

bsdsocket

COLLABORATORS

	TITLE : bsdsocket		
ACTION	NAME	DATE	SIGNATURE
WRITTEN BY		August 22, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1	bsdsocket	1
1.1	bsdsocket.guide	1
1.2	bsdsocket.guide/Disclaimer & Copyright	1
1.3	bsdsocket.guide/accept()	2
1.4	bsdsocket.guide/bind()	4
1.5	bsdsocket.guide/CloseSocket()	4
1.6	bsdsocket.guide/connect()	5
1.7	bsdsocket.guide/Dup2Socket()	7
1.8	bsdsocket.guide/Errno()	8
1.9	bsdsocket.guide/getdtablesize()	8
1.10	bsdsocket.guide/gethostbyname()	9
1.11	bsdsocket.guide/gethostid()	11
1.12	bsdsocket.guide/gethostname()	11
1.13	bsdsocket.guide/getnetbyname()	12
1.14	bsdsocket.guide/getpeername()	13
1.15	bsdsocket.guide/getprotobyname()	14
1.16	bsdsocket.guide/getservbyname()	15
1.17	bsdsocket.guide/GetSocketEvents()	16
1.18	bsdsocket.guide/getsockname()	18
1.19	bsdsocket.guide/getsockopt()	18
1.20	bsdsocket.guide/htonl()	22
1.21	bsdsocket.guide/inet_addr()	23
1.22	bsdsocket.guide/IoctlSocket()	25
1.23	bsdsocket.guide/listen()	27
1.24	bsdsocket.guide/ObtainSocket()	28
1.25	bsdsocket.guide/recv()	29
1.26	bsdsocket.guide/ReleaseSocket()	32
1.27	bsdsocket.guide/select()	33
1.28	bsdsocket.guide/send()	35
1.29	bsdsocket.guide/SetErrnoPtr()	37

1.30 bsdsocket.guide/SetSocketSignals()	38
1.31 bsdsocket.guide/shutdown()	38
1.32 bsdsocket.guide/socket()	39
1.33 bsdsocket.guide/SocketBaseTagList()	42
1.34 bsdsocket.guide/syslog()	46
1.35 bsdsocket.guide/Function Reference	48

Chapter 1

bsdsocket

1.1 bsdsocket.guide

AMITCP/IP API Function Reference

Disclaimer & Copyright
Function Reference

1.2 bsdsocket.guide/Disclaimer & Copyright

Disclaimer & Copyright

This manual is based on original documents from BSD net/2 release,
which is hereby acknowledged:

Copyright © 1982, 1986, 1988, 1990 Regents of the University of
California.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above
copyright notice, this list of conditions and the following
disclaimer in the documentation and/or other materials
provided with the distribution.
3. All advertising materials mentioning features or use of this
software must display the following acknowledgment:
This product includes software developed by the

University of California, Berkeley and its contributors.

4. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This manual is:

Copyright © 1994 AmiTCP/IP Group,
Network Solutions Development Inc.
All rights reserved.

1.3 bsdsocket.guide/accept()

accept()
=====

NAME

accept -- accept a connection on a socket

SYNOPSIS

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

```
int accept(int s, struct sockaddr * addr, int * addrlen)
D0          D0          A0          A1
```

DESCRIPTION

The argument *s* is a socket that has been created with `socket()`, bound to an address with `bind()`, and is listening for connections after a `listen()`. The `accept()` argument extracts the first connection request on the queue of pending connections, creates a new socket with the same properties of *s* and allocates a new file descriptor for the socket. If no pending connections are present on the queue, and the socket is not marked as non-blocking, `accept()` blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, `accept()` returns an error as described below. The accepted socket is used to read and write data to and from the socket which connected to this one; it is not used to

accept more connections. The original socket `s` remains open.

The argument `addr` is a result parameter that is filled in with the address of the connecting entity, as known to the communications layer. The exact format of the `addr` parameter is determined by the domain in which the communication is occurring. The `addrlen` is a value-result parameter; it should initially contain the amount of space pointed to by `addr`; on return it will contain the actual length (in bytes) of the address returned. This call is used with connection-based socket types, currently with `SOCK_STREAM`.

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

One can obtain user connection request data without confirming the connection by issuing a `recv()` call with an `msg_iovlen` of 0 and a non-zero `msg_controllen`, or by issuing a `getsockopt()` request. Similarly, one can provide user connection rejection information by issuing a `send()` call with providing only the control information, or by calling `setsockopt()`.

RETURN VALUES

The call returns `-1` on error. If it succeeds, it returns a non-negative integer that is a descriptor for the accepted socket.

ERRORS

The `accept()` will fail if:

[EBADF]

The descriptor is invalid.

[EINTR]

The operation was interrupted by a break signal.

[EOPNOTSUPP]

The referenced socket is not of type `SOCK_STREAM`.

[EWOULDBLOCK]

The socket is marked non-blocking and no connections are present to be accepted.

SEE ALSO

`bind()`, `connect()`, `listen()`, `select()`, `SetSocketSignals()`, `socket()`, `SocketBaseTagList()`

NOTES

`accept()` calls the `fdCallback()` with action codes `FDCB_CHECK` and `FDCB_ALLOC` to check and mark the new descriptor as allocated if the callback is defined. See `SocketBaseTagList()` for more information on `fdCallback()`.

HISTORY

The `accept()` function appeared in 4.2BSD.

1.4 bsdsocket.guide/bind()

bind()
=====

NAME

bind -- bind a name to a socket

SYNOPSIS

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

```
int bind(int s, struct sockaddr * name, int namelen)
D0          D0          A0          D1
```

DESCRIPTION

Bind() assigns a name to an unnamed socket. When a socket is created with socket() it exists in a name space (address family) but has no name assigned. Bind() requests that name be assigned to the socket.

RETURN VALUES

If the bind is successful, a 0 value is returned. A return value of -1 indicates an error, which is further specified in the library variable errno (see Errno().)

ERRORS

The bind() call will fail if:

[EBADF]

S is not a valid descriptor.

[EADDRNOTAVAIL]

The specified address is not available from the local machine.

[EADDRINUSE]

The specified address is already in use.

[EINVAL]

The socket is already bound to an address.

[EACCES]

The requested address is protected, and the current user has inadequate permission to access it.

SEE ALSO

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

HISTORY

The bind() function call appeared in 4.2BSD.

1.5 bsdsocket.guide/CloseSocket()

CloseSocket()
=====

NAME

CloseSocket -- delete a socket descriptor

SYNOPSIS

```
int CloseSocket(int s)
D0              D0
```

DESCRIPTION

The CloseSocket() call deletes a descriptor from the library base socket reference table. If this is the last reference to the underlying object, the object will be deactivated and socket() associated naming information and queued data are discarded.

All sockets are automatically closed when the socket library is closed, but closing sockets as soon as possible is recommended to save system resources.

RETURN VALUES

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the library variable errno (see Errno()) is set to indicate the error.

ERRORS

CloseSocket() will fail if:
[EBADF]
S is not an active socket descriptor.

[EINTR]
An interrupt was received.

SEE ALSO

accept(),
shutdown(),
socket(),
SocketBaseTagList(),
exec.library/CloseLibrary()

NOTES

CloseSocket() calls the fdCallback() with action code FDCB_FREE if the callback is defined. See SocketBaseTagList() for more information on fdCallback().

1.6 bsdsocket.guide/connect()

connect()
=====

NAME

connect -- initiate a connection on a socket

SYNOPSIS

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

```
int connect(int s, struct sockaddr * name, int namelen)
D0          D0          A0          D1
```

DESCRIPTION

The parameter `s` is a socket. If it is of type `SOCK_DGRAM`, this call specifies the peer with which the socket is to be associated; this address is that to which datagrams are to be sent, and the only address from which datagrams are to be received. If the socket is of type `SOCK_STREAM`, this call attempts to make a connection to another socket. The other socket is specified by `name`, which is an address in the communications space of the socket. Each communications space interprets the `name` parameter in its own way. Generally, stream sockets may successfully `connect()` only once; datagram sockets may use `connect()` multiple times to change their association. Datagram sockets may dissolve the association by connecting to an invalid address, such as a null address.

RETURN VALUES

If the connection or binding succeeds, 0 is returned. Otherwise a -1 is returned, and a more specific error code is stored in `errno`.

ERRORS

The `connect()` call fails if:

[EBADF]

`S` is not a valid descriptor.

[EADDRNOTAVAIL]

The specified address is not available on this machine.

[EAFNOSUPPORT]

Addresses in the specified address family cannot be used with this socket.

[EISCONN]

The socket is already connected.

[ETIMEDOUT]

Connection establishment timed out without establishing a connection.

[ECONNREFUSED]

The attempt to connect was forcefully rejected.

[ENETUNREACH]

The network isn't reachable from this host.

[EADDRINUSE]

The address is already in use.

[EINPROGRESS]

The socket is non-blocking and the connection cannot be

completed immediately. It is possible to `select()` for completion by selecting the socket for writing.

[EALREADY]

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

SEE ALSO

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

HISTORY

The `connect()` function call appeared in 4.2BSD.

1.7 bsdsocket.guide/Dup2Socket()

`Dup2Socket()`

=====

NAME

`Dup2Socket` -- duplicate a socket descriptor

SYNOPSIS

```
int Dup2Socket(int fd1, int fd2)
D0              D0      D1
```

DESCRIPTION

`Dup2Socket()` duplicates an existing socket descriptor. The argument `fd1` is small non-negative value that indexes the socket on library base socket descriptor table. The value must be less than the size of the table, which is returned by `getdtablesize()`. `fd2` specifies the desired value of the new descriptor. If socket pointed by `fd2` is already in use, it is first deallocated as if it were closed by `CloseSocket()`. If the value of `fd2` is `-1`, the new descriptor used and returned is the lowest numbered descriptor that is not currently in use by the task's socket library base.

Since AMITCP/IP 3.0 `Dup2Socket()` has also an feature to mark a file descriptor as being used. If `fd1` is given as `-1`, `fd2` is marked as being used socket descriptor table and it won't be allocated for any socket. This mark can be removed using `CloseSocket()` call.

RETURN VALUES

The value `-1` is returned if an error occurs in either call. The library variable `errno` (see `Errno()`) indicates the cause of the error.

ERRORS

`Dup2Socket()` fail if:

[EBADF]

`sd1` or `sd2` is not a valid active descriptor

[EMFILE]

Too many descriptors are active.

SEE ALSO

accept(), CloseSocket(), socket(), getdtablesize(), SocketBaseTagList()

NOTES

Dup2Socket() calls the fdCallback() with action codes FDCB_CHECK and FDCB_ALLOC to check and mark the new descriptor as allocated if the callback is defined. See SocketBaseTagList() for more information on fdCallback().

1.8 bsdsocket.guide/Errno()

Errno()

=====

NAME

Errno -- get error value after unsuccessful function call

SYNOPSIS

long Errno(void)
D0

FUNCTION

When some function in socket library return an error condition value, they also set a specific error value. This error value can be (among other things) extracted by this function.

RESULT

Error value indicating the error on last failure of some bsdsocket library function call.

NOTES

Return value of Errno() is not changed after successful function so so it cannot be used to determine success of any function call of this library. Also, another function call to this library may change the return value of Errno() so use it right after error occurred.

SEE ALSO

SetErrnoPtr(), SocketBaseTagList()

1.9 bsdsocket.guide/getdtablesize()

getdtablesize()

=====

NAME

getdtablesize -- get descriptor table size

SYNOPSIS

```
int getdtablesize(void)
D0
```

DESCRIPTION

Each socket library base has a socket descriptor table which initially has 64 slots. The entries in the descriptor table are numbered with small integers starting at 0. The call `getdtablesize()` returns the size of this table.

SEE ALSO

`CloseSocket()`, `Dup2Socket()`, `select()` `SocketBaseTagList()`

NOTES

AMITCP/IP versions below 3.0 used to have function `SetDtableSize()` in place of this function. From now on, use `SocketBaseTagList()` to set new size for socket descriptor table.

HISTORY

The `getdtablesize()` function call appeared in 4.2BSD.

1.10 bsdsocket.guide/gethostbyname()

```
gethostbyname(), gethostbyaddr()
```

```
=====
```

NAME

`gethostbyname`, `gethostbyaddr` -- get network host entry

SYNOPSIS

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

struct hostent * gethostbyname(char * name)
D0                                A0

struct hostent * gethostbyaddr(char * addr, int len, int type)
D0                                A0            D0            D1
```

DESCRIPTION

The `gethostbyname()` and `gethostbyaddr()` functions each return a pointer to an object with the following structure describing an internet host referenced by name or by address, respectively. This structure contains either the information obtained from the name server, named or broken-out fields from a line in `/etc/hosts`. If the local name server is not running these routines do a lookup in `/etc/hosts`.

```
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 from name server */
}
```

```
};  
#define h_addr  h_addr_list[0]  /* address, for backward compatibility */
```

The members of this structure are:

h_name

Official name of the host.

h_aliases

A zero 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 zero terminated array of network addresses for the host.
Host addresses are returned in network byte order.

h_addr

The first address in **h_addr_list**; this is for backward
(source) compatibility.

DIAGNOSTICS

Error return status from `gethostbyname()` and `gethostbyaddr()` is indicated by return of a null pointer. The library integer `h_errno` (see `SocketBaseTagList()`) may then be checked to see whether this is a temporary failure or an invalid or unknown host. If its argument string is non-NULL, it is printed, followed by a colon and a space. The error message is printed with a trailing newline.

The library variable `h_errno` can have 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 later time may succeed.

NO_RECOVERY

Some unexpected server failure was encountered. This is a non-recoverable error.

NO_DATA

The requested name is valid but does not have an IP address; this is not a temporary error. This means that the name is known to the name server but there is no address associated with this name. Another type of request to the name server using this domain name will result in an answer; for example, a mail-forwarder may be registered for this domain.

SEE ALSO

AMITCP/IP configuration

HISTORY

The `herror()` function appeared in 4.3BSD. `gethostbyaddr()` and `gethostbyname()` functions appeared in 4.2BSD.

BUGS

These functions use static data storage; if the data is needed for future use, it should be copied before any subsequent calls overwrite it. Only the Internet address format is currently understood.

1.11 bsdsocket.guide/gethostid()

`gethostid()`
=====

NAME

`gethostid` -- get unique identifier of current host

SYNOPSIS

```
long gethostid(void)
D0
```

DESCRIPTION

`gethostid()` returns the 32-bit identifier for the current processor that is intended to be unique among all UNIX systems in existence. This is normally a DARPA Internet address for the local machine.

SEE ALSO

`gethostname()`

BUGS

32 bits for the identifier is too small.

HISTORY

The `gethostid()` function call appeared in 4.2BSD.

1.12 bsdsocket.guide/gethostname()

`gethostname()`
=====

NAME

`gethostname` -- get name of current host

SYNOPSIS

```
int gethostname(char * name, int namelen)
D0                      A0                      D0
```

DESCRIPTION

`Gethostname()` returns the standard host name for the current

processor. The parameter `namelen` specifies the size of the name array. The returned name is null-terminated unless insufficient space is provided.

SEE ALSO
`gethostid()`

HISTORY
 The `gethostname()` function call appeared in 4.2BSD.

1.13 bsdsocket.guide/getnetbyname()

`getnetbyname()`, `getnetbyaddr()`
 =====

NAME
 `getnetbyaddr`, `getnetbyname` -- get network entry

SYNOPSIS
 #include <netdb.h>

```

struct netent * getnetbyname(char * name)
D0                                A0

struct netent * getnetbyaddr(long net, int type)
D0                                D0      D1
  
```

DESCRIPTION
 The `getnetbyname()` and `getnetbyaddr()` functions each return a pointer to an object with the following structure containing the broken-out fields of a line in the network data base.

```

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

The members of this structure are:
`n_name`
 The official name of the network.

`n_aliases`
 A zero terminated list of alternate names for the network.

`n_addrtype`
 The type of the network number returned; currently only `AF_INET`.

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

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

SEE ALSO

AMITCP/IP configuration

HISTORY

The `getnetbyaddr()` and `getnetbyname()` functions appeared in 4.2BSD.

BUGS

The data space used by these functions is static; if future use requires the data, it should be copied before any subsequent calls to these functions overwrite it. Only Internet network numbers are currently understood. Expecting network numbers to fit in no more than 32 bits is probably naive.

1.14 bsdsocket.guide/getpeername()

`getpeername()`

=====

NAME

`getpeername` -- get name of connected peer

SYNOPSIS

```
int getpeername(int s, struct sockaddr * name, int * namelen)
D0              D0              A0              A1
```

DESCRIPTION

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

DIAGNOSTICS

A 0 is returned if the call succeeds, -1 if it fails.

ERRORS

The call succeeds unless:

[EBADF]

The argument `s` is not a valid descriptor.

[ENOTCONN]

The socket is not connected.

[ENOBUFS]

Insufficient resources were available in the system to perform the operation.

SEE ALSO

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

HISTORY

The `getpeername()` function call appeared in 4.2BSD.

1.15 bsdsocket.guide/getprotobyname()

```
getprotobyname(), getprotobynumber()
=====
```

NAME

`getprotobynumber, getprotobyname` -- get protocol entry

SYNOPSIS

```
#include <netdb.h>
```

```
struct protoent * getprotobyname(char * name)
D0                                     A0
```

```
struct protoent * getprotobynumber(int proto)
D0                                     D0
```

DESCRIPTION

The `getprotobyname()` and `getprotobynumber()` functions each return a pointer to an object with the following structure containing the broken-out fields of a line in the network protocol data base.

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

The members of this structure are:

`p_name`
The official name of the protocol.

`p_aliases`
A zero terminated list of alternate names for the protocol.

`p_proto`
The protocol number.

RETURN VALUES

Null pointer (0) returned on EOF or error.

SEE ALSO

AMITCP/IP configuration

HISTORY

The `getprotobynumber()` and `getprotobyname()` 4.2BSD.

BUGS

These functions use a static data space; if the data is needed for future use, it should be copied before any subsequent calls overwrite it. Only the Internet protocols are currently understood.

1.16 bsdsocket.guide/getservbyname()

```
getservbyname(), getservbyport()
=====
```

NAME

```
getservbyport, getservbyname -- get service entry
```

SYNOPSIS

```
#include <netdb.h>
```

```
struct servent * getservbyname(char * name, char * proto)
D0                                A0                                A1
```

```
struct servent * getservbyport(int port, char * proto)
D0                                D0                                A0
```

DESCRIPTION

The getservbyname() and getservbyport() functions each return a pointer to an object with the following structure containing the broken-out fields of a line in the network services data base.

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

The members of this structure are:

s_name

The official name of the service.

s_aliases

A zero terminated list of alternate names for the service.

s_port

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

s_proto

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

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

SEE ALSO

getprotobyname() and AMITCP/IP configuration

HISTORY

The getservbyport() and getservbyname() functions appeared in 4.2BSD.

BUGS

These functions use static data storage; if the data is needed for future use, it should be copied before any subsequent calls overwrite it. Expecting port numbers to fit in a 32 bit quantity is probably naive.

1.17 bsdsocket.guide/GetSocketEvents()

GetSocketEvents()
=====

NAME

GetSocketEvents -- get asynchronous socket events if any

SYNOPSIS

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

```
LONG GetSocketEvents(ULONG *eventsp)
                        A0
```

FUNCTION

GetSocketEvents() is used to retrieve asynchronous events of sockets. The socket descriptor for which events happened is returned as a return value. If the return value is -1, then there are no sockets with asynchronous events. Note that the errno variable is not set in this case.

The asynchronous socket events are notified with the event signal. The application must set this signal with the SocketBaseTagList() tag code SBTC_SIGEVTMASK. In addition to that, the application must set the event filter masks for each socket with the getsockopt() SO_EVENTMASK option. The socket must also be in async mode (IoctlSocket() FIOASYNC), but this mode will be set on by the getsockopt() for SO_EVENTMASK automatically. The socket must also have an owner task which to notify. The owner task is set to the creating task of the socket by default. A socket got from ObtainSocket() retains the old owner, however. So, after ObtainSocket(), the owner must be set explicitly with the IoctlSocket() code FIOSETOWN.

The events are returned via eventsp as a bit mask of event codes defined in <sys/socket.h>. The event codes should be tested in the order they are defined in the header file and described below, since it is possible to get FD_CONNECT, FD_READ and FD_CLOSE at the same time, for example. (That would mean that the connection was established, data was received and the connection was closed).

FD_ACCEPT

Indicates that there is a new connection waiting for acceptance. This event is level triggered, meaning that the event is set again if there is yet another connection after the accept() call. Because of this it is enough to respond with one accept() call for each FD_ACCEPT received.

FD_CONNECT

Indicates that the pending connection has been established. This event should be waited for after a non-blocking `connect()`. Note that this event is generated even if the socket is not in non-blocking mode.

FD_OOB

Indicates that the socket has received a notification of urgent (out-of-band) data. The urgent data might not yet be received, and it might be impossible to receive, if the socket receive buffer is full. This means that the application should read the normal data from the buffer so that the urgent data could be received and read by the application.

FD_READ

This event is set whenever new data arrives on the socket receive buffer. Note that the application should read all the data it can from the socket, since there is no guarantee that new data would arrive, and thus cause a new event to be generated.

FD_WRITE

This indicates that the application can write to the non-blocking socket again. This event is set only if the socket is in non-blocking mode. This event is generated only after the application could not write all the data to the socket (i.e., the operation would have blocked if the socket was not in non-blocking mode).

FD_ERROR

This indicates an asynchronous error that happened on the socket. When this event is returned the `errno` is set to the socket error code.

Note that the socket error code is not cleared, so that the forthcoming socket functions might notice the error, too. If the socket error code must be cleared, the `getsockopt()` for `SO_ERROR` should be called, since the internal socket error is cleared as a side effect of that function (and code).

FD_CLOSE

This indicates that the connection was closed. Note that the socket receive buffer might still have some data which should be received before the socket is closed.

If the connection was closed due to an error, the `FD_ERROR` event is sent before this event. The `FD_ERROR` could be either returned by an earlier `GetSocketEvents()` for this socket or included in the same event mask with the `FD_CLOSE`.

The exact semantics for the `FD_CLOSE` depend on the application protocol. If the other end uses `shutdown()` to close only the sending side of the connection, the receiver of the `FD_CLOSE` might still be able to send data.

EXAMPLE
NOTES

SEE ALSO
 IoctlSocket(), recv(), send(), setsockopt(),
 SocketBaseTagList()
 !!

1.18 bsdsocket.guide/getsockname()

getsockname()
 =====

NAME

getsockname -- get socket name

SYNOPSIS

```
int getsockname(int s, struct sockaddr * name, int * namelen)
D0                D0                A0                A1
```

DESCRIPTION

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

DIAGNOSTICS

A 0 is returned if the call succeeds, -1 if it fails.

ERRORS

The call succeeds unless:

[EBADF]

The argument s is not a valid descriptor.

[ENOBUFFS]

Insufficient resources were available in the system to
 perform the operation.

SEE ALSO

bind(), socket(), getpeername()

HISTORY

The getsockname() function call appeared in 4.2BSD.

1.19 bsdsocket.guide/getsockopt()

getsockopt(), setsockopt()
 =====

NAME

getsockopt, setsockopt -- get and set options on sockets

SYNOPSIS

```

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

int getsockopt(int s, int level,
D0              D0      D1
                int optname, void * optval, int * optlen)
                D2              A0              A1

int setsockopt(int s, int level,
D0              D0      D1
                int optname, const void * optval, int optlen)
                D2              A0              D3

```

DESCRIPTION

getsockopt() and setsockopt() manipulate the options associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost "socket" level.

When manipulating socket options the level at which the option resides and the name of the option must be specified. To manipulate options at the socket level, level is specified as SOL_SOCKET. To manipulate options at any other level the protocol number of the appropriate protocol controlling the option is supplied. For example, to indicate that an option is to be interpreted by the TCP protocol, level should be set to the protocol number of TCP; see getprotobyname().

The parameters optval and optlen are used to access option values for setsockopt(). For getsockopt() they identify a buffer in which the value for the requested option(s) are to be returned. For getsockopt() optlen is a value-result parameter, initially containing the size of the buffer pointed to by optval, and modified on return to indicate the actual size of the value returned. If no option value is to be supplied or returned, optval may be NULL.

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, described below. Options at other protocol levels vary in format and name.

Most socket-level options utilize an int parameter for optval. For setsockopt() the parameter should be non-zero to enable a boolean option, or zero if the option is to be disabled.

SO_LINGER uses a struct linger parameter, defined in <sys/socket.h>, which specifies the desired state of the option and the linger interval (see below). SO_SNDTIMEO and SO_RCVTIMEO use a struct timeval parameter, defined in <sys/time.h>.

The following options are recognized at the socket level. Except as noted, each may be examined with getsockopt() and set with setsockopt()

`SO_DEBUG`
enables recording of debugging information

`SO_REUSEADDR`
enables local address reuse

`SO_KEEPAIVE`
enables keep connections alive

`SO_DONTROUTE`
enables routing bypass for outgoing messages

`SO_LINGER`
linger on close if data present

`SO_BROADCAST`
enables permission to transmit broadcast messages

`SO_OOBINLINE`
enables reception of out-of-band data in band

`SO_SNDBUF`
set buffer size for output

`SO_RCVBUF`
set buffer size for input

`SO_SNDLOWAT`
set minimum count for output

`SO_RCVLOWAT`
set minimum count for input

`SO_SNDTIMEO`
set timeout value for output

`SO_RCVTIMEO`
set timeout value for input

`SO_TYPE`
get the type of the socket (get only)

`SO_ERROR`
get and clear error on the socket (get only)

`SO_EVENTMASK`
set event mask to specify which events should be notified to the application

`SO_DEBUG` enables debugging in the underlying protocol modules. `SO_REUSEADDR` indicates that the rules used in validating addresses supplied in a `bind()` call should allow reuse of local addresses. `SO_KEEPAIVE` enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken and processes using the socket are notified via a `SIGPIPE` signal when attempting to send data. `SO_DONTROUTE` indicates that

outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.

SO_LINGER controls the action taken when unsent messages are queued on socket and a CloseSocket() is performed. If the socket promises reliable delivery of data and SO_LINGER is set, the system will block the process on the CloseSocket() attempt until it is able to transmit the data or until it decides it is unable to deliver the information (a timeout period, termed the linger interval, is specified in the setsockopt() call when SO_LINGER is requested). If SO_LINGER is disabled and a CloseSocket() is issued, the system will process the close in a manner that allows the process to continue as quickly as possible.

The option SO_BROADCAST requests permission to send broadcast datagrams on the socket. Broadcast was a privileged operation in earlier versions of the system. With protocols that support out-of-band data, the SO_OOBINLINE option requests that out-of-band data be placed in the normal data input queue as received; it will then be accessible with recv() calls without the MSG_OOB flag. Some protocols always behave as if this option is set. SO_SNDBUF and SO_RCVBUF are options to adjust the normal buffer sizes allocated for output and input buffers, respectively. The buffer size may be increased for high-volume connections, or may be decreased to limit the possible backlog of incoming data. The system places an absolute limit on these values.

SO_SNDLOWAT is an option to set the minimum count for output operations. Most output operations process all of the data supplied by the call, delivering data to the protocol for transmission and blocking as necessary for flow control. Nonblocking output operations will process as much data as permitted subject to flow control without blocking, but will process no data if flow control does not allow the smaller of the low water mark value or the entire request to be processed. A select() operation testing the ability to write to a socket will return true only if the low water mark amount could be processed. The default value for SO_SNDLOWAT is set to a convenient size for network efficiency, often 1024. SO_RCVLOWAT is an option to set the minimum count for input operations. In general, receive calls will block until any (non-zero) amount of data is received, then return with smaller of the amount available or the amount requested. The default value for SO_SNDLOWAT is 1. If SO_SNDLOWAT is set to a larger value, blocking receive calls normally wait until they have received the smaller of the low water mark value or the requested amount. Receive calls may still return less than the low water mark if an error occurs, a signal is caught, or the type of data next in the receive queue is different than that returned.

SO_SNDTIMEO is an option to set a timeout value for output operations. It accepts a struct timeval parameter with the number of seconds and microseconds used to limit waits for output operations to complete. If a send operation has blocked for this much time, it returns with a partial count or with the error

EWouldBlock if no data were sent. In the current implementation, this timer is restarted each time additional data are delivered to the protocol, implying that the limit applies to output portions ranging in size from the low water mark to the high water mark for output. `SO_RCVTIMEO` is an option to set a timeout value for input operations. It accepts a `struct timeval` parameter with the number of seconds and microseconds used to limit waits for input operations to complete. In the current implementation, this timer is restarted each time additional data are received by the protocol, and thus the limit is in effect an inactivity timer. If a receive operation has been blocked for this much time without receiving additional data, it returns with a short count or with the error `EWouldBlock` if no data were received.

Finally, `SO_TYPE` and `SO_ERROR` are options used only with `setsockopt()`. `SO_TYPE` returns the type of the socket, such as `SOCK_STREAM`; it is useful for servers that inherit sockets on startup. `SO_ERROR` returns any pending error on the socket and clears the error status. It may be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors.

Additionally, `SO_EVENTMASK` is used to specify which asynchronous events are to be notified to the socket owner. See `GetSocketEvents()` for more info on this. `SO_EVENTMASK` is AMITCP/IP addition.

RETURN VALUES

A 0 is returned if the call succeeds, -1 if it fails.

ERRORS

The call succeeds unless:

[EBADF]

The argument `s` is not a valid descriptor.

[ENOPROTOOPT]

The option is unknown at the level indicated.

SEE ALSO

`ioctlsocket()`, `socket()`, `getprotobyname()`, `GetSocketEvents()`

BUGS

Several of the socket options should be handled at lower levels of the system.

HISTORY

The `getsockopt()` system call appeared in 4.2BSD.

1.20 bsdsocket.guide/htonl()

`htonl()`, `htons()`, `ntohl()`, `ntohs()`

=====

NAME

htonl, htons, ntohl, ntohs -- convert values between host and network byte order

SYNOPSIS

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

u_long htonl(u_long hostlong)

u_short htons(u_short hostshort)

u_long ntohl(u_long netlong)

u_short ntohs(u_short netshort)
```

DESCRIPTION

These routines convert 16 and 32 bit quantities between network byte order and host byte order. On machines which have a byte order which is the same as the network order, routines are defined as null macros. Commodore Amiga is such a machine.

These routines are most often used in conjunction with Internet addresses and ports as returned by gethostbyname() and getservbyname().

SEE ALSO

gethostbyname(), getservbyname()

HISTORY

The byteorder -- functions appeared in 4.2BSD.

BUGS

On the VAX bytes are handled backwards from most everyone else in the world. This is not expected to be fixed in the near future.

1.21 bsdsocket.guide/inet_addr()

```
inet_addr(), inet_network(), inet_ntoa(), inet_makeaddr(), inet_lnaof(), ↵
inet_netof()
```

NAME

inet_addr, inet_network, Inet_NtoA, Inet_MakeAddr, Inet_LnaOf, Inet_NetOf -- Internet address manipulation routines
inet_ntoa, inet_makeaddr, inet_lnaof, inet_netof -- inline/stub functions of above to handle structure args

SYNOPSIS

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

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

```

D0                                     A0

unsigned long inet_network(const char * cp)
D0                                     A0

char * Inet_Ntoa(unsigned long in)
D0                                     D0

unsigned long Inet_MakeAddr(int net, int lna)
D0                                     D0      D1

unsigned long Inet_LnaOf(unsigned long in)
D0                                     D0

unsigned long Inet_NetOf(unsigned long in)
D0                                     D0

char * inet_ntoa(struct in_addr in)

struct in_addr inet_makeaddr(int net, int lna)

unsigned long inet_lnaof(struct in_addr in)

unsigned long inet_netof(struct in_addr in)

```

IMPLEMENTATION NOTE

Return value of `Inet_MakeAddr()` and argument types of `Inet_LnaOf()`, `Inet_NetOf()` and `Inet_Ntoa()` are longs instead of `struct in_addr`. The original behaviour is achieved by using included stub functions (lower case function names) which handle structure arguments.

DESCRIPTION

The routines `inet_addr()` and `inet_network()` interpret character strings representing numbers expressed in the Internet standard `'.'` notation, returning numbers suitable for use as Internet addresses and Internet network numbers, respectively. The routine `inet_ntoa()` takes an Internet address and returns an ASCII string representing the address in base 256 notation `"d.d.d.d"` described below. The routine `inet_makeaddr()` takes an Internet network number and a local network address and constructs an Internet address from it. The routines `inet_netof()` and `inet_lnaof()` break apart Internet host addresses, returning the network number and local network address part, respectively.

All Internet addresses are returned in network order (bytes ordered from left to right). All network numbers and local address parts are returned as machine format integer values.

INTERNET ADDRESSES

Values specified using the `'.'` 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. Note that when an Internet address is viewed as a 32-bit integer quantity on the VAX the bytes referred to above appear as "d.c.b.a ." That is, VAX bytes are ordered from right to left.

When a three part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right-most two bytes of the network address. This makes the three part address format convenient for specifying Class B network addresses as "128.net.host ."

When a two part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right most three bytes of the network address. This makes the two part address format convenient for specifying Class A network addresses as "net.host ."

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as "parts" in a '.' notation may be decimal, octal, or hexadecimal, as specified in the C language (i.e., a leading 0x or 0X implies hexadecimal; otherwise, a leading 0 implies octal; otherwise, the number is interpreted as decimal).

DIAGNOSTICS

The constant `INADDR_NONE` is returned by `inet_addr()` and `inet_network()` for malformed requests.

SEE ALSO

`gethostbyname()`

HISTORY

These functions appeared in 4.2BSD.

BUGS

The value `INADDR_NONE` (0xffffffff) is a valid broadcast address, but `inet_addr()` cannot return that value without indicating failure. The problem of host byte ordering versus network byte ordering is confusing. A simple way to specify Class C network addresses in a manner similar to that for Class B and Class A is needed. The string returned by `inet_ntoa()` resides in a static memory area.

`Inet_addr` should return a struct `in_addr`.

1.22 bsdsocket.guide/loctlSocket()

`IoctlSocket()`

=====

NAME

`IoctlSocket` -- control sockets

SYNOPSIS

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

```
int IoctlSocket(int s, unsigned long request, char * argp)
D0                D0                D1                A0
```

DESCRIPTION

`IoctlSocket()` performs a special function on the object referred to by the open socket descriptor `s`. Note: the `getsockopt()` call is the primary method for operating on sockets as such, rather than on the underlying protocol or network interface.

For most `IoctlSocket()` functions, `argp` is a pointer to data to be used by the function or to be filled in by the function. Other functions may ignore `arg` or may treat it directly as a data item; they may, for example, be passed an `int` value.

An `ioctl` request has encoded in it whether the argument is an "in" parameter or "out" parameter, and the size of the argument `argp` in bytes. Macros and defines used in specifying an `ioctl` request are located in the file `<sys/ioctl.h>`.

The following requests are supported:

FIOASYNC

The argument is a pointer to a long. Set or clear asynchronous I/O. If the value of that long is a 1 (one) the descriptor is set for asynchronous I/O. If the value of that long is a 0 (zero) the descriptor is cleared for asynchronous I/O.

FIOCLEX

FIONCLEX

Ignored, no use for close-on-exec flag in Amiga.

FIOGETOWN

SIOCGPGRP

The argument is pointer to struct `Task*`. Set the value of that pointer to the Task that is receiving SIGIO or SIGURG signals for the socket referred to by the descriptor passed to `IoctlSocket()`.

FIONBIO

The argument is a pointer to a long. Set or clear non-blocking I/O. If the value of that long is a 1 (one) the descriptor is set for non-blocking I/O. If the value of that long is a 0 (zero) the descriptor is cleared for non-blocking I/O.

FIONREAD

The argument is a pointer to a long. Set the value of that long to the number of immediately readable characters from the socket fd.

FIOSETOWN

SIOCSPGRP

The argument is pointer to struct Task*, pointer to the task that will subsequently receive SIGIO or SIGURG signals for the socket referred to by the descriptor passed.

SIOCCATMARK

The argument is a pointer to a long. Set the value of that long to 1 if the read pointer for the socket referred to by the descriptor passed to IoctlSocket() points to a mark in the data stream for an out-of-band message, and to 0 if it does not point to a mark.

RETURN VALUES

IoctlSocket() returns 0 on success for most requests. Some specialized requests may return non-zero values on success; On failure, IoctlSocket() returns -1 and sets errno to indicate the error.

ERRORS

Ioctl() will fail:

[EBADF]

s is not a valid descriptor.

[EINVAL]

Request or argp is not valid.

IoctlSocket() will also fail if the object on which the function is being performed detects an error. In this case, an error code specific to the object and the function will be returned.

SEE ALSO

setsockopt(), SocketBaseTagList()

HISTORY

An ioctl() function call appeared in Version 7 AT&T UNIX.

1.23 bsdsocket.guide/listen()

listen()

=====

NAME

listen -- listen for connections on a socket

SYNOPSIS

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

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

```
D0          D0          D1
```

DESCRIPTION

To accept connections, a socket is first created with socket(), a willingness to accept incoming connections and a queue limit for incoming connections are specified with listen() and then the connections are accepted with accept(). The listen() call applies

only to sockets of type SOCK_STREAM or SOCK_SEQPACKET.

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

RETURN VALUES

A 0 return value indicates success; -1 indicates an error.

ERRORS

Listen() will fail if:

[EBADF]

The argument s is not a valid descriptor.

[EOPNOTSUPP]

The socket is not of a type that supports the operation listen()

SEE ALSO

accept(), connect(), socket()

BUGS

The backlog is currently limited (silently) to 5.

HISTORY

The listen() function call appeared in 4.2BSD.

1.24 bsdsocket.guide/ObtainSocket()

ObtainSocket()

=====

NAME

ObtainSocket -- get a previously released socket

SYNOPSIS

```
LONG ObtainSocket(LONG id, LONG domain, LONG type, LONG protocol)
D0                D0        D1        D2        D3
```

FUNCTION

When one task wants to give a socket to an another one, it releases it (with a key value) to a special socket list held by AmiTCP/IP. This function requests that socket and receives it if id and other parameters match.

After release 3.0 of AMITCP/IP ObtainSocket() can be called with an unique id and domain as ZERO to request any kind of socket that is available with that id.

INPUTS

id

a key value for finding the correct available socket.

domain

see documentation of `socket()`.

type

see documentation of `socket()`.

protocol

see documentation of `socket()`.

RESULT

Non negative socket descriptor on success. On failure, -1 is returned and the `errno` is set to indicate the error.

ERRORS

EMFILE

The per-process descriptor table is full.

EPROTONOSUPPORT

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

EPROTOTYPE

The protocol is the wrong type for the socket.

EWouldBlock

Matching socket is not found.

SEE ALSO

`ReleaseSocket()`, `socket()`, `SocketBaseTagList()`

NOTES

`ObtainSocket()` calls the `fdCallback()` with action codes `FDCB_CHECK` and `FDCB_ALLOC` to check and mark the new descriptor as allocated if the callback is defined. It may also call the callback again with `FDCB_FREE`, if an error condition occurs. See `SocketBaseTagList()` for more information on `fdCallback()`.

1.25 bsdsocket.guide/recv()

`recv()`, `recvfrom()`, `recvmsg()`

=====

NAME

`recv`, `recvfrom`, `recvmsg` -- receive a message from a socket

SYNOPSIS

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

```
int recv(int s, void * buf, int len, int flags)
D0          D0          A0          D1          D2
```

```

int recvfrom(int s, void * buf, int len, int flags,
D0          D0          A0          D1          D2
              struct sockaddr * from, int * fromlen)
                  A1          A2

int recvmsg(int s, struct msghdr * msg, int flags)
D0          D0          A0          D1

```

DESCRIPTION

Recv(), recvfrom() and recvmsg() are used to receive messages from a socket. Recv() may be used only on a connected socket (see connect()), while recvfrom() and recvmsg() and may be used to receive data on a socket whether or not it is connection-oriented.

If from is non-nil, and the socket is not connection-oriented, the source address of the message is filled in. Fromlen is a value-result parameter, initialized to the size of the buffer associated with from, and modified on return to indicate the actual size of the address stored there.

All three routines return the length of the message on successful completion. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from (see socket()).

If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is nonblocking (see IoctlSocket()) in which case the value -1 is returned and the library variable errno (see Errno()) set to EWOULDBLOCK. The receive calls normally return any data available, up to the requested amount, rather than waiting for receipt of the full amount requested; this behavior is affected by the socket-level options SO_RCVLOWAT and SO_RCVTIMEO described in getsockopt().

The select() call may be used to determine when more data arrive.

The flags argument to a recv call is formed by or'ing one or more of the values:

MSG_OOB
process out-of-band data

MSG_PEEK
peek at incoming message

MSG_WAITALL
wait for full request or error

The MSG_OOB flag requests receipt of out-of-band data that would not be received in the normal data stream. Some protocols place expedited data at the head of the normal data queue, and thus this flag cannot be used with such protocols. The MSG_PEEK flag causes the receive operation to return data from the beginning of the receive queue without removing that data from the queue. Thus, a subsequent receive call will return the same data. The MSG_WAITALL flag requests that the operation block until the full request is satisfied. However, the call may still return less data than requested if a signal is caught, an error or disconnect occurs, or the next data to be received is of a

different type than that returned.

The `recvmsg()` 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 */
    u_int   msg_namelen;   /* size of address */
    struct  iovec *msg_iov; /* scatter/gather array */
    u_int   msg_iovlen;    /* # elements in msg_iov */
    caddr_t msg_control;   /* ancillary data, see below */
    u_int   msg_controllen; /* ancillary data buffer len */
    int     msg_flags;     /* flags on received message */
};
```

Here `msg_name` and `msg_namelen` specify the destination address if the socket is unconnected; `msg_name` may be given as a null pointer if no names are desired or required. `msg_iov` and `msg_iovlen` describe scatter gather locations, the input data is scattered into `msg_iovlen` buffers specified by the members of the `msg_iov` array: `msg_iov[0]`, `msg_iov[0]`, ..., `msg_iov[msg_iovlen - 1]`.

The `iovec` structure is defined as

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

Each `iovec` entry specifies the base address and length of an area in memory where data should be placed. `Recvmsg()` will always fill an area completely before proceeding to the next.

The fields `msg_control` and `msg_controllen` are not currently used in this implementation, set them as 0 before use.

The `msg_flags` field is set on return according to the message received. `MSG_EOR` indicates end-of-record; the data returned completed a record (generally used with sockets of type `SOCK_SEQPACKET`). `MSG_TRUNC` indicates that the trailing portion of a datagram was discarded because the datagram was larger than the buffer supplied. `MSG_CTRUNC` indicates that some control data were discarded due to lack of space in the buffer for ancillary data. `MSG_OOB` is returned to indicate that expedited or out-of-band data were received.

RETURN VALUES

These calls return the number of bytes received, or -1 if an error occurred.

ERRORS

The calls fail if:

[EBADF]

The argument `s` is an invalid descriptor.

[ENOTCONN]

The socket is associated with a connection-oriented protocol and has not been connected (see `connect()` and `accept()`).

[EWOULDBLOCK]

The socket is marked non-blocking, and the receive operation would block, or a receive timeout had been set, and the timeout expired before data were received.

[EINTR]

The receive was interrupted by delivery of a signal before any data were available.

SEE ALSO

`getsockopt()`, `IoctlSocket()`, `select()`, `send()`, `socket()`

HISTORY

The `recv()` function call appeared in 4.2BSD.

1.26 bsdsocket.guide/ReleaseSocket()

`ReleaseSocket()`, `ReleaseCopyOfSocket()`

=====

NAME

`ReleaseSocket`, `ReleaseCopyOfSocket` -- make a socket be available with `ObtainSocket()` call.

SYNOPSIS

```
LONG ReleaseSocket(LONG sd, LONG id)
D0                  D0          D1
```

```
LONG ReleaseCopyOfSocket(LONG sd, LONG id)
D0                        D0          D1
```

FUNCTION

The `ReleaseSocket()` removes the socket pointed by descriptor `sd` from its descriptor table and puts it available via next `ObtainSocket()` call.

The `ReleaseCopyOfSocket()` makes a new reference to given socket and that is made available to another caller.

INPUTS

`sd`

descriptor of the socket to be released.

`id`

the key value to identify use of this socket. It can be unique or not, depending on its value. If `id` value is between 0 and 65535, inclusively, it is considered nonunique and it can be used as a port number, for example. If `id` is greater than 65535 and less than 2^{31} it must be unique in currently held sockets in

AMITCP/IP socket list, Otherwise an error will be returned and socket is not released. If `id == UNIQUE_ID` (defined in `<sys/socket.h>`) an unique id will be generated.

RESULT

-1 in case of error, otherwise the key value of the socket put in the list. Most useful when an unique id is generated by this routine.

ERRORS

EINVAL

Requested unique id is already used.

ENOMEM

The memory needed could not be allocated.

NOTE

In case of `ReleaseCopyOfSocket()`. The socket descriptor is not deallocated from the descriptor table of the calling task.

SEE ALSO

`ObtainSocket()`. `SocketBaseTagList()`

NOTES

`ReleaseSocket()` (but not `ReleaseCopyOfSocket()`) calls the `fdCallback()` with action code `FDCB_FREE` if the callback is defined. See `SocketBaseTagList()` for more information on `fdCallback()`.

1.27 bsdsocket.guide/select()

`select()`, `WaitSelect()`
=====

NAME

`select` -- synchronous I/O multiplexing (stub/inline function)
`WaitSelect` -- `select()` with `Exec Wait()` function.

SYNOPSIS

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

```
int select(int nfds, fd_set * readfds, fd_set * writefds,
           fd_set * exceptfds, struct timeval * timeout)
```

```
int WaitSelect(int nfds, fd_set * readfds, fd_set * writefds,
D0             D0             A0             A1
                fd_set * exceptfds, struct timeval * timeout,
                A2             A3
                ULONG * sigmask)
                D1
```

```
FD_SET(fd, &fdset)
FD_CLR(fd, &fdset)
FD_ISSET(fd, &fdset)
FD_ZERO(&fdset)
```

DESCRIPTION

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

`WaitSelect()` also takes a signal mask which is waited during normal `select()` operation. If one of these signals is received, `WaitSelect()` returns and has re-set the signal mask to return those signals that have arrived and value 0 is returned (total number of ready descriptors in all the sets, which is currently zero for efficiency reasons).

The descriptor sets are stored as bit fields in arrays of integers. The following macros are provided for manipulating such descriptor sets: `FD_ZERO(&fdset)` initializes a descriptor set `fdset` to the null set. `FD_SET(fd, &fdset)` includes a particular descriptor `fd` in `fdset`. `FD_CLR(fd, &fdset)` removes `fd` from `fdset`. `FD_ISSET(fd, &fdset)` is non-zero if `fd` is a member of `fdset`, zero otherwise. The behavior of these macros is undefined if a descriptor value is less than zero or greater than or equal to `FD_SETSIZE`, which is normally at least equal to the maximum number of descriptors supported by the system.

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

Any of `readfds`, `writefds`, and `exceptfds` may be given as NULL pointers if no descriptors are of interest.

Selecting true for reading on a socket descriptor upon which a `listen()` call has been performed indicates that a subsequent `accept()` call on that descriptor will not block.

RETURN VALUES

`Select()` returns the number of ready descriptors that are contained in the descriptor sets, or -1 if an error occurred. If the time limit expires (or signal in mask arrives), the functions returns 0. If error is returned, including one due to an interrupted call, the descriptor sets will be unmodified.

ERRORS

An error return from `select()` indicates:

[EBADF]

One of the descriptor sets specified an invalid descriptor.

[EINTR]

A signal was delivered before the time limit expired and before any of the selected events occurred.

[EINVAL]

The specified time limit is invalid. One of its components is negative or too large.

SEE ALSO

`accept()`, `connect()`, `recv()`, `send()`, `getdtablesize()`,
`SocketBaseTagList()`

NOTES

`Select()` is actually a stub/inline function that calls `WaitSelect()` with a signal mask set as NULL pointer.

BUGS

`WaitSelect()` should probably return the time remaining from the original timeout, if any, by modifying the time value in place. This may be implemented in future versions of the system. Thus, it is unwise to assume that the timeout value will be unmodified by the `WaitSelect()` call.

HISTORY

The `select()` function call appeared in 4.2BSD.

1.28 bsdsocket.guide/send()

`send()`, `sendto()`, `sendmsg()`
 =====

NAME

`send`, `sendto`, `sendmsg` -- send a message from a socket

SYNOPSIS

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

```
int send(int s, const void * msg, int len, int flags)
D0          D0          A0          D1          D2
```

```
int sendto(int s, const void * msg, int len, int flags,
D0          D0          A0          D1          D2
           const struct sockaddr * to, int tolen)
           A1          D3
```

```
int sendmsg(int s, const struct msghdr * msg, int flags)
D0          D0          A0          D1
```

DESCRIPTION

Send() sendto() and sendmsg() are used to transmit a message to another socket. Send() may be used only when the socket is in a connected state, while sendto() and sendmsg() may be used at any time.

The address of the target is given by to with tolen specifying its size. The length of the message is given by len. If the message is too long to pass atomically through the underlying protocol, the error EMSGSIZE is returned, and the message is not transmitted.

No indication of failure to deliver is implicit in a send() Locally detected errors are indicated by a return value of -1.

If no messages space is available at the socket to hold the message to be transmitted, then send() normally blocks, unless the socket has been placed in non-blocking I/O mode. The select() call may be used to determine when it is possible to send more data.

The flags parameter may include one or more of the following:

```
#define MSG_OOB      0x1  /* process out-of-band data */
#define MSG_DONTROUTE 0x4  /* bypass routing, use direct interface */
```

The flag MSG_OOB is used to send "out-of-band" data on sockets that support this notion (e.g. SOCK_STREAM); the underlying protocol must also support "out-of-band" data. MSG_DONTROUTE is usually used only by diagnostic or routing programs.

See recv() for a description of the msghdr structure.

RETURN VALUES

The call returns the number of characters sent, or -1 if an error occurred.

ERRORS

Send() sendto() and sendmsg() fail if:

[EBADF]

An invalid descriptor was specified.

[EMSGSIZE]

The socket requires that message be sent atomically, and the size of the message to be sent made this impossible.

[EWOULDBLOCK]

The socket is marked non-blocking and the requested operation would block.

[ENOBUFS]

The system was unable to allocate an internal buffer. The operation may succeed when buffers become available.

[ENOBUFS]

The output queue for a network interface was full. This generally indicates that the interface has stopped sending, but may be caused by transient congestion.

SEE ALSO

IoctlSocket(), recv(), select(), getsockopt(), socket()

HISTORY

The send() function call appeared in 4.2BSD.

1.29 bsdsocket.guide/SetErrnoPtr()

SetErrnoPtr()

=====

NAME

SetErrnoPtr -- set the "global" errno pointer (OBSOLETE)

SYNOPSIS

```
void SetErrnoPtr(void * ptr, int size)
                   A0      D0
```

FUNCTION

This functions allows caller to redirect error variable inside scope of caller task. Usually this is used to make task's global variable errno as error variable.

Since AMITCP/IP version 3.0 you can use function SocketBaseTagList() to set error value pointer.

INPUTS

ptr

pointer to error variable that is to be modified on every error condition on this library function.

size

size of the error variable. It can be 1, 2 or 4 bytes long.

EXAMPLE

```
#include <errno.h>

struct Library;
struct Library * SocketBase = NULL;

int main(void)
{
    ...
    if ((SocketBase = OpenLibrary("bsdsocket.library", 2))
        != NULL) {
        SetErrnoPtr(&errno, sizeof errno);
        ...
    }
}
```

NOTES

Be sure that this new error variable exists until library base is finally closed or SetErrnoPtr() is called again for another variable.

SEE ALSO

Errno(), SocketBaseTagList()

1.30 bsdsocket.guide/SetSocketSignals()

SetSocketSignals()

=====

NAME

SetSocketSignals -- set SIGINTR, SIGIO and SIGURG signals
(OBSOLETE).

SYNOPSIS

```
void SetSocketSignals(ULONG intrmask, ULONG iomask, ULONG urgmask)
                        D0                D1                D2
```

FUNCTION

SetSocketSignals() tells the AMITCP/IP which signal masks corresponds unix SIGINT, SIGIO and SIGURG signals to be used in current amiga software. The intrmask mask is used to determine which Amiga signals interrupt blocking library calls.

The iomask is sent when asynchronous notification of socket events is done and the urgmask is sent when out-of-band data arrives. The signals are sent only to the owning task of particular socket. The task that creates the socket is set to the owner of the socket by default. If the socket is inherited with the ObtainSocket() function, the owner must be set explicitly with the FIOSETOWN (SIOCSPGRP) IoctlSocket() call.

Note that the supplied values write over old ones. If this function is used and CTRL-C is still wanted to interrupt the calls (the default behaviour), the value SIGBREAKF_CTRL_C must be explicitly or'ed to the intrmask.

NOTES

The function SetSocketSignals() is obsoleted by the function SocketBaseTagList().

SEE ALSO

IoctlSocket(), recv(), send(), select(), SetSocketSignals()

1.31 bsdsocket.guide/shutdown()

shutdown()

=====

NAME

shutdown -- shut down part of a full-duplex connection

SYNOPSIS

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

int shutdown(int s, int how)
D0           D0       D1
```

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 0, further receives will be disallowed. If `how` is 1, further sends will be disallowed. If `how` is 2, further sends and receives will be disallowed.

DIAGNOSTICS

A 0 is returned if the call succeeds, -1 if it fails.

ERRORS

The call succeeds unless:

- [EBADF]
S is not a valid descriptor.
- [ENOTCONN]
The specified socket is not connected.

SEE ALSO

`connect()`, `socket()`

HISTORY

The `shutdown()` function call appeared in 4.2BSD.

1.32 bsdsocket.guide/socket()

```
socket()
=====
```

NAME

`socket` -- create an endpoint for communication

SYNOPSIS

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

int socket(int domain, int type, int protocol)
D0         D0       D1       D2
```

DESCRIPTION

`Socket()` creates an endpoint for communication and returns a descriptor.

The `domain` parameter specifies a communications domain within which communication will take place; this selects the protocol family which should be used. These families are defined in the include file `<sys/socket.h>`. The currently understood formats are

AF_UNIX	(UNIX internal protocols),
AF_INET	(ARPA Internet protocols),
AF_ISO	(ISO protocols),
AF_NS	(Xerox Network Systems protocols), and
AF_IMPLINK	(IMP ``host at IMP'' link layer).

AMITCP/IP currently supports only AF_INET protocol family.

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

```
SOCK_STREAM
SOCK_DGRAM
SOCK_RAW
SOCK_SEQPACKET
SOCK_RDM
```

A SOCK_STREAM type provides sequenced, reliable, two-way connection based byte streams. An out-of-band data transmission mechanism may be supported. A SOCK_DGRAM socket supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length). A SOCK_SEQPACKET socket may provide a sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length; a consumer may be required to read an entire packet with each read system call. This facility is protocol specific, and presently implemented only for PF_NS. SOCK_RAW sockets provide access to internal network protocols and interfaces. The types SOCK_RAW, which is available only to the super-user, and SOCK_RDM, which is planned, but not yet implemented, are not described here.

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 particular to the "communication domain" in which communication is to take place.

Sockets of type SOCK_STREAM are full-duplex byte streams, similar to pipes. 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() call. Once connected, data may be transferred using recv() and send() or their variant calls. When a session has been completed a CloseSocket() may be performed. Out-of-band data may also be transmitted as described in send() and received as described in recv().

The communications protocols used to implement a SOCK_STREAM insure 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 broken and calls will indicate an error with -1 returns and with ETIMEDOUT as the specific error code (see Errno()). The protocols optionally keep sockets "warm" by forcing transmissions roughly every minute in the absence of other

activity. An error is then indicated if no response can be elicited on an otherwise idle connection for a extended period (e.g. 5 minutes).

SOCK_SEQPACKET sockets employ the same system calls as SOCK_STREAM sockets. The only difference is that `recv()` calls will return only the amount of data requested, and any remaining in the arriving packet will be discarded.

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

An `IoctlSocket()` call can be used to specify a task to receive a SIGURG signal when the out-of-band data arrives. It may also enable non-blocking I/O and asynchronous notification of I/O events via SIGIO.

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

RETURN VALUES

A -1 is returned if an error occurs, otherwise the return value is a descriptor referencing the socket.

ERRORS

The `socket()` call fails if:

[EPROTONOSUPPORT]

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

[EMFILE]

The per-process descriptor table is full.

[EACCESS]

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

[ENOBUFFS]

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

SEE ALSO

`accept()`, `bind()`, `connect()`, `CloseSocket()`, `getprotobyname()`, `getsockname()`, `getsockopt()`, `IoctlSocket()`, `listen()`, `recv()`, `select()`, `send()`, `shutdown()`, `SocketBaseTagList()`

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

'BSD Interprocess Communication Tutorial', reprinted in UNIX Programmer's Supplementary Documents Volume 1.

NOTES

socket() calls the fdCallback() with action codes FDCB_CHECK and FDCB_ALLOC to check and mark the new descriptor as allocated if the callback is defined. See SocketBaseTagList() for more information on fdCallback().

HISTORY

The socket() function call appeared in 4.2BSD.

1.33 bsdsocket.guide/SocketBaseTagList()

SocketBaseTagList()
=====

NAME

SocketBaseTagList -- Set/Get SocketBase attributes.

SYNOPSIS

```
#include <amitcp/socketbasetags.h>

ULONG SocketBaseTagList(struct TagItem * taglist);
D0                                         A0

ULONG SocketBaseTags(ULONG tag, ...);
```

FUNCTION

Set or get a list of (mostly) SocketBase instance dependent attributes from the AMITCP/IP.

INPUTS

These functions expect as their argument a standard tag list, one or several array of struct TagItem as defined in the header file <utility/tagitem.h>. The structure contains two fields: ti_Tag and ti_Data. The ti_Tag field contains tag code, which determines what the SocketBaseTagList() should do with its argument, the ti_Data field.

The include file <amitcp/socketbasetags.h> defines macros for base tag code values. Base tag code macros begin with 'SBTC_' (as Socket Base Tag Code). The base tag value defines what data item the tag item refers.

The tag code contains other information besides the referred data item. It controls, whether the SocketBaseTagList() should set or get the appropriate parameter, and whether the argument of the tag in question is passed by value or by reference.

The include file <amitcp/socketbasetags.h> defines the following macros, which are used to construct the ti_Tag values from the base tag codes:

```
SBTM_GETREF(code) - get by reference
SBTM_GETVAL(code) - get by value
SBTM_SETREF(code) - set by reference
SBTM_SETVAL(code) - set by value
```

If the actual data is stored directly into the `ti_Data` field, you should use the 'by value' macros, `SBTM_GETVAL()` or `SBTM_SETVAL()`. However, if the `ti_Data` field contains a pointer to actual data, you should use the 'by reference' macros, `SBTM_GETREF()` or `SBTM_SETREF()`. In either case the actual data should always be a LONG aligned to even address.

According the used tag naming scheme a tag which has "PTR" suffix takes an pointer as its argument. Don't mix the pointer arguments with 'by reference' argument passing. It is possible to pass a pointer by reference (in which case the `ti_Data` is a pointer to the actual pointer).

The list of all defined base tag codes is as follows:

SBTC_BREAKMASK

Tag data contains the INTR signal mask. If the calling task receives a signal in the INTR mask, the AMITCP/IP interrupts current function calls and returns with the error code `EINTR`. The INTR mask defaults to the CTRL-C signal (`SIGBREAKF_C`, bit 12).

SBTC_DTABLESIZE

Socket Descriptor Table size. This defaults to 64.

SBTC_ERRNO

The `errno` value. The values are defined in `<sys-errno.h>`.

SBTC_ERRNOBYTEPTR

SBTC_ERRNOWORDPTR

SBTC_ERRNOLONGPTR

SBTC_ERRNOPTR(size)

Set (only) the pointer to the `errno` variable defined by the program. AMITCP/IP defines a value for this by default, but the application must set the pointer (and the size of the `errno`) with one of these tags, if it wishes to access the `errno` variable directly.

The `SBTC_ERRNOPTR(size)` is a macro, which expands to one of the other (BYTE, WORD or LONG) tag codes, meaning that only 1, 2 and 4 are legal size values.

The `NetLib:autoinit.c` sets the `errno` pointer for the application, if the application is linked with it.

SBTC_ERRNOSTRPTR

Returns an error string pointer describing the `errno` value given on input. You can not set the error message, only get is allowed.

On call the `ti_Data` must contain the error code number. On return the `ti_Data` is assigned to the string pointer. (`*ti_Data`, if passed by reference). See the file `<sys-errno.h>` for symbolic definitions for the `errno` codes.

SBTC_FD CALLBACK

A callback function pointer for coordination of file

descriptor usage between AMITCP/IP and link-library. By default no callback is called and the value of this pointer is NULL. The prototype for the callback function is:

```
int error = fdCallback(int fd, int action);
D0                D0      D1
```

where
error

- 0 for success or one of the error codes in <sys-errno.h> in case of error. The AMITCP/IP API function that calls the callback usually returns the error back to the caller without any further modification.

fd

- file descriptor number to take action on.

action

- one of the action codes, which are defined in the header file <amitcp-socketbasetags.h>) as follows:

FDCB_FREE

- mark the fd as unused on the link library structure. If fd represents a file handled by the link library, the error ENOTSOCK should be returned.

FDCB_ALLOC

- mark the fd allocated as a socket.

FDCB_CHECK

- check if the fd is free. If an error is returned, the fd is marked as used in the AMITCP/IP structures.

The AMITCP/IP calls the callback every time a socket descriptor is allocated or freed. AMITCP/IP uses the FDCB_CHECK before actual allocation to check that it agrees with the link library on the next free descriptor number. Thus the link library doesn't need to tell the AMITCP/IP if it creates a new file handle in open(), for example.

See file _chkufb.c on the net.lib sources for an example implementation of the callback function for the SAS/C.

SBTC_HERRNO

The name resolver error code value. Get this to find out why the gethostbyname() or gethostbyaddr() failed. The values are defined in <netdb.h>.

SBTC_HERRNOSTRPTR

Returns host error string for error number in tag data. Host error is set on unsuccessful gethostbyname() and gethostbyaddr() calls. See the file <netdb.h> for the symbolic definitions for the herrno valus.

Notes for the SBTC_ERRNOSTRPTR apply also to this tag code.

SBTC_IOERRNOSTRPTR

Returns an error string for standard AmigaOS I/O error number as defined in the header file <exec-errors.h>. Note that the error number taken by this tag code is positive, so the error codes must be negated (to be positive). The positive error codes depend on the particular IO device, the standard Sana-II error codes can be retrieved by the tag code SBTC_S2ERRNOSTRPTR.

Notes for the SBTC_ERRNOSTRPTR apply also to this tag code.

SBTC_LOGFACILITY

Facility code for the syslog messages as defined in <sys/syslog.h>. Defaults to LOG_USER.

SBTC_LOGMASK

Sets the filter mask of the syslog messages. By default the mask is 0xff, meaning that all messages are passed to the log system.

SBTC_LOGSTAT

Syslog options defined in <sys/syslog.h>.

SBTC_LOGTAGPTR

A pointer to a string which is used by Syslog() to mark individual syslog messages. This defaults to NULL, but is set to the name of the calling program by the autoint code in netlib:autoint.c. This is for compatibility with pre-3.0 programs.

SBTC_S2ERRNOSTRPTR

Returns an error string for a Sana-II specific I/O error code as defined in the header file <devices-sana2.h>.

Notes for the SBTC_ERRNOSTRPTR apply also to this tag code.

SBTC_S2WERRNOSTRPTR

Returns an error string for a Sana-II Wire Error code as defined in the header file <devices-sana2.h>.

Notes for the SBTC_ERRNOSTRPTR apply also to this tag code.

SBTC_SIGEVENTMASK

Tag data contains the signal mask to be sent to the application whenever notification about socket events is in order. The default value for this is zero, inhibiting any event notifications. The application must set this mask if it desires to be notified about asynchronous socket events. When the application receives the signal specified in the mask, it can use the function GetSocketEvents() to find out what happened.

SBTC_SIGIOMASK

The signals specified in the mask in the tag data are sent to the calling task when asynchronous I/O is to be notified. The

default value is zero, i.e., no signals are sent. The signals in the mask are sent whenever something happens on the socket. This mechanism is compatible with the Unix SIGIO signal.

Since AmigaOS signals may get combined, one signal may include notification for originally distinct events on the socket. One example of this is the reception of data and connection closure.

Usage of the socket events (see `GetSocketEvents()`) is recommended over this because of the problem described above.

SBTC_SIGURGMASK

The signals specified in the mask in the tag data are sent to the calling task when notification about urgent data arrives. The default value is zero, ie. no signals are sent. This mechanism is compatible with the Unix SIGURG signal.

Note that this signal does not indicate the arrival of the actual out-of-band data. If the receive buffer of the socket is full, the urgent data can't even be received. Because of this the application may need to read some normal data off the socket before it can read the urgent data.

RESULT

Returns 0 on success, and a (positive) index of the failing tag on error. Note that the value 1 means first TagItem, 2 the second one, and so on. The return value is NOT a C-language index, which are 0 based.

EXAMPLES

To be written, see net.lib sources for various examples.

SEE ALSO

<netinclude:amitcp-socketbasetags.h>, <include:utility-tagitem.h>

1.34 bsdsocket.guide/syslog()

`syslog()`, `vsyslog()`
=====

NAME

`syslog`, `vsyslog` -- write message to AMITCP/IP log

SYNOPSIS

```
#include <syslog.h>
```

```
void syslog(int priority, const char * message, ...)
```

```
void vsyslog(int level, const char * format, LONG * args)
               D0                A0                A1
```

DESCRIPTION

Writes the message given as format string and arguments (printf-style) both to the log file and to the console. The message is prepended with the name of the calling application, if the name is known by AMITCP/IP (the standard autoinitializer module in the net.lib passes the name of the application to AMITCP/IP).

The message is identical to a printf() format string, except that %m is replaced by the current error message. (As denoted by the library variable errno (see Errno()).) A trailing newline is added if none is present.

The level is selected from an ordered list:

LOG_EMERG

A panic condition.

LOG_ALERT

A condition that should be corrected immediately, such as a corrupted system database.

LOG_CRIT

Critical conditions, e.g., hard device errors.

LOG_ERR

Errors.

LOG_WARNING

Warning messages.

LOG_NOTICE

Conditions that are not error conditions, but should possibly be handled specially.

LOG_INFO

Informational messages.

LOG_DEBUG

Messages that contain information normally of use only when debugging a program.

INPUTS

Level

indicates the type of the message. The levels are defined in sys/syslog.h and listed above.

format

This is a printf-style format string. In addition %m is expanded to current error message.

...

arguments as in printf().

args

pointer to an array of arguments.

RESULT

Returns no value.

EXAMPLES

To log a message at priority LOG_INFO, it would make the following call to syslog:

```
syslog(LOG_ALERT, "who: internal error 23");
syslog(LOG_INFO, "Connection from host %s", CallingHost);
```

NOTES

In contrast to the previous releases of the AMITCP/IP, the integer arguments are expected to be 32 bits wide, thus eliminating the need to specify the 'l' size modifier for the number formatters.

This function is callable from interrupts.

BUGS

Because there is a limited number of internal messages used by the logging system, some log messages may get lost if a high priority task or interrupt handler sends many messages in succession. If this happens, the next log message tells the fact.

SEE ALSO

src/netlib/syslog.c for syslog utility functions (openlog(), closelog() and setlogmask()), C-library printf() documentation

HISTORY

These functions appeared in 4.2BSD.

1.35 bsdsocket.guide/Function Reference

Function Index

accept	accept()
bind	bind()
CloseSocket	CloseSocket()
connect	connect()
Dup2Socket	Dup2Socket()
Errno	Errno()
getdtablesize	getdtablesize()
gethostbyaddr	gethostbyname()
gethostbyname	gethostbyname()
gethostid	gethostid()
gethostname	gethostname()
getnetbyaddr	getnetbyname()
getnetbyname	getnetbyname()
getpeername	getpeername()
getprotobyname	getprotobyname()
getprotobynumber	getprotobyname()
getservbyname	getservbyname()
getservbyport	getservbyname()
GetSocketEvents	GetSocketEvents()
getsockname	getsockname()

getsockopt	getsockopt ()
htonl	htonl ()
htons	htonl ()
inet_addr	inet_addr ()
inet_lnaof	inet_addr ()
Inet_LnaOf	inet_addr ()
inet_makeaddr	inet_addr ()
Inet_MakeAddr	inet_addr ()
inet_netof	inet_addr ()
Inet_NetOf	inet_addr ()
inet_network	inet_addr ()
inet_ntoa	inet_addr ()
Inet_Ntoa	inet_addr ()
IoctlSocket	IoctlSocket ()
listen	listen ()
ntohl	htonl ()
ntohs	htonl ()
ObtainSocket	ObtainSocket ()
recv	recv ()
recvfrom	recv ()
recvmsg	recv ()
ReleaseCopyOfSocket	ReleaseSocket ()
ReleaseSocket	ReleaseSocket ()
select	select ()
send	send ()
sendmsg	send ()
sendto	send ()
SetErrnoPtr	SetErrnoPtr ()
SetSocketSignals	SetSocketSignals ()
setsockopt	getsockopt ()
shutdown	shutdown ()
socket	socket ()
SocketBaseTagList	SocketBaseTagList ()
syslog	syslog ()
vsyslog	syslog ()
WaitSelect	select ()
