#### **NAME**

socket - create an endpoint for communication

# **SYNOPSIS**

```
#include <sys/types.h> /* See NOTES */
#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* argument 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
AF_UNIX, AF_LOCAL	Local communication	<b>unix</b> (7)
AF_INET	IPv4 Internet protocols	<b>ip</b> (7)
AF_INET6	IPv6 Internet protocols	<b>ipv6</b> (7)
AF_IPX	IPX – Novell protocols	
AF_NETLINK	Kernel user interface device	netlink(7)
AF_X25	ITU-T X.25 / ISO-8208 protocol	<b>x25</b> (7)
AF_AX25	Amateur radio AX.25 protocol	
AF_ATMPVC	Access to raw ATM PVCs	
AF_APPLETALK	Appletalk	<b>ddp</b> (7)
AF 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 input 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**.

Since Linux 2.6.27, the *type* argument serves a second purpose: in addition to specifying a socket type, it may include the bitwise OR of any of the following values, to modify the behavior of **socket**():

### SOCK\_NONBLOCK

Set the **O\_NONBLOCK** file status flag on the new open file description. Using this flag saves extra calls to **fcntl**(2) to achieve the same result.

# SOCK\_CLOEXEC

Set the close-on-exec (**FD\_CLOEXEC**) flag on the new file descriptor. See the description of the **O\_CLOEXEC** flag in **open**(2) for reasons why this may be useful.

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, in which case *protocol* can be

Linux 2009-01-19 1

specified as 0. 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 data 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 **sendto**(2) calls. Datagrams are generally received with **recvfrom**(2), which returns the next datagram along with the address of its sender.

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

An fcntl(2) F\_SETOWN operation can be used to specify a process or process group to receive a SIG-URG signal when the out-of-band data arrives or SIGPIPE signal when a SOCK\_STREAM connection breaks unexpectedly. This operation 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 FIOSETOWN or SIOCSPGRP 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**

On success, a file descriptor for the new socket is returned. On error, -1 is returned, and *errno* is set appropriately.

### **ERRORS**

#### **EACCES**

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

# **EAFNOSUPPORT**

The implementation does not support the specified address family.

#### **EINVAL**

Unknown protocol, or protocol family not available.

#### **EINVAL**

Invalid flags in type.

Linux 2009-01-19 2

#### **EMFILE**

Process file table overflow.

# **ENFILE**

The system limit on the total number of open files has been reached.

# **ENOBUFS** or **ENOMEM**

Insufficient memory is available. The socket cannot be created until sufficient resources are freed.

### **EPROTONOSUPPORT**

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

Other errors may be generated by the underlying protocol modules.

### **CONFORMING TO**

4.4BSD, POSIX.1-2001.

The **SOCK NONBLOCK** and **SOCK CLOEXEC** flags are Linux-specific.

**socket**() appeared in 4.2BSD. It is generally portable to/from non-BSD systems supporting clones of the BSD socket layer (including System V variants).

### **NOTES**

POSIX.1-2001 does not require the inclusion of *<sys/types.h>*, and this header file is not required on Linux. However, some historical (BSD) implementations required this header file, and portable applications are probably wise to include it.

The manifest constants used under 4.x BSD 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.

# **EXAMPLE**

An example of the use of **socket()** is shown in **getaddrinfo(3)**.

### **SEE ALSO**

 $accept(2), \ bind(2), \ connect(2), \ fcntl(2), \ getpeername(2), \ getsockname(2), \ getsockopt(2), \ ioctl(2), \ listen(2), \ read(2), \ recv(2), \ select(2), \ send(2), \ socketpair(2), \ write(2), \ getprotoent(3), \ ip(7), \ socket(7), \ tcp(7), \ udp(7), \ unix(7)$ 

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

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

## **COLOPHON**

This page is part of release 3.22 of the Linux *man-pages* project. A description of the project, and information about reporting bugs, can be found at http://www.kernel.org/doc/man-pages/.

Linux 2009-01-19 3