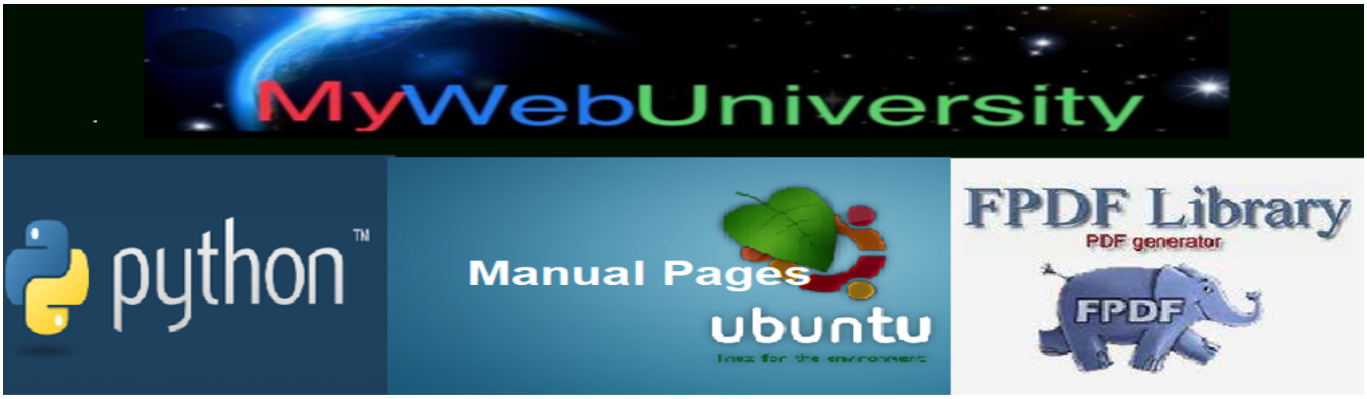

Rocky Enterprise Linux 9.2 Manual Pages on command 'IO::Socket.3perl'

$ man IO::Socket.3perl

IO::Socket(3perl) Perl Programmers Reference Guide IO::Socket(3perl)

NAME

IO::Socket - Object interface to socket communications

SYNOPSIS

```
use strict;
use warnings;
use IO::Socket qw(AF_INET AF_UNIX);
# create a new AF_INET socket
my $sock = IO::Socket->new(Domain => AF_INET);
# which is the same as
$sock = IO::Socket::INET->new();
# create a new AF_UNIX socket
$sock = IO::Socket->new(Domain => AF_UNIX);
# which is the same as
$sock = IO::Socket::UNIX->new();
```

DESCRIPTION

"IO::Socket" provides an object-oriented, IO::Handle-based interface to creating and using sockets via Socket, which provides a near one-to-one interface to the C socket library. "IO::Socket" is a base class that really only defines methods for those operations which are common to all types of sockets. Operations which are specific to a particular socket domain have methods defined in subclasses of "IO::Socket". See IO::Socket::INET, IO::Socket::UNIX, and IO::Socket::IP for examples of such a subclass.

"IO::Socket" will export all functions (and constants) defined by Socket.

CONSTRUCTOR ARGUMENTS

Given that "IO::Socket" doesn't have attributes in the traditional sense, the following arguments, rather than attributes, can be passed into the constructor.

Constructor arguments should be passed in "Key => 'Value'" pairs.

The only required argument is "Domain" in IO::Socket.

Blocking

```
my $sock = IO::Socket->new(..., Blocking => 1);
$sock = IO::Socket->new(..., Blocking => 0);
```

If defined but false, the socket will be set to non-blocking mode. If not specified it defaults to 1 (blocking mode).

Domain

```
my $sock = IO::Socket->new(Domain => IO::Socket::AF_INET);
$sock = IO::Socket->new(Domain => IO::Socket::AF_UNIX);
```

The socket domain will define which subclass of "IO::Socket" to use. The two options available along with this distribution are "AF_INET" and "AF_UNIX".

"AF_INET" is for the internet address family of sockets and is handled via IO::Socket::INET. "AF_INET" sockets are bound to an internet address and port.

"AF_UNIX" is for the unix domain socket and is handled via IO::Socket::UNIX. "AF_UNIX" sockets are bound to the file system as their address name space.

This argument is required. All other arguments are optional.

Listen

```
my $sock = IO::Socket->new(..., Listen => 5);
```

Listen should be an integer value or left unset.

If provided, this argument will place the socket into listening mode. New connections can then be accepted using the "accept" in IO::Socket method. The value given is used as the listen(2) queue size.

If the "Listen" argument is given, but false, the queue size will be set to 5.

Timeout

```
my $sock = IO::Socket->new(..., Timeout => 5);
```

The timeout value, in seconds, for this socket connection. How exactly this value is utilized is defined in the socket domain subclasses that make use of the value.

Type

```
my $sock = IO::Socket->new(..., Type => IO::Socket::SOCK_STREAM);
```

The socket type that will be used. These are usually "SOCK_STREAM", "SOCK_DGRAM", or "SOCK_RAW". If this argument is left undefined an attempt will be made to infer the type from the service name.

For example, you'll usually use "SOCK_STREAM" with a "tcp" connection and "SOCK_DGRAM" with a "udp" connection.

CONSTRUCTORS

"IO::Socket" extends the IO::Handle constructor.

new

```
my $sock = IO::Socket->new();
# get a new IO::Socket::INET instance
$sock = IO::Socket->new(Domain => IO::Socket::AF_INET);
# get a new IO::Socket::UNIX instance
$sock = IO::Socket->new(Domain => IO::Socket::AF_UNIX);
# Domain is the only required argument
$sock = IO::Socket->new(
    Domain => IO::Socket::AF_INET, # AF_INET, AF_UNIX
    Type => IO::Socket::SOCK_STREAM, # SOCK_STREAM, SOCK_DGRAM, ...
    Proto => 'tcp', # 'tcp', 'udp', IPPROTO_TCP, IPPROTO_UDP
    # and so on...
);
```

Creates an "IO::Socket", which is a reference to a newly created symbol (see the Symbol package). "new" optionally takes arguments, these arguments are defined in "CONSTRUCTOR ARGUMENTS" in IO::Socket.

Any of the "CONSTRUCTOR ARGUMENTS" in IO::Socket may be passed to the constructor, but if any arguments are provided, then one of them must be the "Domain" in IO::Socket argument.

The "Domain" in IO::Socket argument can, by default, be either "AF_INET" or "AF_UNIX".

Other domains can be used if a proper subclass for the domain family is registered. All other arguments will be passed to the "configuration" method of the package for that domain.

If the constructor fails it will return "undef" and set the $errstr package variable to contain an error message.

```
$sock = IO::Socket->new(...)
    or die "Cannot create socket - $IO::Socket::errstr\n";
```

For legacy reasons the error message is also set into the global $@ variable, and you may still find older code which looks here instead.

```
$sock = IO::Socket->new(...)
    or die "Cannot create socket - $@\n";
```

METHODS

"IO::Socket" inherits all methods from IO::Handle and implements the following new ones.

accept

```
my $client_sock = $sock->accept();
my $inet_sock = $sock->accept('IO::Socket::INET');
```

The accept method will perform the system call "accept" on the socket and return a new object. The new object will be created in the same class as the listen socket, unless a specific package name is specified. This object can be used to communicate with the client that was trying to connect.

This differs slightly from the "accept" function in perlfunc.

In a scalar context the new socket is returned, or "undef" upon failure. In a list context a two-element array is returned containing the new socket and the peer address; the list will be empty upon failure.

atmark

```
my $integer = $sock->atmark();
# read in some data on a given socket
my $data;
$sock->read($data, 1024) until $sock->atmark;
# or, export the function to use:
use IO::Socket 'sockatmark';
$sock->read($data, 1024) until sockatmark($sock);
```

True if the socket is currently positioned at the urgent data mark, false otherwise. If your system doesn't yet implement "sockatmark" this will throw an exception.

If your system does not support "sockatmark", the "use" declaration will fail at compile time.

autoflush

```
# by default, autoflush will be turned on when referenced
$sock->autoflush(); # turns on autoflush
# turn off autoflush
```

```
    $sock->autoflush(0);
    # turn on autoflush
    $sock->autoflush(1);
```

This attribute isn't overridden from IO::Handle's implementation. However, since we turn it on by default, it's worth mentioning here.

bind

```
    use Socket qw(pack_sockaddr_in);
    my $port = 3000;
    my $ip_address = '0.0.0.0';
    my $packed_addr = pack_sockaddr_in($port, $ip_address);
    $sock->bind($packed_addr);
```

Binds a network address to a socket, just as bind(2) does. Returns true if it succeeded, false otherwise. You should provide a packed address of the appropriate type for the socket.

connected

```
    my $peer_addr = $sock->connected();
    if ($peer_addr) {
        say "We're connected to $peer_addr";
    }
```

If the socket is in a connected state, the peer address is returned. If the socket is not in a connected state, "undef" is returned.

Note that this method considers a half-open TCP socket to be "in a connected state". Specifically, it does not distinguish between the ESTABLISHED and CLOSE-WAIT TCP states; it returns the peer address, rather than "undef", in either case. Thus, in general, it cannot be used to reliably learn whether the peer has initiated a graceful shutdown because in most cases (see below) the local TCP state machine remains in CLOSE-WAIT until the local application calls "shutdown" in IO::Socket or "close". Only at that point does this function return "undef".

The "in most cases" hedge is because local TCP state machine behavior may depend on the peer's socket options. In particular, if the peer socket has "SO_LINGER" enabled with a zero timeout, then the peer's "close" will generate a "RST" segment. Upon receipt of that segment, the local TCP transitions immediately to CLOSED, and in that state, this method will return "undef".

getsockopt

```
my $value = $sock->getsockopt(SOL_SOCKET, SO_REUSEADDR);
my $buf = $socket->getsockopt(SOL_SOCKET, SO_RCVBUF);
say "Receive buffer is $buf bytes";
```

Get an option associated with the socket. Levels other than "SOL_SOCKET" may be specified here. As a convenience, this method will unpack a byte buffer of the correct size back into a number.

listen

```
$sock->listen(5);
```

Does the same thing that the listen(2) system call does. Returns true if it succeeded, false otherwise. Listens to a socket with a given queue size.

peername

```
my $sockaddr_in = $sock->peername();
```

Returns the packed "sockaddr" address of the other end of the socket connection. It calls "getpeername".

protocol

```
my $proto = $sock->protocol();
```

Returns the number for the protocol being used on the socket, if known. If the protocol is unknown, as with an "AF_UNIX" socket, zero is returned.

recv

```
my $buffer = "";
my $length = 1024;
my $flags = 0; # default. optional
$sock->recv($buffer, $length);
$sock->recv($buffer, $length, $flags);
```

Similar in functionality to "recv" in perlfunc.

Receives a message on a socket. Attempts to receive $length characters of data into $buffer from the specified socket. $buffer will be grown or shrunk to the length actually read. Takes the same flags as the system call of the same name. Returns the address of the sender if socket's protocol supports this; returns an empty string otherwise. If there's an error, returns "undef". This call is actually implemented in terms of the recvfrom(2) system call.

Flags are ORed together values, such as "MSG_BCAST", "MSG_OOB", "MSG_TRUNC". The default

value for the flags is 0.

The cached value of "peername" in IO::Socket is updated with the result of "recv".

Note: In Perl v5.30 and newer, if the socket has been marked as ":utf8", "recv" will throw an exception. The ":encoding(...)" layer implicitly introduces the ":utf8" layer. See "binmode" in perlfunc.

Note: In Perl versions older than v5.30, depending on the status of the socket, either (8-bit) bytes or characters are received. By default all sockets operate on bytes, but for example if the socket has been changed using "binmode" in perlfunc to operate with the ":encoding(UTF-8)" I/O layer (see the "open" in perlfunc pragma), the I/O will operate on UTF8-encoded Unicode characters, not bytes. Similarly for the ":encoding" layer: in that case pretty much any characters can be read.

send

```
my $message = "Hello, world!";
my $flags = 0; # defaults to zero
my $to = '0.0.0.0'; # optional destination
my $sent = $sock->send($message);
$sent = $sock->send($message, $flags);
$sent = $sock->send($message, $flags, $to);
```

Similar in functionality to "send" in perlfunc.

Sends a message on a socket. Attempts to send the scalar message to the socket. Takes the same flags as the system call of the same name. On unconnected sockets, you must specify a destination to send to, in which case it does a sendto(2) syscall. Returns the number of characters sent, or "undef" on error. The sendmsg(2) syscall is currently unimplemented.

The "flags" option is optional and defaults to 0.

After a successful send with $to, further calls to "send" on an unconnected socket without $to will send to the same address, and $to will be used as the result of "peername" in IO::Socket.

Note: In Perl v5.30 and newer, if the socket has been marked as ":utf8", "send" will throw an exception. The ":encoding(...)" layer implicitly introduces the ":utf8" layer. See "binmode" in perlfunc.

Note: In Perl versions older than v5.30, depending on the status of the socket, either (8-bit) bytes or characters are sent. By default all sockets operate on bytes, but for example if the socket has been changed using "binmode" in perlfunc to operate with the

":encoding(UTF-8)" I/O layer (see the "open" in perlfunc pragma), the I/O will operate on UTF8-encoded Unicode characters, not bytes. Similarly for the ":encoding" layer: in that case pretty much any characters can be sent.

setsockopt

```
$sock->setsockopt(SOL_SOCKET, SO_REUSEADDR, 1);
$sock->setsockopt(SOL_SOCKET, SO_RCVBUF, 64*1024);
```

Set option associated with the socket. Levels other than "SOL_SOCKET" may be specified here. As a convenience, this method will convert a number into a packed byte buffer.

shutdown

```
$sock->shutdown(SHUT_RD); # we stopped reading data
$sock->shutdown(SHUT_WR); # we stopped writing data
$sock->shutdown(SHUT_RDWR); # we stopped using this socket
```

Shuts down a socket connection in the manner indicated by the value passed in, which has the same interpretation as in the syscall of the same name.

This is useful with sockets when you want to tell the other side you're done writing but not done reading, or vice versa. It's also a more insistent form of "close" because it also disables the file descriptor in any forked copies in other processes.

Returns 1 for success; on error, returns "undef" if the socket is not a valid filehandle, or returns 0 and sets $! for any other failure.

sockdomain

```
my $domain = $sock->sockdomain();
```

Returns the number for the socket domain type. For example, for an "AF_INET" socket the value of &AF_INET will be returned.

socket

```
my $sock = IO::Socket->new(); # no values given
# now let's actually get a socket with the socket method
# domain, type, and protocol are required
$sock = $sock->socket(AF_INET, SOCK_STREAM, 'tcp');
```

Opens a socket of the specified kind and returns it. Domain, type, and protocol are specified the same as for the syscall of the same name.

socketpair

```
my ($r, $w) = $sock->socketpair(AF_UNIX, SOCK_STREAM, PF_UNSPEC);
($r, $w) = IO::Socket::UNIX
```

```
    ->socketpair(AF_UNIX, SOCK_STREAM, PF_UNSPEC);
```

Will return a list of two sockets created (read and write), or an empty list on failure.

Differs slightly from "socketpair" in perlfunc in that the argument list is a bit simpler.

sockname

```
    my $packed_addr = $sock->sockname();
```

Returns the packed "sockaddr" address of this end of the connection. It's the same as getsockname(2).

sockopt

```
    my $value = $sock->sockopt(SO_REUSEADDR);
    $sock->sockopt(SO_REUSEADDR, 1);
```

Unified method to both set and get options in the "SOL_SOCKET" level. If called with one argument then "getsockopt" in IO::Socket is called, otherwise "setsockopt" in IO::Socket is called.

socktype

```
    my $type = $sock->socktype();
```

Returns the number for the socket type. For example, for a "SOCK_STREAM" socket the value of &SOCK_STREAM will be returned.

timeout

```
    my $seconds = $sock->timeout();
    my $old_val = $sock->timeout(5); # set new and return old value
```

Set or get the timeout value (in seconds) associated with this socket. If called without any arguments then the current setting is returned. If called with an argument the current setting is changed and the previous value returned.

This method is available to all "IO::Socket" implementations but may or may not be used by the individual domain subclasses.

EXAMPLES

Let's create a TCP server on "localhost:3333".

```
    use strict;
    use warnings;
    use feature 'say';
    use IO::Socket qw(AF_INET AF_UNIX SOCK_STREAM SHUT_WR);
    my $server = IO::Socket->new(
        Domain => AF_INET,
```

```
    Type => SOCK_STREAM,
    Proto => 'tcp',
    LocalHost => '0.0.0.0',
    LocalPort => 3333,
    ReusePort => 1,
    Listen => 5,
) || die "Can't open socket: $IO::Socket::errstr";
say "Waiting on 3333";
while (1) {
    # waiting for a new client connection
    my $client = $server->accept();
    # get information about a newly connected client
    my $client_address = $client->peerhost();
    my $client_port = $client->peerport();
    say "Connection from $client_address:$client_port";
    # read up to 1024 characters from the connected client
    my $data = "";
    $client->recv($data, 1024);
    say "received data: $data";
    # write response data to the connected client
    $data = "ok";
    $client->send($data);
    # notify client that response has been sent
    $client->shutdown(SHUT_WR);
}
$server->close();
```

A client for such a server could be

```
use strict;
use warnings;
use feature 'say';
use IO::Socket qw(AF_INET AF_UNIX SOCK_STREAM SHUT_WR);
my $client = IO::Socket->new(
    Domain => AF_INET,
```

```
    Type => SOCK_STREAM,
    proto => 'tcp',
    PeerPort => 3333,
    PeerHost => '0.0.0.0',
) || die "Can't open socket: $IO::Socket::errstr";
say "Sending Hello World!";
my $size = $client->send("Hello World!");
say "Sent data of length: $size";
$client->shutdown(SHUT_WR);
my $buffer;
$client->recv($buffer, 1024);
say "Got back $buffer";
$client->close();
```

LIMITATIONS

On some systems, for an IO::Socket object created with "new_from_fd", or created with "accept" in IO::Socket from such an object, the "protocol" in IO::Socket, "sockdomain" in IO::Socket and "socktype" in IO::Socket methods may return "undef".

SEE ALSO

Socket, IO::Handle, IO::Socket::INET, IO::Socket::UNIX, IO::Socket::IP

AUTHOR

Graham Barr. atmark() by Lincoln Stein. Currently maintained by the Perl Porters.

Please report all bugs to <perlbug@perl.org>.

COPYRIGHT