EUROSplus TCP/IP Networking

Programming Guide and Reference

Document version: 11/2014

EUROS Embedded Systems GmbH
Campestraße 12 | D-90419 Nuremberg | Germany
Phone: +49-911-300328-0 | Fax: +49-911-300328-9
Web: www.euros-embedded.com
eMail: support@euros-embedded.com

Rev.	Changes	Date
-001	Original issue	06.08.1998
-002	Minor corrections	14.09.1998
-003	Minor corrections	01.10.1998
-004	Corrected TCP server example	17.02.1999
-005	Corrected chapter numbering; added #include statements in reference	02.07.1999
-006	Reformatted socket level options; added TCP level options	04.08.1999
-007	Minor corrections	30.08.1999
-008	New title sheet Symbols for caller context	18.10.1999
-009	netctl() codes GETTCPCONNS and GETUDPCONNS	16.03.2000
-010	Error corrections	03.04.2000
-011	IP level options for getsockopt/setsockopt	20.04.2000
-012	Resolver functions	18.05.2000
-013	netctl() code GETROUTES	31.05.2000
-014	Removed description of netctl(GETIFSTATS)	15.06.2000
-015	BootRequest function	31.08.2000
-016	NetInit parameters	09.03.2001
-017	Longer explanation of netctl parameters	21.05.2001
-018	Small error correction	07.06.2001
-019	netctl function SIOCGIFLLADDR	13.07.2001
-020	Corrected prototypes of sethostname and gethostname	19.07.2001
-021	Reformatted soioctl	28.08.2001
-022	New chapter about network interface management	06.09.2001
-023	Expanded netctl() reference	24.05.2002
-024	Corrected a few function descriptions	05.03.2004
-025	SIOCGIFSTAT	27.08.2004
-026	Function NetMemStatus()	08.09.2004
-027	NetInit()	25.02.2005
-028	NetInitJumbo()	10.10.2005
-029	Adaptive window scaling	12.06.2007

Rev.	Changes	Date
-030	Zero copy data sending	14.04.2010
-031	Improved timeout handling for all sockets	20.09.2011
-032	Route and ARP cache API exposed for use	15.02.2012
-033	Reduced memory usage	23.01.2013
-034	Multicast filtering	16.12.2014

General remarks

This product information describes the present status of development. Modifications therefore are reserved.

This product information was prepared with utmost care. However, no guarantee or liability for the faultlessness and correctness of the contents can be taken upon.

Printed in Germany

The EUROS*plus* documentation

The Documentation of the operating system EUROS*plus* is devided in four manuals: EUROS*plus* Programmer's Guide, EUROS*plus* User's Guide, EUROS*plus* Reference Manual and the EUROS*plus* Installation Guide, which are part of the EUROS*plus* development licence. The four basic manuals of the operating system have the following goals:

EUROS*plus* Programmer's Guide

The Programmer's Guide gives an overview over the concepts, the components and the system services of the operating system EUROS*plus*. The EUROS*plus* components and system objects are introduced and their properties and use are described.

EUROS*plus* Reference Manual

The EUROS*plus* Reference Manual contains detailled and complete descriptions of the system calls implemented under EUROS*plus*. It is the basic tool in order to write succesfully applications under the operating system EUROS*plus*. The system services of the Microkernel, I/O System, Process Manager, C-Library and the POSIX Interface are described.

EUROS*plus* Installation Guide

The EUROS*plus* Installation Guide contains information for the system administrator concerning the configuration, installation and adaption of the operating system. It is included in the development package and describes, how the adapt a target board monitor, how to configure a timer, an UART and an interrupt controller used by the operating system and how to dimension the EUROS*plus* data areas.

Table of Contents

Chapter 1
Socket programming guide

Chapter 2
Socket function reference

Definitions

The following notational conventions are used for this manual:

`Block print`	User input, examples, name of variables and functions are displayed in block print.
<CR>	Non-printable characters are displayed as their names in angle brackets.
[ ]	Options and optional parameters are displayed in square brackets.
|	Options and parameters of which exactly one can be used are seperated by a vertical line.
`M`	Function may be called in `main()`.
I	Function may be called in I state.
N	Function may be called in N state.
S	Function may be called in S state.
A	Function may be called in A state.

Chapter 1
Socket programming guide

1.1 Installation

1.1.1 Files

The following files are shipped with the Network Manager:

`net.lib`	Library (Debug and No-Debug version)
`socket.h`	C header file for main socket calls
`netctl.h`	C header file for `netctl`
`sockio.h`	C header file for `soioctl`
`*_var.h`	Network statistics structures
`resolv.h`	C header file for resolver
`route.h`	Structures for routing table manipulation
`services.h`	Definitions of standard port numbers
`types.h`	Networking data types
`if.h`	Structures for network interface manipulation
`if_arp.h`	Structures for ARP cache manipulation
`if_types.h`	Definitions of interface types

1.1.2 Debug version/No-Debug version

The Network Manager library is shipped in a Debug version and No-Debug version. The Debug version should be used when developing networking applications. The No-Debug version should be used for production code.

The main differences between the two versions are:

- The Debug version prints additional information on the console, e.g. protocol problems, sent and received ICMP messages etc.
- The Debug version performs parameter checking and stack checking.
- In the No-Debug version, switching on the `SO_DEBUG` socket option has no effect.
- Some `netctl` options are not supported in the No-Debug version.

1.2 Managing network interfaces

A network interface is a special EUROS driver (Resource Manager or Port Driver) that is used by the network component to send and receive IP datagrams (and possibly other data like ARP frames). Network interfaces have names. These names are assigned when attaching the interface to the network component (see chapter 1.2.1) and are later used with certain system calls when referring to a specific interface. Interface names are independent from I/O object names and are only known within the network component. Interface names must be unique.

The EUROS network component has the ability to manage more than a single network interface. IP datagrams can be received and sent on all interfaces concurrently. Which interface is used when sending an IP datagram depends on the interface addresses and on the routing table.

1.2.1 Attaching network interfaces to the network component

The driver object must be open when it is attached to the Network Manager. The following example illustrates how to attach an interface to the Network Manager:

```
#include <net/netctl.h>
#include <net/if.h>

int ChannelId; /* created with IoCreate and opened with IoOpen */

struct ifattach myattach;

myattach.UnitId = ChannelId;
myattach.pName = "MyIf0";

netctl(ATTACHINTERFACE, &myattach, sizeof(myattach));

/* configure interface... */
```

1.2.2 Interface flags

Every interface has its own set of addresses and flags. The interface flags and other characteristics are communicated from the driver to the network component. Most flags cannot be changed and describe certain characteristics of the interface type. The interface flags can be queried using the `netctl` call.

Example:

```
struct ifreq ifreq;
strcpy(ifreq.ifr_name, "ppp0");
netctl(SIOCGIFFLAGS, &ifreq, sizeof(ifreq));
/* flags are returned in ifreq.ifr_flags */
```

The interface flags are:

`IFF_UP`	The interface is activated, i.e. it can now send and receive data.
`IFF_BROADCAST`	Broadcast datagrams can be sent on this interface.
`IFF_DEBUG`	Debugging is enabled. How debugging affects the interface depends on the driver.
`IFF_LOOPBACK`	The interface is a loopback interface, i.e. sent datagrams are looped back as received datagrams.

IFF_POINTOPOINT	The interface is a point-to-point link, i.e. there is exactly one other host at the other end of the link that can be reached directly. Otherwise more than one host can be reached directly.
IFF_NOARP	The address resolution protocol can not be used with this interface.
IFF_SIMPLEX	The interface does not receive its own transmissions.
IFF_MULTICAST	The interface supports multicast addressing.

Of these flags only IFF_UP and IFF_DEBUG can be changed by the application. Network interfaces are de-activated by default. Setting the IFF_UP flag activates the interface.

Please note that setting IFF_UP may not be effective immediately. For example, with PPP interfaces setting this flag only starts the dialing and negotiation of connection parameters. The application has to read back the interface flags to see if the interface has completed this process, indicating this by actually setting the flag.

Example:

```
struct ifreq ifreq;


strcpy(ifreq.ifr_name, "ppp0");


/* bring up interface */
ifreq.ifr_flags = IFF_UP;
netctl(SIOCSIFFLAGS, &ifreq, sizeof(ifreq));


/* wait until interface is up */
do
{
   TaskSleep(MS|100);
   netctl(SIOCGIFFLAGS, &ifreq, sizeof(ifreq));
} while (!(ifreq.ifr_flags & IFF_UP));
```

1.2.3 Interface addresses

Hosts don't have IP addresses, interfaces do.

Every IP interface has at least its own local IP address. Point-to-point interfaces (like PPP connections) have a second address, the address of the peer at the other end of the line. Broadcast interfaces (like Ethernet) have a network mask instead.

Local address, peer address ("destination address"), network mask and broadcast address are assigned to the interface using the netctl call.

Example:

```
struct ifreq ifreq;
struct sockaddr_in addr;


strcpy(ifreq.ifr_name, "lan");


inet_aton("192.168.1.1", &addr.sin_addr); /* IP address */
```

```
addr.sin_len = sizeof(addr);
addr.sin_family = AF_INET;
addr.sin_port = 0; /* unused */
memcpy(&ifreq.ifr_addr, &addr, sizeof(addr));
netctl(SIOCSIFADDR, &ifreq, sizeof(ifreq));
```

See chapter 2 for a complete description of `netctl`.

1.3 Socket programming

1.3.1 Definitions

Network byte order

Order of bytes in multibyte data as it occurs on the network. For TCP/IP the network byte order is “big endian”, i.e. higher order bytes are transmitted first.

Host byte order

Native order of bytes in multibyte data on a host. The order depends on the CPU and operating system and may be different from the network byte order. Multibyte data usually must be converted to network byte order before transmission.

Address

IP address/port number pair specified in a `struct sockaddr_in` (see `net/socket.h`). Address and port number must be in network byte order. `INADDR_ANY` and 0 can be used as wildcard addresses.

Socket

Data structure internal to the network component.

Socket descriptor

Integer value identifying a socket. Socket descriptors are positive non-null values. Socket descriptors are not EUROS object IDs, so they can’t be used with EUROS’ `Object...()` functions.

Connection

1:1 relationship between a client and a server. Connections must be established before data can be transferred between both ends. After all data has been exchanged, the connection must be closed. After that, no more data can be exchanged.

Peer

Other (non-local) side of a connection.

Client

Program or node initiating a connection (active open) to a server.

Server

Program or node accepting connections (passive open) from clients.

1.3.2 Preparing tasks for network programming

In order to use the Network Manager a task must have enough stack space. The stack space required by the Network Manager varies from CPU to CPU. A task should have at least 800 bytes of stack.

Since threads originating from network interface drivers also use the Network Manager, the thread stack must be made large enough. This is done in the configuration table of the application.

1.3.3 Basic data structures

Socket address

Socket functions expect addresses passed in a `struct sockaddr` structure. This structure has the following components:

`sa_len`	Length of the entire structure
`sa_family`	Family of address contained in this structure. Must contain one of the `AF_*` values.
`sa_data`	Address data. The format of this field varies with each address family.

The EUROS Network Manager only supports the `AF_INET` address family (see below).

Socket address (Internet)

The structure `struct sockaddr_in` is a special version of the `struct sockaddr` structure. It is used to specify addresses of the Internet address family (`AF_INET`). This structure has the following components:

`sin_len`	Length of the entire structure (must be `sizeof(struct sockaddr_in)`)
`sin_family`	Family of address contained in this structure. Must be `AF_INET`.
`sin_port`	Port address in network byte order. This component is ignored when only the IP address is required (e.g. when specifying an interface address or when adding routes).
`sin_addr`	IP address in network byte order.
`sin_zero`	Reserved, must be zero-filled.

Since all socket functions expect a pointer to `struct sockaddr` instead of `struct sockaddr_in`, a typecast must be used when passing a pointer to a `struct sockadd_in`.

1.3.4 Typical TCP client application

The following program excerpt illustrates the typical flow of a TCP client application. For real-world applications additional error checking is required.

```
/* TCP client */

#include <net/socket.h>
#include <net/services.h>

...
int s;
char buf[32] = "Hello";
struct sockaddr_in server;

/* prepare server address */
server.sin_family = AF_INET;
server.sin_len = sizeof(server);
server.sin_port = htons(TCPSERV_ECHO); /* Port 7 (Echo)*/
server.sin_addr.s_addr = inet_addr("1.2.3.4");

/* create stream socket */
if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0)
{
   /* error */
   return 1;
}
```

```
/* connect to server */
if (connect(s, (struct sockaddr*)&server, sizeof(server)) < 0)
{
   /* error */
   return 1;
}

/* send data */

if (send(s, buf, sizeof(buf), 0) < 0)
{
   /* error */
   return 1;
}

/* receive echo */
if (recv(s, buf, sizeof(buf), 0) < 0)
{
   /* error */
   return 1;
}

/* success, close socket */
soclose(s);

...
```

1.3.5 Typical TCP server application

The following program excerpt illustrates the typical flow of a TCP server application. For real-world applications additional error checking is required.

```
/* TCP Server */

#include <types.h>
#include <net/socket.h>
#include <net/services.h>

int s;
char buf[32];
struct sockaddr_in server, client;
int ns, namelen;

/* create socket */
if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0)
{
   /* error */
   return 1;
}

/* bind socket to address and port */
server.sin_family = AF_INET;
```

```
server.sin_port = htons(TCPSERV_ECHO); /* Port 7 (Echo) */
server.sin_addr.s_addr = INADDR_ANY; /* any local addr.*/
server.sin_len = sizeof(server);

if (bind(s, (struct sockaddr*)&server, sizeof(server)) < 0)
{
   /* error */
   return 1;
}

/* listen for connection, max. 1 queued connections */
if (listen(s, 1) != 0)
{
   /* error */
   return 1;
}

/* accept connection */
namelen = sizeof(client);
if ((ns = accept(s, (struct sockaddr*)&client, &namelen)) < 0)
{
   /* error */
   return 1;
}

/* receive data */
if (recv(ns, buf, sizeof(buf), 0) < 0)
{
  /* error */
  return 1;
}

/* echo back data */
if (send(ns, buf, sizeof(buf), 0) < 0)
{
   /* error */
   return 1;
}

soclose(ns);
soclose(s);
...
```

1.4 Technical information

1.4.1 Supported features

Application level protocols:

DNS-Resolver:

- Up to 3 name servers configurable
- configurable number of re-tries and interval between tries
- UDP queries and TCP queries (UDP is default)

BOOTP-Client:

- for automatic address assignment only

Transport protocols:

TCP with:

- Slow start and congestion avoidance
- Fast retransmit
- Window scaling
- keepalive
- delayed ACK
- Nagle algorithm

UDP with:

- optional UDP data checksumming

Internetwork protocols:

IPv4 with:

- optional datagram forwarding
- subnetting
- multicasting
- configurable TTL
- configurable TOS
- fragmentation and reassembly

ICMP

IGMP

ARP

Link layer protocols:

Point-to-point interfaces (PPP)

Broadcast interfaces (Ethernet, IEEE 802.2)

1.4.2 Changes compared to BSD sockets

- The `select()` call is not supported
- `read/write` on sockets is not supported, use `recv/send` instead
- the `close` call can not be used for sockets, use `soclose` instead
- the `ioctl` call can not be used for sockets, use `soioctl` instead
- interface parameters and routing parameters must be changed with `netctl`

- a reentrant version of `inet_ntoa` is added, called `inet_ntoa_r`
- there are no functions to handle the files SERVICES, PROTOCOLS or HOSTS. Instead, two header files `socket.h` and `services.h` are provided containing symbolic definitions for protocols and services.
- There is no built-in loopback interface.
- Timeout values are specified with an EUROS standard `TimeLimit` value in an `uint32`. See Reference Manual chapter 1 for details.

Chapter 2
Socket function reference

2.1 Initialization and configuration

The initialization and configuration functions are used to initialize and configure the Network Manager component and to set and query operational parameters.

Function prototypes, macros and data structures are defined in the C header files `socket.h` and `netctl.h`.

S
A

`NetInit` - Initialize network component

Syntax:

```
#include <net/socket.h>
int NetInit(uint16 NumSockets, uint16 NumClusters,
            uint16 NumBuffers, uint16 NumPcb,
            uint16 NumUtilBlocks);
```

Description:

Initialize Network Manager

Parameters:

`NumSockets`	Number of available sockets. When a socket is created using the `socket()` or `accept()` calls, one of these blocks is used.
`NumClusters`	Number of available clusters (large buffers, about 1600 bytes). These are used to buffer large amounts of protocol data. These buffers are used for incoming and outgoing packets as well as for storing data in a socket buffer.
`NumBuffers`	Number of available buffers (about 140 bytes). These are used to buffer small amounts of protocol data. These buffers are used for incoming and outgoing packets as well as for storing data in a socket buffer.
`NumPcb`	Number of available protocol control blocks. For each TCP socket one of these blocks is required. UDP sockets and raw IP sockets do not require an additional PCB. The network component actually allocates space for `NumSockets+NumPcb` blocks. Each TCP socket allocates two PCBs, other sockets allocate one PCB. `NumPcb` effectively specifies the maximum number of concurrent TCP sockets.
`NumUtilBlocks`	Number of available utility blocks. These are used to store other information of the network component like routes, interface definitions, interface addresses, ARP cache entries, TCP header templates and IP reassembly queue headers. The network component actually allocates space for `NumUtilBlocks+NumPcb` blocks. Each TCP connection requires one utility block.

Return values:

`OK`	Network component successfully initialized
`FAIL`	Initialization failed. The error cause can be read from `errno`, `errno_loc` and `errno_id`.

See also:

`NetMemStatus, NetInitJumbo`

Remarks:

All parameters must have a non-null value. The memory is taken from system memory. The system memory must be configured large enough to hold the data.

The exact memory requirements of the network component depend heavily on network traffic. You can derive initial values for the `NetInit()` parameters from the number of interfaces, routes, sockets and concurrent TCP connections and the sizes of your socket buffers. Due to the dynamic nature of networking,

you should adjust these values to expected or allowed network traffic during testing, especially `NumClusters`, `NumBuffers` and `NumUtilBlocks`.

Unlike the `Init*` functions of other components, `NetInit` must be called from a real EUROS task, not from `main()`.

`NetInit` ist implemented as a macro that calls `NetInitJumbo` with `NumJumboClusters=0`.

S
A

NetInitJumbo - Initialize network component with large buffers

Syntax:

```
#include <net/socket.h>
int NetInit(uint16 NumSockets, uint16 NumClusters,
            uint16 NumBuffers, uint16 NumJumboClusters,
            uint16 NumPcb, uint16 NumUtilBlocks);
```

Description:

Initialize Network Manager with large buffers

Parameters:

`NumSockets`	Number of available sockets. When a socket is created using the `socket()` or `accept()` calls, one of these blocks is used.
`NumClusters`	Number of available clusters (large buffers, about 1600 bytes). These are used to buffer large amounts of protocol data. These buffers are used for incoming and outgoing packets as well as for storing data in a socket buffer.
`NumBuffers`	Number of available buffers (about 140 bytes). These are used to buffer small amounts of protocol data. These buffers are used for incoming and outgoing packets as well as for storing data in a socket buffer.
`NumJumboClusters`	Number of available huge clusters (huge buffers, about 9100 bytes). These are used to buffer very large amounts of protocol data. These buffers are used for incoming and outgoing packets as well as for storing data in a socket buffer.
`NumPcb`	Number of available protocol control blocks. For each TCP socket one of these blocks is required. UDP sockets and raw IP sockets do not require an additional PCB. The network component actually allocates space for `NumSockets+NumPcb` blocks. Each TCP socket allocates two PCBs, other sockets allocate one PCB. `NumPcb` effectively specifies the maximum number of concurrent TCP sockets.
`NumUtilBlocks`	Number of available utility blocks. These are used to store other information of the network component like routes, interface definitions, interface addresses, ARP cache entries, TCP header templates and IP reassembly queue headers. The network component actually allocates space for `NumUtilBlocks+NumPcb` blocks. Each TCP connection requires one utility block.

Return values:

`OK`	Network component successfully initialized
`FAIL`	Initialization failed. The error cause can be read from `errno`, `errno_loc` and `errno_id`.

See also:

`NetMemStatus, NetInit`

Remarks:

See remarks for `NetInit`. The additional parameter `NumJumboClusters` specifies the number of buffers that are large enough to support Jumbo Frames.

S
A

NetMemStatus - Query status of network memory

Syntax:

```
#include <net/socket.h>
int NetMemStatus(tNetMemStatus *pStatus);
```

Description:

Query status of various network memory pools.

Parameters:

pStatus	Pointer to a structure of the type tNetMemStatus. After successful execution of this function, this structure contains information about the network's memory pools. The structure tNetMemStatus is explained below.

Return values:

OK	Status information obtained
FAIL	Execution failed. The error cause can be read from errno, errno_loc and errno_id.

See also:

NetInit

Remarks:

The structure tNetMemStatus contains the following fields:

SocketsStatus	Memory information about sockets, contained in a structure of the type tMemStatus. This structure is explained below.
ClustersStatus	Memory information about clusters, contained in a structure of the type tMemStatus. This structure is explained below.
BuffersStatus	Memory information about buffers, contained in a structure of the type tMemStatus. This structure is explained below.
PcbStatus	Memory information about PCBs, contained in a structure of the type tMemStatus. This structure is explained below.
UtilBlockStatus	Memory information about utility blocks, contained in a structure of the type tMemStatus. This structure is explained below.
JumboClustersStatus	Memory information about jumbo clusters, contained in a structure of the type tMemStatus. This structure is explained below.

The structure tMemStatus contains the following fields:

Size	Size of one block in the memory pool.
Avail	Number of available blocks in the pool.
Max	Maximum number of available blocks in the pool. This value is the same as that of the corresponding NetInit parameter.
Min	Minimum number of available blocks in the pool.

The information in these structures is purely informational and may change at any time, depending on network activity. It should be used for debugging purposes only.

S
A

netctl - Set parameters of network component

Syntax:

```
#include <net/netctl.h>
int netctl(uint16 Option, void *pData, size_t Size);
```

Description:

Set configuration data of the network component.

Parameters:

Option	Option code, see following pages
pData	Pointer to option data
Size	Size of option data

Return values:

OK	Option successfully set
FAIL	Option not set. The error cause can be read from errno, errno_loc and errno_id.

See also:

-

Remarks:

For every Option value, pData and Size have different meanings. All supported options are explained in detail on the following pages.

netctl(SETHOSTNAME) - Set host name

Syntax:

```
#include <net/netctl.h>
int netctl(SETHOSTNAME, void *pName, size_t Size);
```

Description:

Set name of this host.

Parameters:

Option	SETHOSTNAME
pName	Pointer to new host name
Size	Length of host name (including \0)

Return values:

OK	Name successfully set
FAIL	Name not set. The error cause can be read from errno, errno_loc and errno_id.

See also:

netctl(GETHOSTNAME), sethostname

Remarks:

The host name may be used by some application layer protocols like SMTP. Otherwise it is not used by the network stack itself.

This call is equivalent to sethostname().

S
A

netctl(GETHOSTNAME) - Get host name

Syntax:

```
#include <net/netctl.h>
int netctl(GETHOSTNAME, void *pBuffer, size_t Size);
```

Description:

Get name of this host.

Parameters:

Option	GETHOSTNAME
pBuffer	Pointer to buffer for host name
Size	Length of buffer. The buffer must be large enough to hold the host name and the terminating \0 character.

Return values:

OK	Name successfully retrieved
FAIL	Name not retrieved. The error cause can be read from errno, errno_loc and errno_id.

See also:

netctl(SETHOSTNAME), gethostname

Remarks:

The host name may be used by some application layer protocols like SMTP. Otherwise it is not used by the network stack itself.

When the host name was not set before calling netctl(GETHOSTNAME) an empty string is returned.

This call is equivalent to gethostname().

netctl(SETDEFTTL) - Set default TTL value

Syntax:

```
#include <net/netctl.h>
int netctl(SETDEFTTL, int *pValue, sizeof(int));
```

Description:

Sets the default TTL value for new sockets.

Parameters:

Option	SETDEFTTL
pValue	Pointer to int containing new TTL of outgoing IP datagrams. The default value is 64.
Size	Must be sizeof(int).

Return values:

OK	Value successfully set
FAIL	Value not set. The error cause can be read from errno, errno_loc and errno_id.

See also:

netctl(GETDEFTTL), setsockopt

Remarks:

The default TTL value is used when a new socket is created. It can then be changed on a per-socket basis by using setsockopt.

S
A

netctl(GETDEFTTL) - Get default TTL value

Syntax:

```
#include <net/netctl.h>
int netctl(GETDEFTTL, int *pBuffer, sizeof(int));
```

Description:

Gets the default TTL value for new sockets.

Parameters:

Option	GETDEFTTL
pBuffer	Pointer to int receiving TTL value of outgoing IP datagrams.
Size	Must be sizeof(int).

Return values:

OK	Value successfully received
FAIL	Value not received. The error cause can be read from errno, errno_loc and errno_id.

See also:

netctl(SETDEFTTL), getsockopt

Remarks:

The TTL value can be queried on a per-socket basis by using getsockopt.

netctl(SETFORWARDING) - Enable/disable IP forwarding

Syntax:

```
#include <net/netctl.h>
int netctl(SETFORWARDING, int *pBool, sizeof(int));
```

Description:

Enables and disables IP datagram forwarding.

Parameters:

Option	SETFORWARDING
pBool	Pointer to int containing either TRUE to enable forwarding or FALSE to disable forwarding.
Size	Must be sizeof(int).

Return values:

OK	Value successfully set
FAIL	Value not set. The error cause can be read from errno, errno_loc and errno_id.

See also:

netctl(GETFORWARDING)

Remarks:

By default IP datagram forwarding is disabled.

When forwarding is enabled all received IP datagrams not addressed to one of the local interfaces are forwarded to the destination or next hop router.

In the current version of the network stack IP forwarding can not be enabled.

S
A

netctl(GETFORWARDING) - Query IP forwarding

Syntax:

```
#include <net/netctl.h>
int netctl(GETFORWARDING, int *pBool, sizeof(int));
```

Description:

Queries the current status of IP datagram forwarding.

Parameters:

Option	GETFORWARDING
pBool	Pointer to int receiving either TRUE when forwarding is enabled or FALSE when forwarding is disabled.
Size	Must be sizeof(int).

Return values:

OK	Value successfully queried
FAIL	Value not queried. The error cause can be read from errno, errno_loc and errno_id.

See also:

netctl(SETFORWARDING)

Remarks:

By default IP datagram forwarding is disabled.

netctl(SETREDIR) - Enable/disable ICMP redirects

Syntax:

```
#include <net/netctl.h>
int netctl(SETREDIR, int *pBool, sizeof(int));
```

Description:

Enables or disables generation of ICMP redirect messages.

Parameters:

Option	SETREDIR
pBool	Pointer to int containing either TRUE to enable ICMP redirects or FALSE to disable ICMP redirects.
Size	Must be sizeof(int).

Return values:

OK	Value successfully set
FAIL	Value not set. The error cause can be read from errno, errno_loc and errno_id.

See also:

netctl(GETREDIR)

Remarks:

By default ICMP redirects are disabled.

S
A

netctl(GETREDIR) - Query ICMP redirects

Syntax:

```
#include <net/netctl.h>
int netctl(GETREDIR, int *pBool, sizeof(int));
```

Description:

Queries the current status of ICMP redirect message generation.

Parameters:

Option	GETREDIR
pBool	Pointer to int receiving either TRUE when ICMP redirects are enabled or FALSE when ICMP redirects area disabled.
Size	Must be sizeof(int).

Return values:

OK	Value successfully queried
FAIL	Value not queried. The error cause can be read from errno, errno_loc and errno_id.

See also:

netctl(SETREDIR)

Remarks:

By default ICMP redirects are disabled.

netctl(GETIPSTATS) - Get IP statistics

Syntax:

```
#include <net/netctl.h>
#include <net/ip_var.h>
int netctl(GETIPSTATS, struct ipstat *pBuffer, sizeof(struct ipstat));
```

Description:

Gets IP statistics.

Parameters:

Option	GETIPSTATS
pBuffer	Pointer to struct ipstat receiving IP statistics.
Size	Must be sizeof(struct ipstat).

Return values:

OK	Values successfully received
FAIL	Values not received. The error cause can be read from errno, errno_loc and errno_id.

See also:

-

Remarks:

The structure struct ipstat is defined in net/ip_var.h and contains the following elements:

ips_total	Total number of IP datagrams received
ips_badsum	Number of datagrams received with a bad checksum
ips_tooshort	Number of datagrams that were too short
ips_toosmall	Number of datagrams that did not contain enough data
ips_badhlen	Number of datagrams with an ip header length < data size
ips_badlen	Number of datagrams with an ip length < ip header length
ips_fragments	Number of datagram fragments received
ips_fragdropped	Number of datagram fragments dropped (duplicates, out of space)
ips_fragtimeout	Number of datagram fragments timed out
ips_forward	Number of datagrams forwarded
ips_cantforward	Number of datagrams received for unreachable destination
ips_redirectsent	Number of datagrams forwarded on the same network
ips_noproto	Number of datagrams received for unknown or unsupported protocol
ips_delivered	Number of datagrams delivered to upper level protocols
ips_localout	Total number of IP datagrams generated here
ips_odropped	Number of datagrams lost due to lack of buffer space, etc.
ips_reassembled	Number of datagrams that were reassembled ok
ips_fragmented	Number of datagrams sucessfully fragmented

`ips_ofragments`	Number of datagrams output fragments created
`ips_cantfrag`	Number of datagrams that couldn't be fragmented
`ips_badoptions`	Number of errors in option processing
`ips_noroute`	Number of datagrams discarded due to unavailable route
`ips_badvers`	Number of datagrams with IP version != 4
`ips_rawout`	Total number of raw ip datagrams generated

This function is intended to be used by a SNMP agent. The counters in the structure are of volatile nature and may change at any time. The information should only be used for statistics or informational purposes.

netctl(GETROUTESTATS) - Get routing statistics

Syntax:

```
#include <net/netctl.h>
#include <net/route.h>
int netctl(GETROUTESTATS, struct rtstat *pBuffer,
           sizeof(struct rtstat));
```

Description:

Gets routing statistics.

Parameters:

Option	GETROUTESTATS
pBuffer	Pointer to struct rtstat receiving routing statistics.
Size	Must be sizeof(struct rtstat).

Return values:

OK	Values successfully received
FAIL	Values not received. The error cause can be read from errno, errno_loc and errno_id.

See also:

netctl(GETROUTESTATS)

Remarks:

The structure struct rtstat is defined in net/route.h and contains the following elements:

rts_badredirect	Number of bogus redirect calls
rts_dynamic	Number of routes created by redirects
rts_newgateway	Number of routes modified by redirects
rts_unreach	Number of route lookups which failed
rts_wildcard	Number of route lookups satisfied by a wildcard

This function is intended to be used by a SNMP agent. The counters in the structure are of volatile nature and may change at any time. The information should only be used for statistics or informational purposes.

S
A

netctl(SETREASSTTL) - Set reassembly TTL value

Syntax:

```
#include <net/netctl.h>
int netctl(SETREASSTTL, int *pBuffer, sizeof(int));
```

Description:

Sets the reassembly TTL value for received datagram fragments.

Parameters:

Option	SETREASSTTL
pBuffer	Pointer to int containing reassembly TTL value.
Size	Must be sizeof(int).

Return values:

OK	Value successfully set
FAIL	Value not set. The error cause can be read from errno, errno_loc and errno_id.

See also:

netctl(GETREASSTTL)

Remarks:

Datagram fragments are discarded if the datagram is still incomplete when the reassembly TLL expires. The default value is 60.

`netctl(GETREASSTTL)` - Get reassembly TTL value

Syntax:

```
#include <net/netctl.h>
int netctl(GETREASSTTL, int *pBuffer, sizeof(int));
```

Description:

Gets the reassembly TTL value for received datagram fragments.

Parameters:

`Option`	`GETREASSTTL`
`pBuffer`	Pointer to `int` receiving reassembly TTL value.
`Size`	Must be `sizeof(int)`.

Return values:

`OK`	Value successfully queried
`FAIL`	Value not queried. The error cause can be read from `errno`, `errno_loc` and `errno_id`.

See also:

`netctl(SETREASSTTL)`

Remarks:

Datagram fragments are discarded if the datagram is still incomplete when the reassembly TLL expires. The default value is 60.

S
A

netctl(GETTCPSTATS) - Get TCP statistics

Syntax:

```
#include <net/netctl.h>
#include <net/tcp_var.h>
int netctl(GETTCPSTATS, struct tcpstat *pBuffer,
           sizeof(struct tcpstat));
```

Description:

Gets TCP statistics.

Parameters:

Option	GETTCPSTATS
pBuffer	Pointer to struct tcpstat receiving TCP statistics.
Size	Must be sizeof(struct tcpstat).

Return values:

OK	Values successfully received
FAIL	Values not received. The error cause can be read from errno, errno_loc and errno_id.

See also:

-

Remarks:

The structure struct tcpstat is defined in net/tcp_var.h and contains the following elements:

tcps_connattempt	Number of connections initiated
tcps_accepts	Number of connections accepted
tcps_connects	Number of connections established
tcps_drops	Number of connections dropped
tcps_conndrops	Number of embryonic connections dropped
tcps_closed	Number of connections closed (includes drops)
tcps_segstimed	Number of segments where we tried to get rtt
tcps_rttupdated	Number of times we succeeded
tcps_delack	Number of delayed acks sent
tcps_timeoutdrop	Number of connections dropped in retransmit timeout
tcps_rexmttimeo	Number of retransmit timeouts
tcps_persisttimeo	Number of persist timeouts
tcps_keeptimeo	Number of keepalive timeouts
tcps_keepprobe	Number of keepalive probes sent
tcps_keepdrops	Number of connections dropped in keepalive
tcps_sndtotal	Number of total packets sent
tcps_sndpack	Number of data packets sent

tcps_sndbyte	Number of data bytes sent
tcps_sndrexmitpack	Number of data packets retransmitted
tcps_sndrexmitbyte	Number of data bytes retransmitted
tcps_sndacks	Number of ack-only packets sent
tcps_sndprobe	Number of window probes sent
tcps_sndurg	Number of packets sent with URG only
tcps_sndwinup	Number of window update-only packets sent
tcps_sndctrl	Number of control (SYN|FIN|RST) packets sent
tcps_rcvtotal	Number of total packets received
tcps_rcvpack	Number of packets received in sequence
tcps_rcvbyte	Number of bytes received in sequence
tcps_rcvbadsum	Number of packets received with checksum errors
tcps_rcvbadoff	Number of packets received with bad offset
tcps_rcvshort	Number of packets received too short
tcps_rcvduppack	Number of duplicate-only packets received
tcps_rcvdupbyte	Number of duplicate-only bytes received
tcps_rcvpartduppack	Number of packets with some duplicate data
tcps_rcvpartdupbyte	Number of duplicate bytes in partially-duplicate packets
tcps_rcvoopack	Number of out-of-order packets received
tcps_rcvoobyte	Number of out-of-order bytes received
tcps_rcvpackafterwin	Number of packets with data after window
tcps_rcvbyteafterwin	Number of bytes received after window
tcps_rcvafterclose	Number of packets received after "close"
tcps_rcvwinprobe	Number of received window probe packets
tcps_rcvdupack	Number of received duplicate ACKs
tcps_rcvacktoomuch	Number of received ACKs for unsent data
tcps_rcvackpack	Number of received ACK packets
tcps_rcvackbyte	Number of bytes ACKed by received ACKs
tcps_rcvwinupd	Number of received window update packets
tcps_pawsdrop	Number of segments dropped due to PAWS
tcps_predack	Number of times header predict ok for ACKs
tcps_preddat	Number of times header predict ok for data packets
tcps_pcbcachemiss	Number of PCB cache misses

This function is intended to be used by a SNMP agent. The counters in the structure are of volatile nature and may change at any time. The information should only be used for statistics or informational purposes.

S
A

netctl(GETTCPCONNS) - Get TCP connections

Syntax:

```
#include <net/netctl.h>
#include <net/tcp_var.h>
int netctl(GETTCPCONNS, struct tcpconnlist *pBuffer, int Size);
```

Description:

Get information about TCP connections.

Parameters:

Option	GETTCPCONNS
pBuffer	Pointer to struct tcpconnlist receiving connection information.
Size	Must be either sizeof(int) or size of buffer space.

Return values:

OK	Values successfully received
FAIL	Values not received. The error cause can be read from errno, errno_loc and errno_id.

See also:

-

Remarks:

When Size is sizeof(int), the call returns the number of entries in the list. When Size > sizeof(int), connection entries are returned, up to the buffer size.

The structure struct tcpconnlist is defined in net/tcp_var.h and contains the following elements:

Number	Number of elements in the array connections.
connections	Array of structures containing connection information.

The structure struct tcpconnection contains the following elements:

LocalAddr	Local IP address of the connection, network byte order
ForAddr	Foreign IP address of the connection, network byte order
LocalPort	Local port number of the connection, network byte order
ForPort	Foreign port number of the connection, network byte order
State	Connection state as defined in MIB-2 (see RFC-1213, RFC-2012).

This function is intended to be used by a SNMP agent.

netctl(GETUDPSTATS) - Get UDP statistics

Syntax:

```
#include <net/netctl.h>
#include <net/udp_var.h>
int netctl(GETUDPSTATS, struct udpstat *pBuffer,
           sizeof(struct udpstat));
```

Description:

Gets UDP statistics.

Parameters:

Option	GETUDPSTATS
pBuffer	Pointer to struct udpstat receiving UDP statistics.
Size	Must be sizeof(struct udpstat).

Return values:

OK	Values successfully received
FAIL	Values not received. The error cause can be read from errno, errno_loc and errno_id.

See also:

-

Remarks:

The structure struct udpstat is defined in net/udp_var.h and contains the following elements:

udps_ipackets	Number of total input packets
udps_hdrops	Number of packets shorter than header
udps_badsum	Number of checksum errors
udps_badlen	Number of data length larger than packet
udps_noport	Number of no socket on port
udps_noportbcast	Number of above, arrived as broadcast
udps_fullsock	Number of datagrams not delivered, input socket full
udpps_pcbcachemiss	Number of input packets missing PCB cache
udps_opackets	Number of total output packets

This function is intended to be used by a SNMP agent. The counters in the structure are of volatile nature and may change at any time. The information should only be used for statistics or informational purposes.

S
A

netctl(SETUDPCHECK) - Enable/disable UDP checksum

Syntax:

```
#include <net/netctl.h>
int netctl(SETUDPCHECK, int *pBool, sizeof(int));
```

Description:

Enables or disables generation of UDP checksums in outgoing datagrams.

Parameters:

Option	SETUDPCHECK
pBool	Pointer to int containing either TRUE to enable UDP checksums or FALSE to disable UDP checksums.
Size	Must be sizeof(int).

Return values:

OK	Value successfully set
FAIL	Value not set. The error cause can be read from errno, errno_loc and errno_id.

See also:

netctl(GETUDPCHECK)

Remarks:

UDP checksums are enabled by default. The checksum of incoming UDP datagrams is always checked if present.

netctl(GETUDPCHECK) - Query UDP checksum

Syntax:

```
#include <net/netctl.h>
int netctl(GETUDPCHECK, int *pBool, sizeof(int));
```

Description:

Queries state of UDP checksum generation.

Parameters:

Option	GETUDPCHECK
pBool	Pointer to int receiving either TRUE when UDP checksums are enabled or FALSE when UDP checksums are disabled.
Size	Must be sizeof(int).

Return values:

OK	Value successfully queried
FAIL	Value not queried. The error cause can be read from errno, errno_loc and errno_id.

See also:

netctl(SETUDPCHECK)

Remarks:

UDP checksums are enabled by default.

S
A

netctl(GETUDPCONNS) - Get UDP connections

Syntax:

```
#include <net/netctl.h>
#include <net/udp_var.h>
int netctl(GETUDPCONNS, struct udpconnlist *pBuffer, int Size);
```

Description:

Queries list of UDP connections.

Parameters:

Option	GETUDPCONNS
pBuffer	Pointer to struct udpconnlist receiving connection information.
Size	Must be either sizeof(int) or size of buffer space.

Return values:

OK	Values successfully received
FAIL	Values not received. The error cause can be read from errno, errno_loc and errno_id.

See also:

-

Remarks:

When Size is sizeof(int), the call returns the number of entries in the list. When Size > sizeof(int), connection entries are returned, up to the buffer size.

The structure struct udpconnlist is defined in net/udp_var.h and contains the following elements:

Number	Number of elements in the array connections.
connections	Array of structures containing connection information.

The structure struct udpconnection contains the following elements:

LocalAddr	Local IP address of the connection, network byte order
ForAddr	Foreign IP address of the connection, network byte order
LocalPort	Local port number of the connection, network byte order
ForPort	Foreign port number of the connection, network byte order

This function is intended to be used by a SNMP agent.

`netctl(SETICMPMASKR)` - Enable/disable ICMP Mask Reply

Syntax:

```
#include <net/netctl.h>
int netctl(SETICMPMASKR, int *pBool, sizeof(int));
```

Description:

Enables or disables answers to ICMP Mask Requests.

Parameters:

`Option`	`SETICMPMASKR`
`pBool`	Pointer to `int` containing either `TRUE` to enable replies or `FALSE` to disable replies.
`Size`	Must be `sizeof(int)`.

Return values:

`OK`	Value successfully set
`FAIL`	Value not set. The error cause can be read from `errno`, `errno_loc` and `errno_id`.

See also:

`netctl(GETICMPMASKR)`

Remarks:

When this option is enabled replies to ICMP Mask Requests are sent. Otherwise these requests are ignored. This option is disabled by default.

S
A

netctl(GETICMPMASKR) - Query ICMP Mask Reply

Syntax:

```
#include <net/netctl.h>
int netctl(GETICMPMASKR, int *pBool, sizeof(int));
```

Description:

Queries the state of answers to ICMP Mask Requests.

Parameters:

Option	GETICMPMASKR
pBool	Pointer to int receiving either TRUE when replies are enabled or FALSE when replies are disabled.
Size	Must be sizeof(int).

Return values:

OK	Value successfully queried
FAIL	Value not queried. The error cause can be read from errno, errno_loc and errno_id.

See also:

netctl(SETICMPMASKR)

Remarks:

This option is disabled by default.

netctl(GETICMPSTATS) - Get ICMP statistics

Syntax:

```
#include <net/netctl.h>
#include <net/icmp_var.h>
int netctl(GETICMPSTATS, struct icmpstat *pBuffer,
           sizeof(struct icmpstat));
```

Description:

Gets ICMP statistics.

Parameters:

Option	GETICMPSTATS
pBuffer	Pointer to struct icmpstat receiving ICMP statistics.
Size	Must be sizeof(struct icmpstat).

Return values:

OK	Values successfully received
FAIL	Values not received. The error cause can be read from errno, errno_loc and errno_id.

See also:

-

Remarks:

The structure struct icmpstat is defined in net/icmp_var.h and contains the following elements:

icps_error	Number of calls to icmp_error
icps_oldshort	Number of no error sent because original IP datagram was too short
icps_oldicmp	Number of no error sent because original IP datagram was ICMP message
icps_outhist	Array forming histogram of sent ICMP messages
icps_badcode	Number of ICMP messages with ICMP code out of range
icps_tooshort	Number of ICMP messages with length < ICMP_MINLEN
icps_checksum	Number of ICMP messages with bad checksum
icps_badlen	Number of calculated bound mismatch
icps_reflect	Number of responses sent
icps_inhist	Array forming histogram of received ICMP messages

This function is intended to be used by a SNMP agent. The counters in the structure are of volatile nature and may change at any time. The information should only be used for statistics or informational purposes.

S
A

netctl(SETARPENTRY) - Create entry in ARP cache

Syntax:

```
#include <net/netctl.h>
#include <net/if_arp.h>
int netctl(SETARPENTRY, struct arpreq *pBuffer, sizeof(struct arpreq));
```

Description:

Manually create entry in ARP cache.

Parameters:

Option	SETARPENTRY
pBuffer	Pointer to struct arpreq containing ARP entry.
Size	Must be sizeof(struct arpreq).

Return values:

OK	Value successfully set
FAIL	Value not set. The error cause can be read from errno, errno_loc and errno_id.

See also:

-

Remarks:

The ARP cache is usually maintained automatically. This function allows to create an entry in the ARP cache manually, eg. when the target system can not respond to ARP requests.

The structure struct arpreq is defined in net/if_arp.h and contains the following elements:

arp_pa	Protocol address. The address family must be AF_INET.	
arp_ha	Hardware address. The address family must be AF_UNSPEC.	
arp_flags	Flags for this entry. The following flags are defined:	
	ATF_INUSE	The entry is in use.
	ATF_COM	The entry is complete. This flag may be cleared if an ARP request was sent but no reply was received yet.
	ATF_PERM	The entry is permanent. Non-permanent entries are flushed periodically.
	ATF_PUBL	The entry is public. This host may answer with this entry on behalf of other hosts.
	ATF_USETRAILERS	obsolete flag, do not use.
	ATF_DOWN	The host corresponding to this entry is not responding. Outgoing datagrams addressed to this host are discarded.

When adding entries manually ATF_COM is enforced.

netctl(GETARPENTRY) - Get entry from ARP cache

Syntax:

```
#include <net/netctl.h>
#include <net/if_arp.h>
int netctl(GETARPENTRY, struct arpreq *pBuffer, sizeof(struct arpreq));
```

Description:

Get entry from ARP cache.

Parameters:

Option	GETARPENTRY
pBuffer	Pointer to struct arpreq containing ARP entry.
Size	Must be sizeof(struct arpreq).

Return values:

OK	Value successfully queried
FAIL	Value not queried. The error cause can be read from errno, errno_loc and errno_id.

See also:

netctl(SETARPENTRY)

Remarks:

The structure struct arpreq is defined in net/if_arp.h. See netctl(SETARPENTRY) for details.

The component arp_pa in struct arpreq must be set by the user. When there is an ARP entry for this host arp_ha and arp_flags are set by the network stack.

S
A

netctl(DELARPENTRY) - Remove entry from ARP cache

Syntax:

```
#include <net/netctl.h>
#include <net/if_arp.h>
int netctl(DELARPENTRY, struct arpreq *pBuffer, sizeof(struct arpreq));
```

Description:

Remove entry from ARP cache.

Parameters:

Option	DELARPENTRY
pBuffer	Pointer to struct arpreq containing ARP entry.
Size	Must be sizeof(struct arpreq).

Return values:

OK	Entry successfully removed
FAIL	Entry not removed. The error cause can be read from errno, errno_loc and errno_id.

See also:

netctl(SETARPENTRY)

Remarks:

The structure struct arpreq is defined in net/if_arp.h. See netctl(SETARPENTRY) for details.

The component arp_pa in struct arpreq must be set by the user. When there is an ARP entry for this host the entry is removed from the ARP cache.

`netctl(FLUSHARP)` - Flush ARP cache

Syntax:

```
#include <net/netctl.h>
int netctl(FLUSHARP, NULL, 0);
```

Description:

Remove all entries from the ARP cache.

Parameters:

`Option`	`FLUSHARP`
`pBuffer`	Ignored
`Size`	Ignored

Return values:

`OK`	ARP cache flushed
`FAIL`	ARP cache not flushed. The error cause can be read from `errno`, `errno_loc` and `errno_id`.

See also:

-

Remarks:

This call removes all entries from the ARP cache.

S
A

netctl(DUMPARPENTRIES) - Get all ARP entries

Syntax:

```
#include <net/netctl.h>
#include <net/if_arp.h>
int netctl(DUMPARPENTRIES, struct arpdump *pBuffer, int Size);
```

Description:

Queries list of all ARP cache entries.

Parameters:

Option	DUMPARPENTRIES
pBuffer	Pointer to struct arpdump receiving ARP entries.
Size	Must be either sizeof(int) or size of buffer space.

Return values:

OK	Values successfully received
FAIL	Values not received. The error cause can be read from errno, errno_loc and errno_id.

See also:

netctl(SETARPENTRY)

Remarks:

When Size is sizeof(int), the call returns the number of entries in the list. When Size > sizeof(int), ARP entries are returned, up to the buffer size.

The structure struct arpdump is defined in net/if_arp.h and contains the following elements:

NumEntries	Number of elements in the array entries.
entries	Array of structures containing ARP entries.

The structure struct arpreq is defined in net/if_arp.h. See netctl(SETARPENTRY) for details.

This function is intended to be used by a SNMP agent.

`netctl(SIOCSIFADDR)` - Set local interface address

Syntax:

```
#include <net/netctl.h>
#include <net/if.h>
int netctl(SIOCSIFADDR, struct ifreq *pBuffer, sizeof(struct ifreq));
```

Description:

Set local address of a network interface.

Parameters:

`Option`	`SIOCSIFADDR`
`pBuffer`	Pointer to `struct ifreq` specifying the interface and address.
`Size`	Must be `sizeof(struct ifreq)`.

Return values:

`OK`	Address successfully set
`FAIL`	Address not set. The error cause can be read from `errno`, `errno_loc` and `errno_id`.

See also:

`netctl(SIOCGIFADDR)`

Remarks:

The component `ifr_name` in `struct ifreq` must contain the name of the interface. The component `ifr_addr` must contain the new address.

Setting an interface address for the first time also sets the network mask and broadcast address. A direct network or host route is automatically added or changed. For broadcast networks a gratuitous ARP request ist sent.

S
A

netctl(SIOCGIFADDR) - Get local interface address

Syntax:

```
#include <net/netctl.h>
#include <net/if.h>
int netctl(SIOCGIFADDR, struct ifreq *pBuffer, sizeof(struct ifreq));
```

Description:

Get local address of a network interface.

Parameters:

Option	SIOCGIFADDR
pBuffer	Pointer to struct ifreq specifying the interface.
Size	Must be sizeof(struct ifreq).

Return values:

OK	Address successfully queried
FAIL	Address not queried. The error cause can be read from errno, errno_loc and errno_id.

See also:

netctl(SIOCSIFADDR)

Remarks:

The component ifr_name in struct ifreq must contain the name of the interface. The component ifr_addr receives the current local address.

netctl(SIOCSIFDSTADDR) - Set destination address of interface

Syntax:

```
#include <net/netctl.h>
#include <net/if.h>
int netctl(SIOCSIFDSTADDR, struct ifreq *pBuffer,
           sizeof(struct ifreq));
```

Description:

Set destination (peer) address of a point-to-point a network interface.

Parameters:

Option	SIOCSIFDSTADDR
pBuffer	Pointer to struct ifreq specifying the interface.
Size	Must be sizeof(struct ifreq).

Return values:

OK	Address successfully set
FAIL	Address not set. The error cause can be read from errno, errno_loc and errno_id.

See also:

netctl(SIOCGIFDSTADDR)

Remarks:

The component ifr_name in struct ifreq must contain the name of the interface. The component ifr_dstaddr must contain the new destination address.

A direct route associated with the interface is automatically updated.

S
A

netctl(SIOCGIFDSTADDR) - Get destination address of interface

Syntax:

```
#include <net/netctl.h>
#include <net/if.h>
int netctl(SIOCGIFDSTADDR, struct ifreq *pBuffer,
           sizeof(struct ifreq));
```

Description:

Get destination (peer) address of a point-to-point a network interface.

Parameters:

Option	SIOCGIFDSTADDR
pBuffer	Pointer to struct ifreq specifying the interface.
Size	Must be sizeof(struct ifreq).

Return values:

OK	Address successfully queried
FAIL	Address not queried. The error cause can be read from errno, errno_loc and errno_id.

See also:

netctl(SIOCSIFDSTADDR)

Remarks:

The component ifr_name in struct ifreq must contain the name of the interface. The component ifr_dstaddr receives the current destination address.

netctl(SIOCSIFFLAGS) - Set interface flags

Syntax:

```
#include <net/netctl.h>
#include <net/if.h>
int netctl(SIOCSIFFLAGS, struct ifreq *pBuffer,
           sizeof(struct ifreq));
```

Description:

Set flags of a network interface.

Parameters:

Option	SIOCSIFFLAGS
pBuffer	Pointer to struct ifreq specifying the interface.
Size	Must be sizeof(struct ifreq).

Return values:

OK	Flags successfully set
FAIL	Flags not set. The error cause can be read from errno, errno_loc and errno_id.

See also:

netctl(SIOCGIFFLAGS)

Remarks:

The component ifr_name in struct ifreq must contain the name of the interface. The component ifr_flags must contain the new interface flags.

The following interface flags are defined:

IFF_UP	The interface is activated, i.e. it can now send and receive data.
IFF_BROADCAST	Broadcast datagrams can be sent on this interface.
IFF_DEBUG	Debugging is enabled. How debugging affects the interface depends on the driver.
IFF_LOOPBACK	The interface is a loopback interface, i.e. sent datagrams are looped back as received datagrams.
IFF_POINTOPOINT	The interface is a point-to-point link, i.e. there is exactly one other host at the other end of the link that can be reached directly. Otherwise more than one host can be reached directly.
IFF_NOARP	The address resolution protocol can not be used with this interface.
IFF_SIMPLEX	The interface does not receive its own transmissions.
IFF_MULTICAST	The interface supports multicast addressing.

Not all interface flags can be changes. Also see chapter 1.

S
A

netctl(SIOCGIFFLAGS) - Get interface flags

Syntax:

```
#include <net/netctl.h>
#include <net/if.h>
int netctl(SIOCGIFFLAGS, struct ifreq *pBuffer,
           sizeof(struct ifreq));
```

Description:

Get flags of a network interface.

Parameters:

Option	SIOCGIFFLAGS
pBuffer	Pointer to struct ifreq specifying the interface.
Size	Must be sizeof(struct ifreq).

Return values:

OK	Flags successfully queried
FAIL	Flags not queried. The error cause can be read from errno, errno_loc and errno_id.

See also:

netctl(SIOCSIFFLAGS)

Remarks:

The component ifr_name in struct ifreq must contain the name of the interface. The component ifr_flags receives the current interface flags.

`netctl(SIOCSIFBRDADDR)` - Set interface broadcast address

Syntax:

```
#include <net/netctl.h>
#include <net/if.h>
int netctl(SIOCSIFBRDADDR, struct ifreq *pBuffer,
           sizeof(struct ifreq));
```

Description:

Set broadcast address of a network interface.

Parameters:

`Option`	`SIOCSIFBRDADDR`
`pBuffer`	Pointer to `struct ifreq` specifying the interface and address.
`Size`	Must be `sizeof(struct ifreq)`.

Return values:

`OK`	Address successfully set
`FAIL`	Address not set. The error cause can be read from `errno`, `errno_loc` and `errno_id`.

See also:

`netctl(SIOCGIFBRDADDR)`

Remarks:

The component `ifr_name` in `struct ifreq` must contain the name of the interface. The component `ifr_broadaddr` must contain the new broadcast address.

A direct route associated with the interface is automatically updated.

S
A

netctl(SIOCGIFBRDADDR) - Get interface broadcast address

Syntax:

```
#include <net/netctl.h>
#include <net/if.h>
int netctl(SIOCGIFBRDADDR, struct ifreq *pBuffer,
           sizeof(struct ifreq));
```

Description:

Get broadcast address of a network interface.

Parameters:

Option	SIOCGIFBRDADDR
pBuffer	Pointer to struct ifreq specifying the interface.
Size	Must be sizeof(struct ifreq).

Return values:

OK	Address successfully queried
FAIL	Address not queried. The error cause can be read from errno, errno_loc and errno_id.

See also:

netctl(SIOCSIFBRDADDR)

Remarks:

The component ifr_name in struct ifreq must contain the name of the interface. The component ifr_broadaddr receives the current broadcast address.

netctl(SIOCSIFNETMASK) - Set interface network mask

Syntax:

```
#include <net/netctl.h>
#include <net/if.h>
int netctl(SIOCSIFNETMASK, struct ifreq *pBuffer,
           sizeof(struct ifreq));
```

Description:

Set network mask of a network interface.

Parameters:

Option	SIOCSIFNETMASK
pBuffer	Pointer to struct ifreq specifying the interface and address.
Size	Must be sizeof(struct ifreq).

Return values:

OK	Mask successfully set
FAIL	Mask not set. The error cause can be read from errno, errno_loc and errno_id.

See also:

netctl(SIOCGIFNETMASK)

Remarks:

The component ifr_name in struct ifreq must contain the name of the interface. The component ifr_addr must contain the new network mask.

A direct route associated with the interface is automatically updated.

S
A

netctl(SIOCGIFNETMASK) - Get interface network mask

Syntax:

```
#include <net/netctl.h>
#include <net/if.h>
int netctl(SIOCGIFNETMASK, struct ifreq *pBuffer,
           sizeof(struct ifreq));
```

Description:

Get network mask of a network interface.

Parameters:

Option	SIOCGIFNETMASK
pBuffer	Pointer to struct ifreq specifying the interface.
Size	Must be sizeof(struct ifreq).

Return values:

OK	Mask successfully queried
FAIL	Mask not queried. The error cause can be read from errno, errno_loc and errno_id.

See also:

netctl(SIOCSIFNETMASK)

Remarks:

The component ifr_name in struct ifreq must contain the name of the interface. The component ifr_addr receives the current network mask.

`netctl(SIOCGIFMETRIC)` - Get interface metric

Syntax:

```
#include <net/netctl.h>
#include <net/if.h>
int netctl(SIOCGIFMETRIC, struct ifreq *pBuffer,
           sizeof(struct ifreq));
```

Description:

Get metric of a network interface.

Parameters:

`Option`	`SIOCGIFMETRIC`
`pBuffer`	Pointer to `struct ifreq` specifying the interface.
`Size`	Must be `sizeof(struct ifreq)`.

Return values:

`OK`	Metric successfully queried
`FAIL`	Metric not queried. The error cause can be read from `errno`, `errno_loc` and `errno_id`.

See also:

-

Remarks:

The component `ifr_name` in `struct ifreq` must contain the name of the interface. The component `ifr_metric` receives the current interface metric.

This interface parameter is currently unused.

S
A

netctl(SIOCGIFLLADDR) - Get link-level address of interface

Syntax:

```
#include <net/netctl.h>
#include <net/if.h>
int netctl(SIOCGIFLLADDR, struct ifreq *pBuffer,
           sizeof(struct ifreq));
```

Description:

Get link-level (e.g. IEEE MAC) address of a network interface.

Parameters:

Option	SIOCGIFLLADDR
pBuffer	Pointer to struct ifreq specifying the interface.
Size	Must be sizeof(struct ifreq).

Return values:

OK	Address successfully queried
FAIL	Address not queried. The error cause can be read from errno, errno_loc and errno_id.

See also:

-

Remarks:

The component ifr_name in struct ifreq must contain the name of the interface. The component ifr_addr receives the current link-level address.

netctl(SIOCGIFSTAT) - Get link-level statistics of interface

Syntax:

```
#include <net/netctl.h>
#include <net/if.h>
int netctl(SIOCGIFSTAT, struct ifreq *pBuffer,
           sizeof(struct ifreq));
```

Description:

Get link-level statistics of a network interface.

Parameters:

Option	SIOCGIFSTAT
pBuffer	Pointer to struct ifreq specifying the interface.
Size	Must be sizeof(struct ifreq).

Return values:

OK	Statistics successfully queried
FAIL	Statistics not queried. The error cause can be read from errno, errno_loc and errno_id.

See also:

-

Remarks:

The component ifr_name in struct ifreq must contain the name of the interface. The component ifr_data must point to a structure of the type tIFSTATS. This structure has the following fields:

IPackets	Number of packets received on this interface
IErrors	Number of input errors on interface
OPackets	Number of packets sent on interface
OErrors	Number of output errors on interface
Collisions	Number of collisions on CSMA/CD interfaces
IBytes	Total number of octets received
OBytes	Total number of octets sent
IMcasts	Number of packets received via multicast
OMcasts	Number of packets sent via multicast
IqDrops	Number of packets dropped on input
NoProto	Number of input packets destined for unsupported protocols

The values of these fields are provided by the device driver associated with the interface. Not all drivers provide exact values in all fields. The driver's internal statistics are not reset when they are queried.

S
A

netctl(SIOCGIFCONF) - Get list of interfaces

Syntax:

```
#include <net/netctl.h>
#include <net/if.h>
int netctl(SIOCGIFCONF, struct ifconf *pBuffer,
           sizeof(struct ifconf));
```

Description:

Get list of network interfaces.

Parameters:

Option	SIOCGIFCONF
pBuffer	Pointer to struct ifconf receiving the list.
Size	Must be sizeof(struct ifconf).

Return values:

OK	List successfully queried
FAIL	List not queried. The error cause can be read from errno, errno_loc and errno_id.

See also:

-

Remarks:

On input the component ifc_len of struct ifconf contains the length of a data buffer. ifc_req must point to this data buffer. On output, the data buffer contains an array of struct ifreq with ifr_name and ifr_addr valid. ifc_len contains the size of the unused portion of the data buffer.

netctl(ATTACHINTERFACE) - Attach network interface to network stack

Syntax:

```
#include <net/netctl.h>
#include <net/if.h>
int netctl(ATTACHINTERFACE, struct ifattach *pBuffer,
           sizeof(struct ifattach));
```

Description:

Attach a driver to the network stack, making it a network interface.

Parameters:

Option	ATTACHINTERFACE
pBuffer	Pointer to struct ifattach.
Size	Must be sizeof(struct ifattach).

Return values:

OK	Interface successfully attached
FAIL	Interface not attached. The error cause can be read from errno, errno_loc and errno_id.

See also:

-

Remarks:

The structure struct ifattach is defined in net/if.h and contains the following elements:

UnitId	ID of the EUROS Port Driver channel or Resource Manager Unit
pName	Pointer to an interface name string.

The interface is made known to the network component and is associated with a name. The name is later used in the ifr_name field in struct ifreq.

Also see chapter 1.

S
A

netctl(ADDROUTE) - Add route to routing table

Syntax:

```
#include <net/netctl.h>
#include <net/route.h>
int netctl(ADDROUTE, struct rtreq *pBuffer,
           sizeof(struct rtreq));
```

Description:

Add a route to the host's routing table.

Parameters:

Option	ADDROUTE
pBuffer	Pointer to struct rtreq.
Size	Must be sizeof(struct rtreq).

Return values:

OK	Route successfully added
FAIL	Route not added. The error cause can be read from errno, errno_loc and errno_id.

See also:

netctl(DELROUTE)

Remarks:

The structure struct rtreq is defined in net/route.h and contains the following elements:

rt_dest	Pointer to destination address/net of route
rt_gateway	Pointer to address of router
rt_flags	Route flags

The destination address may be the default address (INADDR_ANY), in which case a default route is added. Otherwise it specifies either a network address (for network routes) or a host/interface address (host routes).

The router specified in the route must be reachable via a directly attached network.

The following route flags are defined:

RTF_UP	The route is valid and usable
RTF_GATEWAY	The route is a gateway (indirect) route
RTF_HOST	The route is a host route, ie. rt_dest specifies a host address. Otherwise the route is a network route, and rt_dest is a network address.
RTF_REJECT	The host or net is marked unreachable
RTF_DYNAMIC	The route was created dynamically (by ICMP redirect messages)
RTF_MODIFIED	The route was modified dynamically (by ICMP redirect messages)
RTF_STATIC	The route was manually added
RTF_BLACKHOLE	Pakets to this destination are discarded (during updates)

netctl(DELROUTE) - Remove route from routing table

Syntax:

```
#include <net/netctl.h>
#include <net/route.h>
int netctl(DELROUTE, struct rtreq *pBuffer,
           sizeof(struct rtreq));
```

Description:

Remove a route from the host's routing table.

Parameters:

Option	DELROUTE
pBuffer	Pointer to struct rtreq.
Size	Must be sizeof(struct rtreq).

Return values:

OK	Route successfully removed
FAIL	Route not removed. The error cause can be read from errno, errno_loc and errno_id.

See also:

netctl(ADDROUTE)

Remarks:

The structure struct rtreq is defined in net/route.h. See netctl(ADDROUTE) for details.

Both destination and gateway of the route must be given to specify the route to remove.

S
A

netctl(GETROUTES) - Get list of all routes

Syntax:

```
#include <net/netctl.h>
#include <net/route.h>
int netctl(GETROUTES, struct routelist *pBuffer, int Size);
```

Description:

Get list of all routing entries.

Parameters:

Option	GETROUTES
pBuffer	Pointer to struct routelist receiving route entries.
Size	Must be either sizeof(int) or size of buffer space.

Return values:

OK	Values successfully received
FAIL	Values not received. The error cause can be read from errno, errno_loc and errno_id.

See also:

-

Remarks:

When Size is sizeof(int), the call returns the number of entries in the list. When Size > sizeof(int), route entries are returned, up to the buffer size.

The structure struct routelist is defined in net/route.h and contains the following elements:

Number	Number of elements in the array Routes.
Routes	Array of structures containing route entries.

The structure struct routeentry is defined in net/route.h. It contains the following elements:

Dest	Destination address of route, network byte order
NextHop	Next hop (gateway) of route, network byte order
Type	Type of route (see MIB-2)
IfIndex	Index of interface associated with this route

This function is intended to be used by a SNMP agent.

netctl(GETIGMPSTATS) - Get IGMP statistics

Syntax:

```
#include <net/netctl.h>
#include <net/igmp_var.h>
int netctl(GETIGMPSTATS, struct igmpstat *pBuffer,
           sizeof(struct igmpstat));
```

Description:

Gets IGMP statistics.

Parameters:

Option	GETIGMPSTATS
pBuffer	Pointer to struct igmpstat receiving IGMP statistics.
Size	Must be sizeof(struct igmpstat).

Return values:

OK	Values successfully received
FAIL	Values not received. The error cause can be read from errno, errno_loc and errno_id.

See also:

-

Remarks:

The structure struct igmpstat is defined in net/igmp_var.h and contains the following elements:

igps_rcv_total	Total number of IGMP messages received
igps_rcv_tooshort	Number of messages received with too few bytes
igps_rcv_badsum	Number of messages received with bad checksum
igps_rcv_queries	Number of received membership queries
igps_rcv_badqueries	Number of received invalid queries
igps_rcv_reports	Number of received membership reports
igps_rcv_badreports	Number of received invalid reports
igps_rcv_ourreports	Number of received reports for our groups
igps_snd_reports	Number of sent membership reports

The counters in the structure are of volatile nature and may change at any time. The information should only be used for statistics or informational purposes.

S
A

gethostname - Get name of current host

Syntax:

```
#include <net/netctl.h>
int gethostname(char *name, u_int namelen);
```

Description:

Get name of host

Parameters:

name	pointer to buffer for name of host
namelen	length of buffer

Return values:

OK	Host name sucessfully returned
FAIL	Host name not returned. The error cause can be read from errno, errno_loc and errno_id.

See also:

sethostname, netctl

Remarks:

Gethostname returns the standard host name for the current processor, as previously set by sethostname. The parameter namelen specifies the size of the name array. The returned name is null-terminated unless insufficient space is provided. The call to gethostname is equivalent to a call to netctl and the option code GETHOSTNAME.

sethostname - Set name of current host

Syntax:

```
#include <net/netctl.h>
int sethostname(char *name, u_int namelen);
```

Description:

Set name of host.

Parameters:

name	pointer to buffer containing new name
namelen	length of buffer name

Return values:

OK	Host name sucessfully set
FAIL	Host name not set. The error cause can be read from errno, errno_loc and errno_id.

See also:

gethostname, netctl

Remarks:

Sethostname sets the name of the host machine to be name, which has length namelen. This call is normally used only when the system is bootstrapped. The call to sethostname is equivalent to a call to netctl and the option code SETHOSTNAME.

2.2 Main socket calls

The main socket calls are used to create and close sockets and to connect and disconnect them.

Function prototypes, macros and data structures are defined in the C header file `socket.h`.

S
A

socket - Create an endpoint for communication

Syntax:

```
#include <net/socket.h>
int socket(int domain, int type, int protocol);
```

Description:

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

Parameters:

`domain`	protocol family for which the socket will be used. Currently only `PF_INET` (ARPA Internet) is supported.
`type`	Type of socket to create. The following values are supported:
	`SOCK_STREAM` sequenced, reliable, two-way connection based byte streams. An out-of-band data transmission mechanism may be supported.
	`SOCK_DGRAM` datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length).
	`SOCK_RAW` access to internal network protocols and interfaces. The type `SOCK_RAW` is not described here.
`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; specify 0 to use the default protocol for the given protocol type (TCP for `SOCK_STREAM` and UDP for `SOCK_DGRAM`).

Return values:

`Descriptor`	Descriptor of created socket. This descriptor must be used when referencing the socket when calling other socket functions.
`FAIL`	Socket was not created.. The error cause can be read from `errno`, `errno_loc` and `errno_id`.

See also:

`accept`, `bind`, `connect`, `getsockname`, `getsockopt`, `soioctl`, `listen`, `recv`, `select`, `send`, `shutdown`

Remarks:

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 `send` and `recv` calls. When a session has been completed a `soclose` 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 FAIL returns and with ETIME as the specific code in the global variable 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). A FAIL return value is returned if a task sends on a broken stream, and EPIPE is returned in errno.

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

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

Socket descriptors are *not* EUROS object IDs. They can't be used when another EUROS function requires an object ID to be passed.

S
A

soclose - Close a socket

Syntax:

```
#include <net/socket.h>
int soclose(int s);
```

Description:

Close a socket

Parameters:

s	Descriptor of socket to close

Return values:

OK	Socket successfully closed
FAIL	Can't close socket. The error cause can be read from errno, errno_loc and errno_id.

See also:

setsockopt

Remarks:

The soclose call closes a socket. If the socket is in a connected state, it is disconnected first. soclose may block depending on the SO_LINGER socket option (set with setsockopt).

connect - Initiate a connection on a socket

Syntax:

```
#include <net/socket.h>
int connect(int s, struct sockaddr *name, int namelen);
```

Description:

Connect a socket

Parameters:

s	Socket to connect
name	Pointer to destination socket address
namelen	Length of destination socket address

Return values:

OK	Socket successfully connected
FAIL	Can't connect socket. The error cause can be read from errno, errno_loc and errno_id.

See also:

accept, socket, getsockname

Remarks:

If s 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.

S
A

shutdown - shut down part of a full-duplex connection

Syntax:

```
#include <net/socket.h>
int shutdown(int s, int how);
```

Description:

The shutdown call causes all or part of a full-duplex connection on the socket associated with `s` to be shut down.

Parameters:

`s`	Socket to shut down	
`how`	May have one of the following values:	
	`0`	shut down receive direction
	`1`	shut down send direction
	`2`	shut down both send and receive direction

Return values:

`OK`	Socket successfully shut down
`FAIL`	Can't shut down socket. The error cause can be read from `errno`, `errno_loc` and `errno_id`.

See also:

`connect`, `socket`

bind - Bind a socket to an address

Syntax:

```
#include <net/socket.h>
int bind(int s, struct sockaddr *name, int namelen);
```

Description:

Bind a socket to a local address.

Parameters:

s	Socket to bind to an address
name	Pointer to a socket address
namelen	Length of the socket address

Return values:

OK	Address successfully bound to socket
FAIL	Can't bind address to socket. The error cause can be read from errno, errno_loc and errno_id.

See also:

connect, listen, socket, getsockname

Remarks:

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

S
A

listen - Listen for connections on a socket

Syntax:

```
#include <net/socket.h>
int listen(int s, int backlog);
```

Description:

Put a socket into listening state.

Parameters:

s	Socket
backlog	Maximum length of queue of incoming connection requests. This parameter is internally limited to 5.

Return values:

OK	Success
FAIL	Error. The error cause can be read from errno, errno_loc and errno_id.

See also:

accept, connect, socket

Remarks:

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.

The backlog parameter defines the approximate 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.

Note that the parameter backlog does not influence the maximum number of concurrent connections on this socket.

accept - Accept a connection on a socket

Syntax:

```
#include <net/socket.h>
int accept(int s, struct sockaddr *addr, int *addrlen);
```

Description:

Accept pending incoming connection.

Parameters:

s	Listening socket
addr	Pointer to socket address buffer. This buffer is used to return the address of the connecting peer (client).
addrlen	Length of socket address buffer. This 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.

Return values:

Socket	Socket descriptor of the accepted connection
FAIL	Error. The error cause can be read from errno, errno_loc and errno_id.

See also:

bind, connect, listen, socket

Remarks:

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 call extracts the first connection request on the queue of pending connections and creates a new socket with the same properties of s. 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 above. The accepted socket may not be used to accept more connections. The original socket s remains open.

This call is used with connection-based socket types, i.e. sockets of type SOCK_STREAM.

2.3 Data transfer

The data transfer functions are used to send and receive data over sockets.

Function prototypes, macros and data structures are defined in the C header file `socket.h`.

S
A

recv - Receive a message from a socket

Syntax:

```
#include <net/socket.h>
ssize_t recv(int s, void *buf, size_t len, int flags);
```

Description:

Receive data from a socket

Parameters:

s	Socket to receive data from	
buf	Pointer to receive buffer	
len	Size of receive buffer	
flags	Receive options. The following options are supported:	
	MSG_OOB	process out-of-band data
	MSG_PEEK	peek at incoming message
	MSG_WAITALL	wait for full request or error

Return values:

Number of bytes	Number of bytes received
0	Connection was closed while waiting for data
FAIL	Error. The error cause can be read from errno, errno_loc and errno_id.

See also:

soioctl, getsockopt, socket, recvfrom

Remarks:

Recv is used to receive messages from a connected socket.

The routine returns 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 soioctl) in which case the value FAIL is returned and the external variable errno set to EAGAIN. 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 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.

recvfrom - Receive datagram

Syntax:

```
#include <net/socket.h>
ssize_t recvfrom(int s, void *buf, size_t len, int flags,
                 struct sockaddr *from, int *fromlen);
```

Description:

Receive datagram from peer

Parameters:

s	Socket to receive data from	
buf	Pointer to receive buffer	
len	Size of receive buffer	
flags	Receive options. The following options are supported:	
	MSG_OOB	process out-of-band data
	MSG_PEEK	peek at incoming message
	MSG_WAITALL	wait for full request or error
from	Pointer to address of peer	
fromlen	Pointer to length of address	

Return values:

Number of bytes	Number of bytes received
0	Connection was closed while waiting for data
FAIL	Error. The error cause can be read from errno, errno_loc and errno_id.

See also:

soioctl, getsockopt, socket, recv

Remarks:

Recvfrom is used to receive messages from a socket, 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.

The routine returns 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 soioctl) in which case the value FAIL is returned and the external variable errno set to EAGAIN. 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 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 an error or disconnect occurs, or the next data to be received is of a different type than that returned.

send - Send a message from a socket

Syntax:

```
#include <net/socket.h>
ssize_t send(int s, const void *msg, size_t len, int flags);
```

Description:

Send data to a socket

Parameters:

s	Socket to send data to	
msg	Pointer to data	
len	Size of data	
flags	Send options. Valid flags are:	
	MSG_OOB	process out-of-band data
	MSG_DONTROUTE	bypass routing, use direct interface

Return values:

Number of bytes	Number of bytes successfully sent
FAIL	Error. The error cause can be read from errno, errno_loc and errno_id.

See also:

recv, getsockopt, socket,sendto

Remarks:

Send is used to transmit a message to another socket. Send may be used only when the socket is in a connected state.

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

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 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.

S
A

sendto - Send message

Syntax:

```
#include <net/socket.h>
ssize_t sendto(int s, const void *msg, size_t len, int flags,
                const struct sockaddr *to, int tolen);
```

Description:

Send message to destination

Parameters:

s	Socket to send data to	
msg	Pointer to data	
len	Size of data	
flags	Send options. Valid flags are:	
	MSG_OOB	process out-of-band data
	MSG_DONTROUTE	bypass routing, use direct interface
to	Pointer to address of destination	
tolen	Length of destination address	

Return values:

Number of bytes	Number of bytes successfully sent
FAIL	Error. The error cause can be read from errno, errno_loc and errno_id.

See also:

recv, getsockopt, socket,send

Remarks:

Sendto is used to transmit a message to another socket.

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 EMSG-SIZE 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 FAIL.

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 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.

`select` - Determine number of ready sockets

Syntax:

```
#include <net/select.h>
int select(int nfds, fd_set *readfds,
                     fd_set *writefds,
                     fd_set *exceptfds,
                     const struct timeval *timeout);
```

Description:

select checks to see if any sockets are ready for reading (**readfds**), writing (**writefds**), or have an exceptional condition pending (**exceptfds**). timeout specifies the maximum time for select to complete. nfds specifies the maximum number of sockets to check. If timeout is a null pointer, select blocks indefinitely. timeout points to a timeval structure. A timeout value of 0 causes select to return immediately. This behavior is useful in applications that periodically poll their sockets. The arrival of out-of-band data is the only possible exceptional condition. fd_set is a type defined in the <net/select.h> header file. fd_set defines a set of file descriptors on which select operates. Passing NULL for any of the three fd_set arguments specifies that select should not monitor that condition.

On return, each of the input sets is modified to indicate the subset of descriptions that are ready. These may be found by using the FD_ISSET macro.

Parameters:

`nfds`	Maximum number of sockets to check	
`readfds`	Array of type fd_set containing identifiers (IDs) of sockets to check for read availability	
`writefds`	Array of type fd_set containing identifiers (IDs) of sockets to check for write availability	
`exceptfds`	Array of type fd_set containing identifiers (IDs) of sockets to check for exceptions	
`timeout`	How long should select wait for a network I/O or exception:	
	`NULL`	wait infinitely
	`0 sec 0 usec`	Poll / check only and do not block/wait
	`N sec K usec`	Wait at maximum for N seconds and K microseconds

Return values:

If select succeeds, it returns the number of ready socket descriptors. select returns a 0 if the time limit expires before any of the sockets is selected. In case of an error return value is FAIL.

The following macros manipulate sets of socket descriptors (specified in <net/select.h>):

FD_CLR(fd, &fdset)

removes the socket descriptor fd from the socket descriptor set fdset .

FD_ISSET(fd, &fdset)

returns nonzero if socket descriptor fd is a member of fdset . Otherwise, it returns a 0 .

FD_SET(fd, &fdset)
adds socket descriptor fd to fdset .

FD_ZERO(& fdset)
initializes fdset to 0 , representing the empty set.

FD_COPY(&fddest, &fdsrc)
Copy a source descriptor set.

See also:

`recv`, `getsockopt`, `socket`,`send`,`pselect`

Remarks:

It is possible to use a posix style select (requires EUROS version 3.20) where select will break when:

* socket is ready to be selected (according to read / write / except) specifications
* timeout interval has elapsed
* user specified signal has been send (see SignalSent)

In brief, a socket will be identified in a particular set when select returns if:

readfds:

* If listen has been called and a connection is pending, accept will succeed.
* Data is available for reading (includes OOB data if SO_OOBINLINE is enabled).
* Connection has been closed/reset/terminated.

writefds:

* If processing a connect call (nonblocking), connection has succeeded.
* Data can be sent.

exceptfds (not implemented yet):

* If processing a connect call (nonblocking), connection attempt failed.
* OOB data is available for reading (only if SO_OOBINLINE is disabled).

2.4 Byte order conversion

The byte order conversion functions are used to convert the byte order of 16 bit and 32 bit values from network byte order to host byte order and vice versa. When the host byte order is the same as the network byte order, some of these functions are implemented as empty macros.

Function prototypes, macros and data structures are defined in the C header file `socket.h`.

S
A

htonl - Convert byte order

Syntax:

```
#include <net/socket.h>
u_long htonl(u_long hostlong);
```

Description:

Convert 32 bit value from host byte order to network byte order

Parameters:

hostlong	32 bit value in host byte order

Return values:

Converted value

See also:

ntohl, lswap

Remarks:

On machines with a host byte order identical to the network byte order this routine is implemented as an empty macro.

`htons` - Convert byte order

Syntax:

```
#include <net/socket.h>
u_short htons(u_short hostshort);
```

Description:

Convert 16 bit value from host byte order to network byte order

Parameters:

`hostshort`	16 bit value in host byte order

Return values:

Converted value

See also:

`ntohs`, `bswap`

Remarks:

On machines with a host byte order identical to the network byte order this routine is implemented as an empty macro.

S
A

ntohl - Convert byte order

Syntax:

```
#include <net/socket.h>
u_long ntohl(u_long netlong);
```

Description:

Convert 32 bit value from network byte order to host byte order

Parameters:

netlong	32 bit value in network byte order

Return values:

Converted value

See also:

htonl, lswap

Remarks:

On machines with a host byte order identical to the network byte order this routine is implemented as an empty macro.

ntohs - Convert byte order

Syntax:

```
#include <net/socket.h>
u_short ntohs(u_short netshort);
```

Description:

Convert 16 bit value from network byte order to host byte order

Parameters:

`netshort`	16 bit value in network byte order

Return values:

Converted value

See also:

`htons`, `bswap`

Remarks:

On machines with a host byte order identical to the network byte order this routine is implemented as an empty macro.

S
A

bswap - Swap bytes of a 16 bit value

Syntax:

```
#include <net/socket.h>
u_short bswap(u_short x);
```

Description:

Swap bytes of a 16 bit value

Parameters:

x	16 bit value

Return values:

Value with bytes swapped

See also:

lswap

Remarks:

-

`lswap` - Swap bytes of a 32 bit value

Syntax:

```
#include <net/socket.h>
u_long lswap(u_long x);
```

Description:

Swap bytes of a 32 bit value

Parameters:

`x`	32 bit value

Return values:

32 bit value with swapped bytes

See also:

`bswap`

Remarks:

-

2.5 Socket utility functions

The socket utility functions are used to query addresses of sockets and to set and query socket options.

Function prototypes, macros and data structures are defined in the C header file `net/socket.h`.

2.5.1 Socket options

The following options can be set for an open socket using `setsockopt` and queried using `getsockopt`. Some options can be set only, some can be queried only.

The include file `net/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 `net/socket.h`, which specifies the desired state of the option and the linger interval (see below). `SO_SNDTIMEO` and `SO_RCVTIMEO` use a `uint32` parameter containing a timeout value (`TimeLimit` standard parameter).

Socket level options

The following options are recognized at the socket level, i.e. when `level` is `SOL_SOCKET`. Except as noted, each may be examined with `getsockopt` and set with `setsockopt`.

`SO_DEBUG` enables recording of debugging information;
`SO_DEBUG` enables debugging in the underlying protocol modules.

`SO_REUSEADDR` enables local address reuse;
`SO_REUSEADDR` indicates that the rules used in validating addresses supplied in a bind call should allow reuse of local addresses.

`SO_REUSEPORT` enables duplicate address and port bindings;
`SO_REUSEPORT` allows completely duplicate bindings by multiple processes if they all set `SO_REUSEPORT` before binding the port. This option permits multiple instances of a program to each receive UDP/IP multicast or broadcast datagrams destined for the bound port.

`SO_KEEPALIVE` enables keep connections alive;
`SO_KEEPALIVE` 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 receive errors with `errno` set to `EPIPE`.

`SO_DONTROUTE` enables routing bypass for outgoing messages;
`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` linger on close if data present;
`SO_LINGER` controls the action taken when unsent messages are queued on socket and a `soclose` is performed. If the socket promises reliable delivery of data and `SO_LINGER` is set, the system will block the process on the `soclose` 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 `soclose` is issued, the system will process the close in a manner that allows the process to continue as quickly as possible.

SO_BROADCAST	enables permission to transmit broadcast messages; The option SO_BROADCAST requests permission to send broadcast datagrams on the socket.
SO_OOBINLINE	enables reception of out-of-band data in band; 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	set buffer size for output; 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_RCVBUF	set buffer size for input; 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	set minimum count for output; 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	set minimum count for input; 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 the smaller of the amount available or the amount requested. The default value for SO_RCVLOWAT is 1. If SO_RCVLOWAT 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	set timeout value for output; SO_SNDTIMEO is an option to set a timeout value for output operations. It accepts a uint32 parameter with the TimeLimit for 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 ETIME 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. Please note that timeout values should be “or”-ed with appropriate EUROS constants. For example: 2 | SEC, instead of just 2.
SO_RCVTIMEO	set timeout value for input; SO_RCVTIMEO is an option to set a timeout value for input operations. It accepts a uint32 parameter with a TimeLimit value 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. Please note that timeout values should be "or"-ed with appropriate EUROS constants. For example: 2 | SEC, instead of just 2.

`SO_ERROR` get and clear error on the socket (get only);
Finally, `SO_ERROR` is an option used only with `getsockopt`. `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.

IP level options

The following options are recognized at IP level, i.e. when `level` is `IPPROTO_IP`. Except as noted, each may be examined with `getsockopt` and set with `setsockopt`.

`IP_OPTIONS` set IP options;
IP options are passed and returned in a `struct ip_opts` structure.

`IP_HDRINCL` application passes IP header in data for raw IP socket sends;
When the `IP_HDRINCL` option is active the application provides the IP header for outgoing datagrams on raw IP sockets at the beginning of the output data (`sendto` call), ie. it provides the entire datagram. When the option is inactive the system automatically prepends a default header.

`IP_TOS` set Type-Of-Service for outgoing datagrams;
set the Type-Of-Service byte for future outgoing datagrams.

`IP_TTL` set Time-To-Live for outgoing datagrams;
set the Time-To-Live byte for future outgoing datagrams.

`IP_RECVDSTADDR` receive destination address of UDP datagrams;
saves destination IP address of incoming UDP datagrams. The address is returned with `recvfrom`.

`IP_RETOPTS` see `IP_OPTIONS`

`IP_MULTICAST_IF` Select interface for outgoing multicast datagrams;
An interface is selected by specifying its (unicast) interface address in a `struct in_addr` structure. Specifying an address if `INADDR_ANY` deselects any previously selected interface.
If an interface is selected, all outgoing datagrams with a multicast destination address are sent on this interface. The source address of these datagrams is set to the (unicast) interface address of the interface. If no interface is selected, the output interface is determined by routing.

`IP_MULTICAST_TTL` Set Time-To-Live for outgoing multicast datagrams;
The TTL value is passed in an `uint8` variable.

`IP_MULTICAST_LOOP` Enable loopback of outgoing multicast datagrams;
Enables loopback of sent multicast datagrams, ie. sent multicast datagrams are also received if the host is a member of the destination multicast group. The parameter is passed in an `uint8` variable. When the parameter is `0`, loopback is disabled. When the parameter is `1`, loopback is enabled.

`IP_ADD_MEMBERSHIP` Add membership to a multicast group (set only);
The calling task is declaring itself as a member of a multicast group on a given interface. The group and interface are specified in a variable of the type

`struct ip_mreq`. The structure member `imr_multiaddr` contains the multicast group address (in network byte order), and `imr_interface` contains the (unicast) interface address of the interface (in network byte order). If the interface address is specified as `INADDR_ANY`, the interface is determined by routing.
Being a member of a multicast group means that datagrams adressed to this group can be received. Sending datagrams to a multicast group does not require membership of the group.

`IP_DROP_MEMBERSHIP` Drop membership to a multicast group (set only);
The calling task is retiring its membership of a multicast group on a given interface. The group and interface are specified in a variable of the type `struct ip_mreq`. The structure member `imr_multiaddr` contains the multicast group address (in network byte order), and `imr_interface` contains the (unicast) interface address of the interface (in network byte order). If the interface address is specified as `INADDR_ANY`, the membership to drop is determined by the multicast address.

TCP level options

The following options are recognized at TCP level, i.e. when `level` is `IPPROTO_TCP`. Except as noted, each may be examined with `getsockopt` and set with `setsockopt`.

`TCP_NODELAY` disable Nagle algorithm;
When the Nagle algorithm is enabled (default), small amounts of data to be sent are buffered until the acknowledgement for a previously sent small amount of data is received. This reduces load on network with long delays (WANs). If it's important that even these small segments are sent immediately `TCP_NODELAY` can be used to disable the Nagle algorithm.

`TCP_MAXSEG` set maximum segment size;
Set a new maximum size of outgoing TCP segments. The maximum segment size is first set when a TCP connection is established. With `TCP_MAXSEG` the segment size can be reduced.

`getpeername` - Get address of connected peer

Syntax:

```
#include <net/socket.h>
int getpeername(int s, struct sockaddr *name, int *namelen);
```

Description:

Return address of peer for a connected socket

Parameters:

`s`	Connected socket
`name`	Pointer to socket address buffer
`namelen`	Pointer to length of buffer

Return values:

`OK`	Address returned
`FAIL`	Error. The error cause can be read from `errno`, `errno_loc` and `errno_id`.

See also:

`accept`, `bind`, `socket`, `getsockname`

Remarks:

`Getpeername` returns the address 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 address returned (in bytes). The address is truncated if the buffer provided is too small.

S
A

getsockname - Get socket address

Syntax:

```
#include <net/socket.h>
int getsockname(int s, struct sockaddr *name, int *namelen);
```

Description:

Return address currently assigned to a socket

Parameters:

s	Socket
name	Pointer to socket address buffer
namelen	Pointer to buffer length

Return values:

OK	Address returned
FAIL	Error. The error cause can be read from errno, errno_loc and errno_id.

See also:

bind, socket

Remarks:

Getsockname returns the current address 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 address returned (in bytes).

getsockopt - Get options on sockets

Syntax:

```
#include <net/socket.h>
int getsockopt(int s, int level, int optname,
                void *optval, int *optlen);
```

Description:

Get socket option

Parameters:

s	Socket descriptor
level	Option level. May either be SOL_SOCKET or protocol number.
optname	Name of option
optval	Pointer to option value
optlen	Pointer to size of option value

Return values:

OK	Option successfully retrieved
FAIL	Error. The error cause can be read from errno, errno_loc and errno_id.

See also:

soioctl, socket, setsockopt

Remarks:

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 IPPROTO_TCP.

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.

See chapter 2.5.1 for an explanation of all socket options.

S
A

setsockopt - Set options on sockets

Syntax:

```
#include <net/socket.h>
int setsockopt(int s, int level, int optname,
               const void *optval, int optlen);
```

Description:

Set socket option

Parameters:

s	Socket descriptor
level	Option level. May either be SOL_SOCKET or protocol number.
optname	Name of option
optval	Pointer to option value
optlen	Size of option value

Return values:

OK	Option successfully set
FAIL	Error. The error cause can be read from errno, errno_loc and errno_id.

See also:

getsockopt

Remarks:

See chapter 2.5.1 for an explanation of all socket options.

soioctl - Set I/O mode for socket

Syntax:

```
#include <net/socket.h>
#include <net/sockio.h>
int soioctl(int s, int cmd, void *data);
```

Description:

The soioctl function sets or queries I/O modes for sockets.

Parameters:

s	Socket descriptor	
cmd	Ioctl command. Valid values for cmd are:	
	FIONBIO	Sets the socket into non-blocking or blocking I/O (default is blocking). data is assumed to point to an int containing 0 for blocking I/O or !=0 for non-blocking I/O. In non-blocking mode, when a socket function is called that would block in blocking mode, an error is reported and errno is set to EWOULDBLOCK.
	FIONREAD	Queries the number of bytes that are available for reading at the specified socket. data is assumed to point to an int. The number of data bytes are returned in this variable.
	SIOCATMARK	data is assumed to point to an int. A value !=0 is returned in this variable when out-of-band data is available at this socket. Otherwise 0 is returned.
data	Pointer to ioctl data (see above)	

Return values:

OK	Operation successful
FAIL	Error. The error cause can be read from errno, errno_loc and errno_id.

See also:

-

Remarks:

-

2.6 Internet address conversion

The internet address conversion functions are used to manipulate internet addresses and to convert them between textual representation and numeric representation.

Function prototypes, macros and data structures are defined in the C header file `socket.h`.

S
A

inet_addr - Convert text to Internet address

Syntax:

```
#include <net/socket.h>
u_long inet_addr(const char *cp);
```

Description:

Converts given textual representation of Internet address to numeric representation.

Parameters:

cp	Pointer to textual representation

Return values:

Numeric representation of address in network byte order. INADDR_NONE is returned for invalid input.

See also:

inet_aton

Remarks:

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.

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).

inet_aton - Convert text to Internet address

Syntax:

```
#include <net/socket.h>
int inet_aton(const char *cp, struct in_addr *addr);
```

Description:

Convert textual representation of internet address to numeric representation

Parameters:

cp	Pointer to textual representation
addr	Pointer to output buffer

Return values:

1	Address converted
0	Invalid input

See also:

inet_addr

Remarks:

The inet_aton routine interprets the specified character string as an Internet address, placing the address into the structure provided. It returns 1 if the string was successfully interpreted, or 0 if the string is invalid.

Also see the remarks for inet_addr.

S
A

inet_lnaof - Return local network address part

Syntax:

```
#include <net/socket.h>
u_long inet_lnaof(struct in_addr in);
```

Description:

Extract local network part from Internet address

Parameters:

in	Internet address

Return values:

Local network part of address

See also:

inet_netof

Remarks:

The routine inet_lnaof breaks apart an Internet host address, returning the local network address part.

`inet_netof` - Return network part of address

Syntax:

```
#include <net/socket.h>
u_long inet_netof(struct in_addr in);
```

Description:

Extract network part from Internet address

Parameters:

`in`	Internet address

Return values:

Network part of address

See also:

`inet_lnaof`

Remarks:

The routine `inet_netof` breaks apart an Internet host address, returning the network number address part.

S
A

inet_makeaddr - Construct Internet address

Syntax:

```
#include <net/socket.h>
struct in_addr inet_makeaddr(u_long net, u_long host);
```

Description:

Construct Internet address from network part and host part.

Parameters:

net	Network part of address
host	Host part of address

Return values:

Constructed Internet address

See also:

-

Remarks:

The routine inet_makeaddr takes an Internet network number and a local network address and constructs an Internet address from it.

inet_network - Convert text to network address

Syntax:

```
#include <net/socket.h>
u_long inet_network(const char *cp);
```

Description:

Convert textual representation of network address to numeric representation

Parameters:

cp	Textual representation of network address

Return values:

Numeric representation of network address in network byte order. INADDR_NONE is returned for invalid input.

See also:

inet_addr

Remarks:

See inet_addr for a description of valid input strings.

S
A

inet_ntoa - Convert Internet address to text

Syntax:

```
#include <net/socket.h>
char  *inet_ntoa(struct in_addr in);
```

Description:

The routine inet_ntoa takes an Internet address and returns an ASCII string representing the address.

Parameters:

in	Internet address

Return values:

Pointer to textual representation

See also:

inet_ntoa_r

Remarks:

The pointer returned by inet_ntoa points to task-local data buffer. Subsequent calls to inet_ntoa overwrite the contents of this buffer.

A re-entrant version of this call is provided, see inet_ntoa_r.

inet_ntoa_r - Convert Internet address to text

Syntax:

```
#include <net/socket.h>
char *inet_ntoa_r(struct in_addr in, char *pDest);
```

Description:

The routine inet_ntoa_r takes an Internet address and returns an ASCII string representing the address.

Parameters:

in	Internet address
pDest	Pointer to destination buffer

Return values:

Pointer to textual representation

See also:

inet_ntoa

Remarks:

This is a re-entrant version of the inet_ntoa call. The pointer returned by this call always points to the destination buffer passed as parameter.

2.7 Resolver functions

The resolver functions can be used to obtain host name information from a name server. These functions use the Domain Name System (DNS) to obtain this information. The resolver must be initialized before any query for information can be sent.

The function `gethostbyname` can be used to get the IP address for a given host name. The function `gethostbyaddr` can be used to get the host name for a given IP address. When these functions execute successfully the return a pointer to a `struct hostent` structure. The structure is local to the current task, i.e. multiple tasks can use `gethostbyname` and `gethostbyaddr` concurrently. However, when one task calls these functions, the information of the previous call is overwritten.

When `gethostbyname` and `gethostbyaddr` execute unsuccessfully `NULL` is returned and the global variable `h_errno` contains one of the following values:

`HOST_NOT_FOUND`	The specified host is unknown.
`TRY_AGAIN`	The information can't be obtained because of some temporary error (e.g. unreachable name server). The query may be repeated at a later time.
`NO_RECOVERY`	The information can't be obtained because of some permanent error. Repeating the query will not help.
`NO_DATA`	The specified host is known, but the requested information is not associated with the host.

The structure `struct hostent` consists of the following fields:

`h_name`	A pointer to the official name of the host.
`h_aliases`	An array of pointers to alternative names of the host. The last pointer in the array is `NULL`.
`h_addrtype`	The type of the address. This field always contains `AF_INET`.
`h_length`	The length of one address in bytes.
`h_addr_list`	An array of pointers to addresses of the host. The last pointer in the array is `NULL`.
`h_addr`	A pointer to the first address in `h_addr_list`.

Host names must always be fully qualified domain names (e.g. `host.domain.com` instead of `host`). Addresses must always be given in network byte order.

The other resolver functions can be used for general name server queries.

A

res_init - Initialize resolver

Syntax:

```
#include <net/resolv.h>
int res_init(const tResolverConfig *pConfig);
```

Description:

Initialize the resolver and configure name servers.

Parameters:

pConfig	Pointer to the structure containing configuration information. The structure tResolverConfig is explaned below.

Return values:

OK	The resolver was initialized successfully.
FAIL	The resolver is not initialized. The error cause can be read from errno, errno_loc and errno_id.

See also:

-

Remarks:

res_init must be called before any other resolver function can be used.

The structure tResolverConfig contains the following fields:

retrans	Retransmission time interval. This is a TimeLimit value (see Reference Manual, chapter 1). The interval should be at least 5 seconds.	
retry	Number of times a query is sent to each name server.	
nscount	Number of name servers in nsaddr_list. A maximum of 3 name servers is supported.	
options	Option flags. The following flags are defined:	
	RES_INIT	The resolver is initialized. Do not use this flag.
	RES_DEBUG	Print debugging information on the console (Debug version only)
	RES_USEVC	Always use virtual connections (i.e. TCP instead of UDP)
	RES_PRIMARY	Query primary server only
	RES_IGNTC	Ignore trucation errors
	RES_RECURSE	Recursion desired
	RES_STAYOPEN	Leave TCP connections open
	RES_DEFAULT	Combination of default flags
nsaddr_list	Array of name server addresses. Addresses and port numbers must be in network byte order.	

herror - Print text for current h_errno

Syntax:

```
#include <net/resolv.h>
void herror(const char *s);
```

Description:

Print error text for current h_errno value on console.

Parameters:

s	Pointer to additional text. When this pointer is NULL only the text for h_errno is printed.

Return values:

none

See also:

-

Remarks:

-

A

gethostbyname - Resolve host name

Syntax:

```
#include <net/resolv.h>
struct hostent *gethostbyname(const char *pName);
```

Description:

Query address information associated with the given host name.

Parameters:

pName	Pointer to host name. The host name must be a fully qualified domain name.

Return values:

pData	Pointer to host information.
NULL	No host information retrieved. h_errno contains more specific information.

See also:

gethostbyaddr

Remarks:

The structure pointed to by the return value is local to the current task. The next call to gethostbyname or gethostbyaddr by the same task will overwrite this information.

gethostbyaddr - Resolve host address

Syntax:

```
#include <net/resolv.h>
struct hostent *gethostbyaddr(const struct in_addr *pAddr, size_t Len,
                              int Type);
```

Description:

Query name information associated with the given address.

Parameters:

pAddr	Pointer to Internet address. The address must be given in network byte order.
Len	Length of the address in bytes.
Type	Type of address. Must always be AF_INET.

Return values:

pData	Pointer to host information.
NULL	No host information retrieved. h_errno contains more specific information.

See also:

gethostbyname

Remarks:

The structure pointed to by the return value is local to the current task. The next call to gethostbyname or gethostbyaddr by the same task will overwrite this information.

A

res_mkquery - Prepare query

Syntax:

```
#include <net/resolv.h>
ssize_t res_mkquery(int Op, const char *pName, uint16 Qclass,
                    uint16 Type, const char *pData, size_t Datalen,
                    u_char *pBuf, size_t Buflen);
```

Description:

Prepare a query to a name server. The query can be sent with res_send.

Parameters:

Op	Operation type. This parameter can be QUERY for standard queries or IQUERY for inverse queries.
pName	Pointer to the name to query.
Qclass	Query class. This Parameter can be any of the C_* macros defined in resolv.h.
Type	Query type. This Parameter can be any of the T_* macros defined in resolv.h.
pData	Pointer to additional data to be sent. May be NULL.
Datalen	Length of the additional data in bytes.
pBuf	Pointer to the buffer receiving the query.
Buflen	Length of the buffer in bytes.

Return values:

Size	Size of the resulting query in bytes.
FAIL	An error occured. The error cause can be read from errno, errno_loc and errno_id.

See also:

res_send

Remarks:

-

res_send - Send query

Syntax:

```
#include <net/resolv.h>
ssize_t res_send(const u_char *pBuf, size_t Buflen,
                 u_char *pAnswer, size_t Anslen);
```

Description:

Send query to name servers and receive reply. The query has been prepared with res_mkquery.

Parameters:

pBuf	Pointer to buffer containing the query.
Buflen	Length of the query in bytes.
pAnswer	Pointer to a buffer receiving the reply.
Anslen	Length of the buffer pointed to by pAnswer.

Return values:

Size	Length of the reply.
FAIL	An error occured. The error cause can be read from errno, errno_loc and errno_id.

See also:

res_mkquery

Remarks:

-

A

dn_comp - Compress domain name

Syntax:

```
#include <net/resolv.h>
int dn_comp(const u_char *exp_dn, u_char *comp_dn, int length,
            u_char **dnptrs, u_char **lastdnptr);
```

Description:

Compress domain name

Parameters:

exp_dn	Pointer to expanded domain name
comp_dn	Pointer to buffer for compressed domain name
length	Length of buffer
dnptrs	List of pointers to previous compressed names
lastdnptr	Pointer to the end of the arrary pointed to by dnptrs

Return values:

Size	Size of the compressed name
FAIL	An error occured. The error cause can be read from errno, errno_loc and errno_id.

See also:

dn_expand

Remarks:

Domain name compression is described in RFC-1035.

dn_expand - Expand compressed domain name

Syntax:

```
#include <net/resolv.h>
int dn_expand(const u_char *msg, const u_char *eomorig,
              const u_char *comp_dn, u_char *exp_dn, int length)
```

Description:

Expand compressed domain name

Parameters:

msg	Pointer to the beginning of the message
eomorig	Pointer to the first location after the message
comp_dn	Pointer to compressed domain name
exp_dn	Pointer to buffer for expanded domain name
length	Length of buffer

Return values:

Size	Size of the compressed name
FAIL	An error occured. The error cause can be read from errno, errno_loc and errno_id.

See also:

dn_comp

Remarks:

Domain name compression is described in RFC-1035.

2.8 BOOTP functions

BOOTP functions are used to obtain an IP address from a BOOTP server. The requesting system does not need to have a preconfigured IP address. It uses the MAC address of the network interface to identify itself to the BOOTP server. The server then responds to the BOOTP request and assigns an IP address to the client system.

To use BOOTP functions the network component must be initialized (`NetInit`) and at least one interface must be attached (`netctl`). The interface must be enabled.

A

BootRequest - Request IP address with BOOTP

Syntax:

```
#include <net/bootp.h>
int BootRequest(const char *pInterfaceName, int NumTries)
```

Description:

Request an IP address from a BOOTP server using the BOOTP protocol.

Parameters:

pInterfaceName	Name of the interface for which an IP address is to be obtained.
NumTries	Maximum number of tries to send a request and wait for a reply.

Return values:

OK	The address was successfully obtained.
FAIL	An error occured. The error cause can be read from errno, errno_loc and errno_id.

See also:

netctl

Remarks:

The interface name is the same as the one used to attach the interface to the network component. The interface must be able to send broadcasts and must have a valid media access address. The interface must be enabled.

NumTries is internally limited to 10.

The network mask of the interface is set according to the class of the obtained address.

2.9 `errno` values

When socket functions indicate an error, the external variable `errno` is set to a value describing the error. These error values are defined in the C header file `errno.h`. In addition to standard `errno` values (see the EUROSplus Reference Manual) the following network specific values may be returned:

`EADDRINUSE`	Address already in use. This error is generated when two sockets are to be bound to the same IP address/port number pair.
`EADDRNOTAVAIL`	Can't assign or find requested address
`EAFNOSUPPORT`	Address family not supported by protocol family. The EUROS Network Manager only supports the `AF_INET` address family.
`ECONNABORTED`	Software caused connection abort
`ECONNREFUSED`	Connection refused by peer. The peer may have no server running to accept connections for the given port number.
`ECONNRESET`	Connection was reset by peer. This usually happens when the peer detects a protocol error or when it terminates the connection.
`EDESTADDRREQ`	The operation requires a destination address but was not given.
`EHOSTDOWN`	Host is down. This error is generated when the destination host doesn't answer to ARP requests.
`EHOSTUNREACH`	No route to host
`EISCONN`	Socket is already connected. This happens when `connect` is called more than once for a stream socket.
`EMSGSIZE`	Message too long. The protocol can only handle shorter messages.
`ENETDOWN`	Network is down
`ENETUNREACH`	The network of the given destination address is unreachable, i.e. there is no route to the destination network.
`ENOBUFS`	No buffer space available
`ENOPROTOOPT`	Protocol not available
`ENOTCONN`	Socket is not connected while an established connection is required
`ENOTSOCK`	The descriptor passed to a socket function is not a socket descriptor.
`EOPNOTSUPP`	Operation not supported
`EPFNOSUPPORT`	Protocol family not supported. The EUROS Network Manager only supports the `PF_INET` protocol family.
`EPROTONOSUPPORT`	The given protocol is not supported
`EPROTOTYPE`	The given protocol type is not supported
`ETIMEDOUT`	Operation timed out
`ETOOMANYREFS`	The maximum number of memberships in multicast groups was reached

Index

O

P

R

S

T

U