does not guarantee ordering.

SOCK_PACKET
Obsolete and should not be used in new programs; see **packet**(7).

Some socket types may not be implemented by all protocol families; for example, **SOCK_SEQPACKET** is not implemented for **AF_INET**.

The *protocol* specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner. The protocol number to use is specific to the "communication domain" in which communication is to take place; see **protocols**(5). See **getprotoent**(3) on how to map protocol name strings to protocol numbers.

Sockets of type **SOCK_STREAM** are full-duplex byte streams, similar to pipes. They do not preserve record boundaries. A stream socket must be in a *connected* state before any data may be sent or received on it. A connection to another socket is created with a **connect**(2) call. Once connected, data may be

transferred using **read**(2) and **write**(2) calls or some variant of the **send**(2) and **recv**(2) calls. When a session has been completed a **close**(2) may be performed. Out-of-band data may also be transmitted as described in **send**(2) and received as described in **recv**(2).

The communications protocols which implement a **SOCK_STREAM** ensure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered to be dead. When **SO_KEEPALIVE** is enabled on the socket the protocol checks in a protocol-specific manner if the other end is still alive. A **SIGPIPE** signal is raised if a process sends or receives on a broken stream; this causes naive processes, which do not handle the signal, to exit. **SOCK_SEQPACKET** sockets employ the same system calls as **SOCK_STREAM** sockets. The only difference is that **read**(2) calls will return only the amount of data requested, and any remaining in the arriving packet will be discarded. Also all message boundaries in incoming datagrams are preserved.

SOCK_DGRAM and **SOCK_RAW** sockets allow sending of datagrams to correspondents named in **send**(2) calls. Datagrams are generally received with **recvfrom**(2), which returns the next datagram with its return address.

SOCK_PACKET is an obsolete socket type to receive raw packets directly from the device driver. Use **packet**(7) instead.

An **fcntl**(2) call with the the **F_SETOWN** argument can be used to specify a process group to receive a **SIGURG** signal when the out-of-band data arrives or **SIGPIPE** signal when a **SOCK_STREAM** connection breaks unexpectedly. It may also be used to set the process or process group that receives the I/O and asynchronous notification of I/O events via **SIGIO.** Using **F_SETOWN** is equivalent to an **ioctl**(2) call with the SIOSETOWN argument.

When the network signals an error condition to the protocol module (e.g. using a ICMP message for IP) the pending error flag is set for the socket. The next operation on this socket will return the error code of the pending error. For some protocols it is possible to enable a per-socket error queue to retrieve detailed information about the error; see **IP_RECVERR** in **ip**(7)**.**

The operation of sockets is controlled by socket level *options*. These options are defined in **<sys/socket.h>**. The functions **setsockopt**(2) and **getsockopt**(2) are used to set and get options, respectively.

RETURN VALUE

−1 is returned if an error occurs; otherwise the return value is a descriptor referencing the socket.

ERRORS

EPROTONOSUPPORT
The protocol type or the specified protocol is not supported within this domain.

EAFNOSUPPORT
The implementation does not support the specified address family.

ENFILE
Not enough kernel memory to allocate a new socket structure.

EMFILE
Process file table overflow.

EACCES
Permission to create a socket of the specified type and/or protocol is denied.

ENOBUFS or **ENOMEM**
Insufficient memory is available. The socket cannot be created until sufficient resources are freed.

EINVAL
Unknown protocol, or protocol family not available.

Other errors may be generated by the underlying protocol modules.

CONFORMING TO

4.4BSD (the **socket** function call appeared in 4.2BSD). Generally portable to/from non-BSD systems supporting clones of the BSD socket layer (including System V variants).

NOTE

The manifest constants used under BSD 4.* for protocol families are PF_UNIX, PF_INET, etc., while AF_UNIX etc. are used for address families. However, already the BSD man page promises: "The protocol family generally is the same as the address family", and subsequent standards use AF_* everywhere.

BUGS

SOCK_UUCP is not implemented yet.

SEE ALSO

accept(2), **bind**(2), **connect**(2), **getprotoent**(3), **getsockname**(2), **getsockopt**(2), **ioctl**(2), **listen**(2), **read**(2), **recv**(2), **select**(2), **send**(2), **shutdown**(2), **socketpair**(2), **write**(2)

"An Introductory 4.3 BSD Interprocess Communication Tutorial" is reprinted in *UNIX Programmer's Supplementary Documents Volume 1*.

"BSD Interprocess Communication Tutorial" is reprinted in *UNIX Programmer's Supplementary Documents Volume 1*.

NAME

packet, PF_PACKET − packet interface on device level.

SYNOPSIS

```
#include <sys/socket.h>
#include <features.h>     /* for the glibc version number */
#if __GLIBC__ >= 2 && __GLIBC_MINOR >= 1
#include <netpacket/packet.h>
#include <net/ethernet.h>          /* the L2 protocols */
#else
#include <asm/types.h>
#include <linux/if_packet.h>
#include <linux/if_ether.h>        /* The L2 protocols */
#endif
```

packet_socket = socket(PF_PACKET, int *socket_type***, int** *protocol***);**

DESCRIPTION

Packet sockets are used to receive or send raw packets at the device driver (OSI Layer 2) level. They allow the user to implement protocol modules in user space on top of the physical layer.

The *socket_type* is either **SOCK_RAW** for raw packets including the link level header or **SOCK_DGRAM** for cooked packets with the link level header removed. The link level header information is available in a common format in a **sockaddr_ll**. *protocol* is the IEEE 802.3 protocol number in network order. See the **<linux/if_ether.h>** include file for a list of allowed protocols. When protocol is set to **htons(ETH_P_ALL)** then all protocols are received. All incoming packets of that protocol type will be passed to the packet socket before they are passed to the protocols implemented in the kernel.

Only processes with effective uid 0 or the **CAP_NET_RAW** capability may open packet sockets.

SOCK_RAW packets are passed to and from the device driver without any changes in the packet data. When receiving a packet, the address is still parsed and passed in a standard **sockaddr_ll** address structure. When transmitting a packet, the user supplied buffer should contain the physical layer header. That packet is then queued unmodified to the network driver of the interface defined by the destination address. Some device drivers always add other headers. **SOCK_RAW** is similar to but not compatible with the obsolete **SOCK_PACKET** of Linux 2.0.

SOCK_DGRAM operates on a slightly higher level. The physical header is removed before the packet is passed to the user. Packets sent through a **SOCK_DGRAM** packet socket get a suitable physical layer header based on the information in the **sockaddr_ll** destination address before they are queued.

By default all packets of the specified protocol type are passed to a packet socket. To only get packets from a specific interface use **bind**(2) specifying an address in a **struct sockaddr_ll** to bind the packet socket to an interface. Only the **sll_protocol** and the **sll_ifindex** address fields are used for purposes of binding.

The **connect**(2) operation is not supported on packet sockets.

When the **MSG_TRUNC** flag is passed to **recvmsg**(2), **recv**(2), **recvfrom**(2) the real length of the packet on the wire is always returned, even when it is longer than the buffer.

ADDRESS TYPES

The sockaddr_ll is a device independent physical layer address.

```
struct sockaddr_ll {
```

```
    unsigned short  sll_family;   /* Always AF_PACKET */
    unsigned short  sll_protocol; /* Physical layer protocol */
    int             sll_ifindex;  /* Interface number */
    unsigned short  sll_hatype;   /* Header type */
    unsigned char   sll_pkttype;  /* Packet type */
    unsigned char   sll_halen;    /* Length of address */
    unsigned char   sll_addr[8];  /* Physical layer address */
};
```

sll_protocol is the standard ethernet protocol type in network order as defined in the **linux/if_ether.h** include file. It defaults to the socket's protocol. **sll_ifindex** is the interface index of the interface (see **netdevice**(7)); 0 matches any interface (only legal for binding). **sll_hatype** is a ARP type as defined in the **linux/if_arp.h** include file. **sll_pkttype** contains the packet type. Valid types are **PACKET_HOST** for a packet addressed to the local host, **PACKET_BROADCAST** for a physical layer broadcast packet, **PACKET_MULTICAST** for a packet sent to a physical layer multicast address, **PACKET_OTHERHOST** for a packet to some other host that has been caught by a device driver in promiscuous mode, and **PACKET_OUTGOING** for a packet originated from the local host that is looped back to a packet socket. These types make only sense for receiving. **sll_addr** and **sll_halen** contain the physical layer (e.g. IEEE 802.3) address and its length. The exact interpretation depends on the device.

When you send packets it is enough to specify **sll_family**, **sll_addr**, **sll_halen**, **sll_ifindex**. The other fields should be 0. **sll_hatype** and **sll_pkttype** are set on received packets for your information. For bind only **sll_protocol** and **sll_ifindex** are used.

SOCKET OPTIONS

Packet sockets can be used to configure physical layer multicasting and promiscuous mode. It works by calling **setsockopt**(2) on a packet socket for SOL_PACKET and one of the options **PACKET_ADD_MEMBERSHIP** to add a binding or **PACKET_DROP_MEMBERSHIP** to drop it. They both expect a **packet_mreq** structure as argument:

```
struct packet_mreq
{
    int             mr_ifindex;   /* interface index */
    unsigned short  mr_type;      /* action */
    unsigned short  mr_alen;      /* address length */
    unsigned char   mr_address[8]; /* physical layer address */
};
```

mr_ifindex contains the interface index for the interface whose status should be changed. The **mr_type** parameter specifies which action to perform. **PACKET_MR_PROMISC** enables receiving all packets on a shared medium - often known as ''promiscuous mode'', **PACKET_MR_MULTICAST** binds the socket to the physical layer multicast group specified in **mr_address** and **mr_alen**, and **PACKET_MR_ALLMULTI** sets the socket up to receive all multicast packets arriving at the interface.

In addition the traditional ioctls **SIOCSIFFLAGS, SIOCADDMULTI, SIOCDELMULTI** can be used for the same purpose.

IOCTLS

SIOCGSTAMP can be used to receive the time stamp of the last received packet. Argument is a **struct timeval.**

In addition all standard ioctls defined in **netdevice**(7) and **socket**(7) are valid on packet sockets.

ERROR HANDLING

Packet sockets do no error handling other than errors occurred while passing the packet to the device driver. They don't have the concept of a pending error.

COMPATIBILITY

In Linux 2.0, the only way to get a packet socket was by calling **socket(PF_INET, SOCK_PACKET,** *protocol***)**. This is still supported but strongly deprecated. The main difference between the two methods is that **SOCK_PACKET** uses the old **struct sockaddr_pkt** to specify an interface, which doesn't provide physical layer independence.

```
struct sockaddr_pkt
{
    unsigned short      spkt_family;
    unsigned char       spkt_device[14];
    unsigned short      spkt_protocol;
};
```

spkt_family contains the device type, **spkt_protocol** is the IEEE 802.3 protocol type as defined in **<sys/if_ether.h>** and **spkt_device** is the device name as a null terminated string, e.g. eth0.

This structure is obsolete and should not be used in new code.

NOTES

For portable programs it is suggested to use **PF_PACKET** via **pcap**(3); although this only covers a subset of the **PF_PACKET** features.

The **SOCK_DGRAM** packet sockets make no attempt to create or parse the IEEE 802.2 LLC header for a IEEE 802.3 frame. When **ETH_P_802_3** is specified as protocol for sending the kernel creates the 802.3 frame and fills out the length field; the user has to supply the LLC header to get a fully conforming packet. Incoming 802.3 packets are not multiplexed on the DSAP/SSAP protocol fields; instead they are supplied to the user as protocol **ETH_P_802_2** with the LLC header prepended. It is thus not possible to bind to **ETH_P_802_3;** bind to **ETH_P_802_2** instead and do the protocol multiplex yourself. The default for sending is the standard Ethernet DIX encapsulation with the protocol filled in.

Packet sockets are not subject to the input or output firewall chains.

ERRORS

ENETDOWN
Interface is not up.

ENOTCONN
No interface address passed.

ENODEV
Unknown device name or interface index specified in interface address.

EMSGSIZE
Packet is bigger than interface MTU.

ENOBUFS
Not enough memory to allocate the packet.

EFAULT
User passed invalid memory address.

EINVAL
Invalid argument.

ENXIO
Interface address contained illegal interface index.

EPERM
User has insufficient privileges to carry out this operation.

EADDRNOTAVAIL
Unknown multicast group address passed.

ENOENT
No packet received.

In addition other errors may be generated by the low-level driver.

VERSIONS

PF_PACKET is a new feature in Linux 2.2. Earlier Linux versions supported only **SOCK_PACKET.**

BUGS

glibc 2.1 does not have a define for **SOL_PACKET.** The suggested workaround is to use

```
#ifndef SOL_PACKET
#define SOL_PACKET 263
#endif
```

This is fixed in later glibc versions and also does not occur on libc5 systems.

The IEEE 802.2/803.3 LLC handling could be considered as a bug.

Socket filters are not documented.

The *MSG_TRUNC* recvmsg extension is an ugly hack and should be replaced by a control message. There is currently no way to get the original destination address of packets via SOCK_DGRAM.

CREDITS

This man page was written by Andi Kleen with help from Matthew Wilcox. PF_PACKET in Linux 2.2 was implemented by Alexey Kuznetsov, based on code by Alan Cox and others.

SEE ALSO

ip(7), **socket**(7), **socket**(2), **raw**(7), **pcap**(3)

RFC 894 for the standard IP Ethernet encapsulation.

RFC 1700 for the IEEE 802.3 IP encapsulation.

The <*linux/if_ether.h*> include file for physical layer protocols.

NAME

write − write to a file descriptor

SYNOPSIS

#include <unistd.h>

ssize_t write(int *fd***, const void *** *buf***, size_t** *count***);**

DESCRIPTION

write writes up to *count* bytes to the file referenced by the file descriptor *fd* from the buffer starting at *buf*. POSIX requires that a **read()** which can be proved to occur after a **write()** has returned returns the new data. Note that not all file systems are POSIX conforming.

RETURN VALUE

On success, the number of bytes written are returned (zero indicates nothing was written). On error, −1 is returned, and *errno* is set appropriately. If *count* is zero and the file descriptor refers to a regular file, 0 will be returned without causing any other effect. For a special file, the results are not portable.

ERRORS

EBADF
fd is not a valid file descriptor or is not open for writing.

EINVAL
fd is attached to an object which is unsuitable for writing.

EFAULT
buf is outside your accessible address space.

EFBIG
An attempt was made to write a file that exceeds the implementation-defined maximum file size or the process' file size limit, or to write at a position past than the maximum allowed offset.

EPIPE *fd* is connected to a pipe or socket whose reading end is closed. When this happens the writing process will receive a **SIGPIPE** signal; if it catches, blocks or ignores this the error **EPIPE** is returned.

EAGAIN
Non-blocking I/O has been selected using **O_NONBLOCK** and the write would block.

EINTR
The call was interrupted by a signal before any data was written.

ENOSPC
The device containing the file referred to by *fd* has no room for the data.

EIO A low-level I/O error occurred while modifying the inode.

Other errors may occur, depending on the object connected to *fd*.

CONFORMING TO

SVr4, SVID, POSIX, X/OPEN, 4.3BSD. SVr4 documents additional error conditions EDEADLK, ENOLCK, ENOLNK, ENOSR, ENXIO, EPIPE, or ERANGE. Under SVr4 a write may be interrupted and return EINTR at any point, not just before any data is written.

NOTES

A successful return from **write** does not make any guarantee that data has been committed to disk. In fact, on some buggy implementations, it does not even guarantee that space has successfully been reserved for the data. The only way to be sure is to call **fsync**(2) after you are done writing all your data.

SEE ALSO

close(2), **fcntl**(2), **fsync**(2), **ioctl**(2), **lseek**(2), **open**(2), **read**(2), **select**(2), **fwrite**(3), **writev**(3)

NAME

strlen − calculate the length of a string

SYNOPSIS

#include <string.h>

size_t strlen(const char * *s***);**

DESCRIPTION

The **strlen()** function calculates the length of the string *s*, not including the terminating '\0' character.

RETURN VALUE

The **strlen()** function returns the number of characters in *s*.

CONFORMING TO

SVID 3, POSIX, BSD 4.3, ISO 9899

SEE ALSO

string(3)

NAME

printf, fprintf, sprintf, snprintf, vprintf, vfprintf, vsprintf, vsnprintf − formatted output conversion

SYNOPSIS

#include <stdio.h>

int printf(const char * *format***, ...);**
int fprintf(FILE * *stream***, const char *** *format***, ...);**
int sprintf(char * *str***, const char *** *format***, ...);**
int snprintf(char * *str***, size_t** *size***, const char *** *format***, ...);**

#include <stdarg.h>

int vprintf(const char * *format***, va_list** *ap***);**
int vfprintf(FILE * *stream***, const char *** *format***, va_list** *ap***);**
int vsprintf(char * *str***, const char *** *format***, va_list** *ap***);**
int vsnprintf(char * *str***, size_t** *size***, const char *** *format***, va_list** *ap***);**

DESCRIPTION

The functions in the **printf** family produce output according to a *format* as described below. The functions **printf** and **vprintf** write output to *stdout*, the standard output stream; **fprintf** and **vfprintf** write output to the given output *stream*; **sprintf**, **snprintf**, **vsprintf** and **vsnprintf** write to the character string *str*.

The functions **vprintf**, **vfprintf**, **vsprintf**, **vsnprintf** are equivalent to the functions **printf**, **fprintf**, **sprintf**, **snprintf**, respectively, except that they are called with a va_list instead of a variable number of arguments. These functions do not call the *va_end* macro. Consequently, the value of *ap* is undefined after the call. The application should call *va_end(ap)* itself afterwards.

These eight functions write the output under the control of a *format* string that specifies how subsequent arguments (or arguments accessed via the variable-length argument facilities of **stdarg**(3)) are converted for output.

Return value

These functions return the number of characters printed (not including the trailing '\0' used to end output to strings). **snprintf** and **vsnprintf** do not write more than *size* bytes (including the trailing '\0'), and return −1 if the output was truncated due to this limit. (Thus until glibc 2.0.6. Since glibc 2.1 these functions follow the C99 standard and return the number of characters (excluding the trailing '\0') which would have been written to the final string if enough space had been available.)

Format of the format string

The format string is a character string, beginning and ending in its initial shift state, if any. The format string is composed of zero or more directives: ordinary characters (not **%**), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character **%**, and ends with a *conversion specifier*. In between there may be (in this order) zero or more *flags*, an optional minimum *field width*, an optional *precision* and an optional *length modifier*.

The arguments must correspond properly (after type promotion) with the conversion specifier. By default, the arguments are used in the order given, where each '*' and each conversion specifier asks for the next argument (and it is an error if insufficiently many arguments are given). One can also specify explicitly which argument is taken, at each place where an argument is required, by writing '%m$' instead of '%' and '*m$' instead of '*', where the decimal integer m denotes the position in the argument list of the desired argument, indexed starting from 1. Thus,

```
printf("%*d", width, num);
```

and

```
printf("%2$*1$d", width, num);
```

are equivalent. The second style allows repeated references to the same argument. The C99 standard does not include the style using '$', which comes from the Single Unix Specification. If the style using '$' is

used, it must be used throughout for all conversions taking an argument and all width and precision arguments, but it may be mixed with '%%' formats which do not consume an argument. There may be no gaps in the numbers of arguments specified using '$'; for example, if arguments 1 and 3 are specified, argument 2 must also be specified somewhere in the format string.

For some numeric conversions a radix character ('decimal point') or thousands' grouping character is used. The actual character used depends on the LC_NUMERIC part of the locale. The POSIX locale uses '.' as radix character, and does not have a grouping character. Thus,

```
printf("%'.2f", 1234567.89);
```

results in '1234567.89' in the POSIX locale, in '1234567,89' in the nl_NL locale, and in '1.234.567,89' in the da_DK locale.

The flag characters

The character % is followed by zero or more of the following flags:

The value should be converted to an ''alternate form''. For **o** conversions, the first character of the output string is made zero (by prefixing a 0 if it was not zero already). For **x** and **X** conversions, a non−zero result has the string '0x' (or '0X' for **X** conversions) prepended to it. For **a**, **A**, **e**, **E**, **f**, **F**, **g**, and **G** conversions, the result will always contain a decimal point, even if no digits follow it (normally, a decimal point appears in the results of those conversions only if a digit follows). For **g** and **G** conversions, trailing zeros are not removed from the result as they would otherwise be. For other conversions, the result is undefined.

0 The value should be zero padded. For **d**, **i**, **o**, **u**, **x**, **X**, **a**, **A**, **e**, **E**, **f**, **F**, **g**, and **G** conversions, the converted value is padded on the left with zeros rather than blanks. If the **0** and − flags both appear, the **0** flag is ignored. If a precision is given with a numeric conversion (**d**, **i**, **o**, **u**, **x**, and **X**), the **0** flag is ignored. For other conversions, the behavior is undefined.

− The converted value is to be left adjusted on the field boundary. (The default is right justification.) Except for **n** conversions, the converted value is padded on the right with blanks, rather than on the left with blanks or zeros. A − overrides a **0** if both are given.

' ' (a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.

+ A sign (+ or -) always be placed before a number produced by a signed conversion. By default a sign is used only for negative numbers. A **+** overrides a space if both are used.

The five flag characters above are defined in the C standard. The SUSv2 specifies one further flag character.

' For decimal conversion (**i**, **d**, **u**, **f**, **F**, **g**, **G**) the output is to be grouped with thousands' grouping characters if the locale information indicates any. Note that many versions of **gcc** cannot parse this option and will issue a warning. SUSv2 does not include %'F.

glibc 2.2 adds one further flag character.

I For decimal integer conversion (**i**, **d**, **u**) the output uses the locale's alternative output digits, if any (for example, Arabic digits). However, it does not include any locale definitions with such **outdigits** defined.

The field width

An optional decimal digit string (with nonzero first digit) specifying a minimum field width. If the converted value has fewer characters than the field width, it will be padded with spaces on the left (or right, if the left-adjustment flag has been given). Instead of a decimal digit string one may write '*' or '*m$' (for some decimal integer m) to specify that the field width is given in the next argument, or in the m-th argument, respectively, which must be of type *int*. A negative field width is taken as a '-' flag followed by a positive field width. In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

The precision

An optional precision, in the form of a period ('.') followed by an optional decimal digit string. Instead of a decimal digit string one may write '*' or '*m$' (for some decimal integer m) to specify that the precision is given in the next argument, or in the m-th argument, respectively, which must be of type *int*. If the precision is given as just '.', or the precision is negative, the precision is taken to be zero. This gives the minimum number of digits to appear for **d**, **i**, **o**, **u**, **x**, and **X** conversions, the number of digits to appear after the radix character for **a**, **A**, **e**, **E**, **f**, and **F** conversions, the maximum number of significant digits for **g** and **G** conversions, or the maximum number of characters to be printed from a string for **s** and **S** conversions.

The length modifier

Here, 'integer conversion' stands for **d**, **i**, **o**, **u**, **x**, or **X** conversion.

hh A following integer conversion corresponds to a *signed char* or *unsigned char* argument, or a following **n** conversion corresponds to a pointer to a *signed char* argument.

h A following integer conversion corresponds to a *short int* or *unsigned short int* argument, or a following **n** conversion corresponds to a pointer to a *short int* argument.

l (ell) A following integer conversion corresponds to a *long int* or *unsigned long int* argument, or a following **n** conversion corresponds to a pointer to a *long int* argument, or a following **c** conversion corresponds to a *wint_t* argument, or a following **s** conversion corresponds to a pointer to *wchar_t* argument.

ll (ell-ell). A following integer conversion corresponds to a *long long int* or *unsigned long long int* argument, or a following **n** conversion corresponds to a pointer to a *long long int* argument.

L A following **a**, **A**, **e**, **E**, **f**, **F**, **g**, or **G** conversion corresponds to a *long double* argument. (C99 allows %LF, but SUSv2 does not.)

q ('quad'. BSD 4.4 and Linux libc5 only. Don't use.) This is a synonym for **ll**.

j A following integer conversion corresponds to an *intmax_t* or *uintmax_t* argument.

z A following integer conversion corresponds to a *size_t* or *ssize_t* argument. (Linux libc5 has **Z** with this meaning. Don't use it.)

t A following integer conversion corresponds to a *ptrdiff_t* argument.

The SUSv2 only knows about the length modifiers **h** (in **hd**, **hi**, **ho**, **hx**, **hX**, **hn**) and **l** (in **ld**, **li**, **lo**, **lx**, **lX**, **ln**, **lc**, **ls**) and **L** (in **Le**, **LE**, **Lf**, **Lg**, **LG**).

The conversion specifier

A character that specifies the type of conversion to be applied. The conversion specifiers and their meanings are:

d,**i** The *int* argument is converted to signed decimal notation. The precision, if any, gives the minimum number of digits that must appear; if the converted value requires fewer digits, it is padded on the left with zeros. The default precision is 1. When 0 is printed with an explicit precision 0, the output is empty.

o,**u**,**x**,**X** The *unsigned int* argument is converted to unsigned octal (**o**), unsigned decimal (**u**), or unsigned hexadecimal (**x** and **X**) notation. The letters **abcdef** are used for **x** conversions; the letters **ABCDEF** are used for **X** conversions. The precision, if any, gives the minimum number of digits that must appear; if the converted value requires fewer digits, it is padded on the left with zeros. The default precision is 1. When 0 is printed with an explicit precision 0, the output is empty.

e,**E** The *double* argument is rounded and converted in the style [−]**d.**ddd**e**±dd where there is one digit before the decimal−point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero, no decimal−point character appears. An **E** conversion uses the letter **E** (rather than **e**) to introduce the exponent. The exponent always contains at least two digits; if the value is zero, the exponent is 00.

f,**F** The *double* argument is rounded and converted to decimal notation in the style [−]ddd**.**ddd, where the number of digits after the decimal−point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is explicitly zero, no decimal−point character appears. If a decimal point appears, at least one digit appears before it.

(The SUSv2 does not know about **F** and says that character string representations for infinity and NaN may be made available. The C99 standard specifies '[-]inf' or '[-]infinity' for infinity, and a string starting with 'nan' for NaN, in the case of **f** conversion, and '[-]INF' or '[-]INFINITY' or 'NAN*' in the case of **F** conversion.)

g,**G** The *double* argument is converted in style **f** or **e** (or **F** or **E** for **G** conversions). The precision specifies the number of significant digits. If the precision is missing, 6 digits are given; if the precision is zero, it is treated as 1. Style **e** is used if the exponent from its conversion is less than −4 or greater than or equal to the precision. Trailing zeros are removed from the fractional part of the result; a decimal point appears only if it is followed by at least one digit.

a,**A** (C99; not in SUSv2) For **a** conversion, the *double* argument is converted to hexadecimal notation (using the letters abcdef) in the style [-]**0x**h**.**hhhh**p**±d; for **A** conversion the prefix **0X**, the letters ABCDEF, and the exponent separator **P** is used. There is one hexadecimal digit before the decimal point, and the number of digits after it is equal to the precision. The default precision suffices for an exact representation of the value if an exact representation in base 2 exists and otherwise is sufficiently large to distinguish values of type *double*. The digit before the decimal point is unspecified for non-normalized numbers, and nonzero but otherwise unspecified for normalized numbers.

c If no **l** modifier is present, the *int* argument is converted to an *unsigned char*, and the resulting character is written. If an **l** modifier is present, the *wint_t* (wide character) argument is converted to a multibyte sequence by a call to the **wcrtomb** function, with a conversion state starting in the initial state, and the resulting multibyte string is written.

s If no **l** modifier is present: The *const char* * argument is expected to be a pointer to an array of character type (pointer to a string). Characters from the array are written up to (but not including) a terminating **NUL** character; if a precision is specified, no more than the number specified are written. If a precision is given, no null character need be present; if the precision is not specified, or is greater than the size of the array, the array must contain a terminating **NUL** character.

If an **l** modifier is present: The *const wchar_t* * argument is expected to be a pointer to an array of wide characters. Wide characters from the array are converted to multibyte characters (each by a call to the **wcrtomb** function, with a conversion state starting in the initial state before the first wide character), up to and including a terminating null wide character. The resulting multibyte characters are written up to (but not including) the terminating null byte. If a precision is specified, no more bytes than the number specified are written, but no partial multibyte characters are written. Note that the precision determines the number of *bytes* written, not the number of *wide characters* or *screen positions*. The array must contain a terminating null wide character, unless a precision is given and it is so small that the number of bytes written exceeds it before the end of the array is reached.

C (Not in C99, but in SUSv2.) Synonym for **lc**. Don't use.

S (Not in C99, but in SUSv2.) Synonym for **ls**. Don't use.

p The *void* * pointer argument is printed in hexadecimal (as if by **%#x** or **%#lx**).

n The number of characters written so far is stored into the integer indicated by the *int* * (or variant) pointer argument. No argument is converted.

% A '%' is written. No argument is converted. The complete conversion specification is '%%'.

EXAMPLES

To print pi to five decimal places:

```
#include <math.h>
#include <stdio.h>
fprintf(stdout, "pi = %.5f\n", 4 * atan(1.0));
```

To print a date and time in the form 'Sunday, July 3, 10:02', where *weekday* and *month* are pointers to strings:

```
#include <stdio.h>
fprintf(stdout, "%s, %s %d, %.2d:%.2d\n",
        weekday, month, day, hour, min);
```

Many countries use the day-month-year order. Hence, an internationalized version must be able to print the arguments in an order specified by the format:

```
#include <stdio.h>
fprintf(stdout, format,
        weekday, month, day, hour, min);
```

where *format* depends on locale, and may permute the arguments. With the value

```
"%1$s, %3$d. %2$s, %4$d:%5$.2d\n"
```

one might obtain 'Sonntag, 3. Juli, 10:02'.

To allocate a sufficiently large string and print into it (code correct for both glibc 2.0 and glibc 2.1):

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
char *
make_message(const char *fmt, ...) {
  /* Guess we need no more than 100 bytes. */
  int n, size = 100;
  char *p;
  va_list ap;
  if ((p = malloc (size)) == NULL)
    return NULL;
  while (1) {
    /* Try to print in the allocated space. */
    va_start(ap, fmt);
    n = vsnprintf (p, size, fmt, ap);
    va_end(ap);
    /* If that worked, return the string. */
    if (n > -1 && n < size)
      return p;
    /* Else try again with more space. */
    if (n > -1)    /* glibc 2.1 */
      size = n+1; /* precisely what is needed */
    else           /* glibc 2.0 */
      size *= 2;  /* twice the old size */
    if ((p = realloc (p, size)) == NULL)
      return NULL;
  }
}
```

CONFORMING TO

The **fprintf**, **printf**, **sprintf**, **vprintf**, **vfprintf**, and **vsprintf** functions conform to ANSI X3.159-1989 (''ANSI C'') and ISO/IEC 9899:1999 (''ISO C99''). The **snprintf** and **vsnprintf** functions conform to ISO/IEC 9899:1999.

Concerning the return value of **snprintf**, the SUSv2 and the C99 standard contradict each other: when **snprintf** is called with *size*=0 then SUSv2 stipulates an unspecified return value less than 1, while C99 allows *str* to be NULL in this case, and gives the return value (as always) as the number of characters that would have been written in case the output string has been large enough.

Linux libc4 knows about the five C standard flags. It knows about the length modifiers h,l,L, and the conversions cdeEfFgGinopsuxX, where F is a synonym for f. Additionally, it accepts D,O,U as synonyms for ld,lo,lu. (This is bad, and caused serious bugs later, when support for %D disappeared.) No locale-dependent radix character, no thousands' separator, no NaN or infinity, no %m$ and *m$.

Linux libc5 knows about the five C standard flags and the ' flag, locale, %m$ and *m$. It knows about the length modifiers h,l,L,Z,q, but accepts L and q both for long doubles and for long long integers (this is a bug). It no longer recognizes FDOU, but adds a new conversion character **m**, which outputs *strerror(errno)*.

glibc 2.0 adds conversion characters C and S.

glibc 2.1 adds length modifiers hh,j,t,z and conversion characters a,A.

glibc 2.2 adds the conversion character F with C99 semantics, and the flag character I.

HISTORY

Unix V7 defines the three routines **printf**, **fprintf**, **sprintf**, and has the flag -, the width or precision *, the length modifier l, and the conversions doxfegcsu, and also D,O,U,X as synonyms for ld,lo,lu,lx. This is still true for BSD 2.9.1, but BSD 2.10 has the flags #, + and <space> and no longer mentions D,O,U,X. BSD 2.11 has **vprintf**, **vfprintf**, **vsprintf**, and warns not to use D,O,U,X. BSD 4.3 Reno has the flag 0, the length modifiers h and L, and the conversions n, p, E, G, X (with current meaning) and deprecates D,O,U. BSD 4.4 introduces the functions **snprintf** and **vsnprintf**, and the length modifier q. FreeBSD also has functions *asprintf* and *vasprintf* , that allocate a buffer large enough for **sprintf**. In glibc there are functions *dprintf* and *vdprintf* that print to a file descriptor instead of a stream.

BUGS

Because **sprintf** and **vsprintf** assume an arbitrarily long string, callers must be careful not to overflow the actual space; this is often impossible to assure. Note that the length of the strings produced is locale-dependent and difficult to predict. Use **snprintf** and **vsnprintf** instead (or **asprintf** and **vasprintf**).

Linux libc4.[45] does not have a **snprintf**, but provides a libbsd that contains an **snprintf** equivalent to **sprintf**, i.e., one that ignores the *size* argument. Thus, the use of **snprintf** with early libc4 leads to serious security problems.

Code such as **printf(** *foo***);** often indicates a bug, since *foo* may contain a % character. If *foo* comes from untrusted user input, it may contain %n, causing the **printf** call to write to memory and creating a security hole.

Some floating point conversions under early libc4 caused memory leaks.

SEE ALSO

printf(1), **asprintf**(3), **dprintf**(3), **wcrtomb**(3), **wprintf**(3), **scanf**(3), **locale**(5)

NAME

shutdown − shut down part of a full-duplex connection

SYNOPSIS

#include <sys/socket.h>

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

DESCRIPTION

The **shutdown** call causes all or part of a full-duplex connection on the socket associated with *s* to be shut down. If *how* is **SHUT_RD**, further receptions will be disallowed. If *how* is **SHUT_WR**, further transmissions will be disallowed. If *how* is **SHUT_RDWR**, further receptions and transmissions will be disallowed.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set appropriately.

ERRORS

EBADF

s is not a valid descriptor.

ENOTSOCK

s is a file, not a socket.

ENOTCONN

The specified socket is not connected.

NOTES

The constants SHUT_RD, SHUT_WR, SHUT_RDWR have the value 0, 1, 2, respectively, and are defined in <*sys/socket.h*> since glibc-2.1.91.

CONFORMING TO

4.4BSD (the **shutdown** function call first appeared in 4.2BSD).

SEE ALSO

connect(2), **socket**(2)

NAME

read − read from a file descriptor

SYNOPSIS

#include <unistd.h>

ssize_t read(int *fd***, void ****buf***, size_t** *count***);**

DESCRIPTION

read() attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*.

If *count* is zero, **read()** returns zero and has no other results. If *count* is greater than SSIZE_MAX, the result is unspecified.

RETURN VALUE

On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. It is not an error if this number is smaller than the number of bytes requested; this may happen for example because fewer bytes are actually available right now (maybe because we were close to end-of-file, or because we are reading from a pipe, or from a terminal), or because **read()** was interrupted by a signal. On error, −1 is returned, and *errno* is set appropriately. In this case it is left unspecified whether the file position (if any) changes.

ERRORS

EINTR
The call was interrupted by a signal before any data was read.

EAGAIN
Non-blocking I/O has been selected using **O_NONBLOCK** and no data was immediately available for reading.

EIO I/O error. This will happen for example when the process is in a background process group, tries to read from its controlling tty, and either it is ignoring or blocking SIGTTIN or its process group is orphaned. It may also occur when there is a low-level I/O error while reading from a disk or tape.

EISDIR
fd refers to a directory.

EBADF
fd is not a valid file descriptor or is not open for reading.

EINVAL
fd is attached to an object which is unsuitable for reading.

EFAULT
buf is outside your accessible address space.

Other errors may occur, depending on the object connected to *fd*. POSIX allows a **read** that is interrupted after reading some data to return −1 (with *errno* set to EINTR) or to return the number of bytes already read.

CONFORMING TO

SVr4, SVID, AT&T, POSIX, X/OPEN, BSD 4.3

RESTRICTIONS

On NFS file systems, reading small amounts of data will only update the time stamp the first time, subsequent calls may not do so. This is caused by client side attribute caching, because most if not all NFS clients leave atime updates to the server and client side reads satisfied from the client's cache will not cause atime updates on the server as there are no server side reads. UNIX semantics can be obtained by disabling client side attribute caching, but in most situations this will substantially increase server load and decrease performance.

Many filesystems and disks were considered to be fast enough that the implementation of **O_NONBLOCK** was deemed unneccesary. So, O_NONBLOCK may not be available on files and/or disks.

SEE ALSO

close(2), **fcntl**(2), **ioctl**(2), **lseek**(2), **readdir**(2), **readlink**(2), **select**(2), **write**(2), **fread**(3), **readv**(3)

NAME

gethostbyname, gethostbyaddr, sethostent, endhostent, herror, hstrerror − get network host entry

SYNOPSIS

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

struct hostent *gethostbyname(const char *name);

#include <sys/socket.h>       /* for AF_INET */
struct hostent *gethostbyaddr(const char *addr,
  int len, int type);

void sethostent(int stayopen);

void endhostent(void);

void herror(const char *s);

const char *hstrerror(int err);


/* GNU extensions */
struct hostent *gethostbyname2(const char *name, int af);

int gethostbyname_r (const char *name,
  struct hostent *ret, char *buf, size_t buflen,
  struct hostent **result, int *h_errnop);

int gethostbyname2_r (const char *name, int af,
  struct hostent *ret, char *buf, size_t buflen,
  struct hostent **result, int *h_errnop);
```

DESCRIPTION

The **gethostbyname()** function returns a structure of type *hostent* for the given host *name*. Here *name* is either a host name, or an IPv4 address in standard dot notation, or an IPv6 address in colon (and possibly dot) notation. (See RFC 1884 for the description of IPv6 addresses.) If *name* is an IPv4 or IPv6 address, no lookup is performed and **gethostbyname**() simply copies *name* into the *h_name* field and its *struct in_addr* equivalent into the *h_addr_list[0]* field of the returned *hostent* structure. If *name* doesn't end in a dot and the environment variable **HOSTALIASES** is set, the alias file pointed to by **HOSTALIASES** will first be searched for *name* (see **hostname**(7) for the file format). The current domain and its parents are searched unless *name* ends in a dot.

The **gethostbyaddr()** function returns a structure of type *hostent* for the given host address *addr* of length *len* and address type *type*. The only valid address type is currently **AF_INET**.

The **sethostent()** function specifies, if *stayopen* is true (1), that a connected TCP socket should be used for the name server queries and that the connection should remain open during successive queries. Otherwise, name server queries will use UDP datagrams.

The **endhostent()** function ends the use of a TCP connection for name server queries.

The (obsolete) **herror()** function prints the error message associated with the current value of *h_errno* on stderr.

The (obsolete) **hstrerror()** function takes an error number (typically *h_errno*) and returns the corresponding message string.

The domain name queries carried out by **gethostbyname()** and **gethostbyaddr()** use a combination of any

or all of the name server **named**(8), a broken out line from */etc/hosts*, and the Network Information Service (NIS or YP), depending upon the contents of the *order* line in */etc/host.conf*. (See **resolv+**(8)). The default action is to query **named**(8), followed by */etc/hosts*.

The *hostent* structure is defined in <*netdb.h*> as follows:

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

The members of the *hostent* structure are:

h_name
The official name of the host.

h_aliases
A zero-terminated array of alternative names for the host.

h_addrtype
The type of address; always **AF_INET** at present.

h_length
The length of the address in bytes.

h_addr_list
A zero-terminated array of network addresses for the host in network byte order.

h_addr The first address in *h_addr_list* for backward compatibility.

RETURN VALUE

The **gethostbyname()** and **gethostbyaddr()** functions return the *hostent* structure or a NULL pointer if an error occurs. On error, the *h_errno* variable holds an error number.

ERRORS

The variable *h_errno* can have the following values:

HOST_NOT_FOUND
The specified host is unknown.

NO_ADDRESS or **NO_DATA**
The requested name is valid but does not have an IP address.

NO_RECOVERY
A non-recoverable name server error occurred.

TRY_AGAIN
A temporary error occurred on an authoritative name server. Try again later.

FILES

/etc/host.conf
resolver configuration file

/etc/hosts
host database file

CONFORMING TO

BSD 4.3.

NOTES

The SUS-v2 standard is buggy and declares the *len* parameter of **gethostbyaddr()** to be of type *size_t*. (That is wrong, because it has to be *int*, and *size_t* is not. POSIX 1003.1-2001 makes it *socklen_t*, which is OK.)

The functions **gethostbyname**() and **gethostbyaddr**() may return pointers to static data, which may be overwritten by later calls. Copying the *struct hostent* does not suffice, since it contains pointers - a deep copy is required.

Glibc2 also has a **gethostbyname2()** that works like **gethostbyname()**, but permits to specify the address family to which the address must belong.

Glibc2 also has reentrant versions **gethostbyname_r()** and **gethostbyname2_r()**. These return 0 on success and nonzero on error. The result of the call is now stored in the struct with address *ret*. After the call, **result* will be NULL on error or point to the result on success. Auxiliary data is stored in the buffer *buf* of length *buflen*. (If the buffer is too small, these functions will return **ERANGE**.) No global variable *h_errno* is modified, but the address of a variable in which to store error numbers is passed in *h_errnop*.

POSIX 1003.1-2001 marks **gethostbyaddr()** and **gethostbyname()** legacy, and introduces

struct hostent *getipnodebyaddr (const void *restrict *addr***,**
socklen_t *len***, int** *type***, int *restrict** *error_num***);**

struct hostent *getipnodebyname (const char ***_name_,**
int *type***, int** *flags***, int ***_error_num_**);**

SEE ALSO

resolver(3), **hosts**(5), **hostname**(7), **resolv+**(8), **named**(8)

NAME

fopen, fdopen, freopen − stream open functions

SYNOPSIS

#include <stdio.h>

FILE *fopen(const char * *path***, const char *** *mode***);**
FILE *fdopen(int *fildes***, const char *** *mode***);**
FILE *freopen(const char * *path***, const char *** *mode***, FILE *** *stream***);**

DESCRIPTION

The **fopen** function opens the file whose name is the string pointed to by *path* and associates a stream with it.

The argument *mode* points to a string beginning with one of the following sequences (Additional characters may follow these sequences.):

r Open text file for reading. The stream is positioned at the beginning of the file.

r+ Open for reading and writing. The stream is positioned at the beginning of the file.

w Truncate file to zero length or create text file for writing. The stream is positioned at the beginning of the file.

w+ Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. The stream is positioned at the beginning of the file.

a Open for appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.

a+ Open for reading and appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.

The *mode* string can also include the letter ''b'' either as a last character or as a character between the characters in any of the two-character strings described above. This is strictly for compatibility with ANSI X3.159-1989 (''ANSI C'') and has no effect; the ''b'' is ignored on all POSIX conforming systems, including Linux. (Other systems may treat text files and binary files differently, and adding the ''b'' may be a good idea if you do I/O to a binary file and expect that your program may be ported to non-Unix environments.)

Any created files will have mode **S_IRUSR**|**S_IWUSR**|**S_IRGRP**|**S_IWGRP**|**S_IROTH**|**S_IWOTH** (0666), as modified by the process' umask value (see **umask**(2).

Reads and writes may be intermixed on read/write streams in any order. Note that ANSI C requires that a file positioning function intervene between output and input, unless an input operation encounters end-of-file. (If this condition is not met, then a read is allowed to return the result of writes other than the most recent.) Therefore it is good practice (and indeed sometimes necessary under Linux) to put an **fseek** or **fgetpos** operation between write and read operations on such a stream. This operation may be an apparent no-op (as in *fseek(..., 0L, SEEK_CUR)* called for its synchronizing side effect.

Opening a file in append mode (**a** as the first character of *mode*) causes all subsequent write operations to this stream to occur at end-of-file, as if preceded by an

```
fseek(stream,0,SEEK_END);
```

call.

The **fdopen** function associates a stream with the existing file descriptor, *fildes*. The *mode* of the stream (one of the values "r", "r+", "w", "w+", "a", "a+") must be compatible with the mode of the file descriptor. The file position indicator of the new stream is set to that belonging to *fildes*, and the error and end-of-file indicators are cleared. Modes "w" or "w+" do not cause truncation of the file. The file descriptor is not dup'ed, and will be closed when the stream created by **fdopen** is closed. The result of applying **fdopen** to a shared memory object is undefined.

The **freopen** function opens the file whose name is the string pointed to by *path* and associates the stream

pointed to by *stream* with it. The original stream (if it exists) is closed. The *mode* argument is used just as in the **fopen** function. The primary use of the **freopen** function is to change the file associated with a standard text stream (*stderr*, *stdin*, or *stdout*).

RETURN VALUE

Upon successful completion **fopen**, **fdopen** and **freopen** return a **FILE** pointer. Otherwise, **NULL** is returned and the global variable *errno* is set to indicate the error.

ERRORS

EINVAL

The *mode* provided to **fopen**, **fdopen**, or **freopen** was invalid.

The **fopen**, **fdopen** and **freopen** functions may also fail and set *errno* for any of the errors specified for the routine **malloc**(3).

The **fopen** function may also fail and set *errno* for any of the errors specified for the routine **open**(2).

The **fdopen** function may also fail and set *errno* for any of the errors specified for the routine **fcntl**(2).

The **freopen** function may also fail and set *errno* for any of the errors specified for the routines **open**(2), **fclose**(3) and **fflush**(3).

CONFORMING TO

The **fopen** and **freopen** functions conform to ANSI X3.159-1989 (''ANSI C''). The **fdopen** function conforms to IEEE Std1003.1-1988 (''POSIX.1'').

SEE ALSO

open(2), **fclose**(3), **fileno**(3)

NAME

connect − initiate a connection on a socket

SYNOPSIS

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

int connect(int *sockfd***, const struct sockaddr ****serv_addr***, socklen_t** *addrlen***);**

DESCRIPTION

The file descriptor *sockfd* must refer to a socket. If the socket is of type **SOCK_DGRAM** then the *serv_addr* address is the address to which datagrams are sent by default, and the only address from which datagrams are received. If the socket is of type **SOCK_STREAM** or **SOCK_SEQPACKET**, this call attempts to make a connection to another socket. The other socket is specified by *serv_addr*, which is an address (of length *addrlen*) in the communications space of the socket. Each communications space interprets the *serv_addr* parameter in its own way.

Generally, connection-based protocol sockets may successfully **connect** only once; connectionless protocol sockets may use **connect** multiple times to change their association. Connectionless sockets may dissolve the association by connecting to an address with the *sa_family* member of **sockaddr** set to **AF_UNSPEC**.

RETURN VALUE

If the connection or binding succeeds, zero is returned. On error, −1 is returned, and *errno* is set appropriately.

ERRORS

The following are general socket errors only. There may be other domain-specific error codes.

EBADF
The file descriptor is not a valid index in the descriptor table.

EFAULT
The socket structure address is outside the user's address space.

ENOTSOCK
The file descriptor is not associated with a socket.

EISCONN
The socket is already connected.

ECONNREFUSED
No one listening on the remote address.

ETIMEDOUT
Timeout while attempting connection. The server may be too busy to accept new connections. Note that for IP sockets the timeout may be very long when syncookies are enabled on the server.

ENETUNREACH
Network is unreachable.

EADDRINUSE
Local address is already in use.

EINPROGRESS
The socket is non-blocking and the connection cannot be completed immediately. It is possible to **select**(2) or **poll**(2) for completion by selecting the socket for writing. After **select** indicates writability, use **getsockopt**(2) to read the **SO_ERROR** option at level **SOL_SOCKET** to determine whether **connect** completed successfully (**SO_ERROR** is zero) or unsuccessfully (**SO_ERROR** is one of the usual error codes listed here, explaining the reason for the failure).

EALREADY
The socket is non-blocking and a previous connection attempt has not yet been completed.

EAGAIN
No more free local ports or insufficient entries in the routing cache. For **PF_INET** see the **net.ipv4.ip_local_port_range** sysctl in **ip**(7) on how to increase the number of local ports.

EAFNOSUPPORT
The passed address didn't have the correct address family in its *sa_family* field.

EACCES, EPERM
The user tried to connect to a broadcast address without having the socket broadcast flag enabled or the connection request failed because of a local firewall rule.

CONFORMING TO

SVr4, 4.4BSD (the **connect** function first appeared in BSD 4.2). SVr4 documents the additional general error codes **EADDRNOTAVAIL**, **EINVAL**, **EAFNOSUPPORT**, **EALREADY**, **EINTR**, **EPROTOTYPE**, and **ENOSR**. It also documents many additional error conditions not described here.

NOTE

The third argument of **connect** is in reality an int (and this is what BSD 4.* and libc4 and libc5 have). Some POSIX confusion resulted in the present socklen_t. The draft standard has not been adopted yet, but glibc2 already follows it and also has socklen_t. See also **accept**(2).

BUGS

Unconnecting a socket by calling **connect** with a **AF_UNSPEC** address is not yet implemented.

SEE ALSO

accept(2), **bind**(2), **listen**(2), **socket**(2), **getsockname**(2)

NAME

bind − bind a name to a socket

SYNOPSIS

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

int bind(int *sockfd***, struct sockaddr ****my_addr***, socklen_t** *addrlen***);**

DESCRIPTION

bind gives the socket *sockfd* the local address *my_addr*. *my_addr* is *addrlen* bytes long. Traditionally, this is called "assigning a name to a socket." When a socket is created with **socket**(2), it exists in a name space (address family) but has no name assigned.

It is normally necessary to assign a local address using **bind** before a **SOCK_STREAM** socket may receive connections (see **accept**(2)).

The rules used in name binding vary between address families. Consult the manual entries in Section 7 for detailed information. For **AF_INET** see **ip**(7), for **AF_UNIX** see **unix**(7), for **AF_APPLETALK** see **ddp**(7), for **AF_PACKET** see **packet**(7), for **AF_X25** see **x25**(7) and for **AF_NETLINK** see **netlink**(7).

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set appropriately.

ERRORS

EBADF
sockfd is not a valid descriptor.

EINVAL
The socket is already bound to an address. This may change in the future: see *linux/unix/sock.c* for details.

EACCES
The address is protected, and the user is not the super-user.

ENOTSOCK
Argument is a descriptor for a file, not a socket.

The following errors are specific to UNIX domain (**AF_UNIX**) sockets:

EINVAL
The *addrlen* is wrong, or the socket was not in the **AF_UNIX** family.

EROFS
The socket inode would reside on a read-only file system.

EFAULT
my_addr points outside the user's accessible address space.

ENAMETOOLONG
my_addr is too long.

ENOENT
The file does not exist.

ENOMEM
Insufficient kernel memory was available.

ENOTDIR
A component of the path prefix is not a directory.

EACCES
Search permission is denied on a component of the path prefix.

ELOOP
Too many symbolic links were encountered in resolving *my_addr*.

BUGS

The transparent proxy options are not described.

CONFORMING TO

SVr4, 4.4BSD (the **bind** function first appeared in BSD 4.2). SVr4 documents additional **EADDRNOTAVAIL**, **EADDRINUSE**, and **ENOSR** general error conditions, and additional **EIO** and **EISDIR** Unix-domain error conditions.

NOTE

The third argument of **bind** is in reality an int (and this is what BSD 4.* and libc4 and libc5 have). Some POSIX confusion resulted in the present socklen_t. See also **accept**(2).

SEE ALSO

accept(2), **connect**(2), **listen**(2), **socket**(2), **getsockname**(2), **ip**(7), **socket**(7)