euros®

Enhanced Universal Realtime Operating System

EUROSplus TCP/IP Networking

Programming Guide and Reference

Document version: 08/2000

Dr. Kaneff Engineering Consultants
Neutorgraben 17 | D-90419 Nuremberg | Germany
Phone: +49|911|33 84 33 | Fax: +49|911|33 86 06
www.kaneff.de | support@kaneff.de

EUROSplus TCP/IP Networking

Revision History

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

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* User’s Guide

The EUROS*plus* User’s Guide includes descriptions of the tools used in the development environment of EUROS*plus*.

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

1.1 Installation . . . 1-3
1.1.1 Files . . . 1-3
1.1.2 Debug version/No-Debug version . . . 1-3
1.1.3 Attaching network interfaces to the network component . . . 1-3
1.2 Socket programming . . . 1-5
1.2.1 Definitions . . . 1-5
1.2.2 Preparing tasks for network programming . . . 1-5
1.2.3 Basic data structures . . . 1-6
1.2.4 Typical TCP client application . . . 1-6
1.2.5 Typical TCP server application . . . 1-7
1.3 Technical information . . . 1-9
1.3.1 Supported features . . . 1-9
1.3.2 Changes compared to BSD sockets . . . 1-9

Chapter 2
Socket function reference

2.1 Initialization and configuration . . . 2-3
`NetInit` - Initialize network component . . . 2-4
`netctl` - Set parameters of network component . . . 2-5
`gethostname` - Get name of current host . . . 2-10
`sethostname` - Set name of current host . . . 2-11
2.2 Main socket calls . . . 2-13
`socket` - Create an endpoint for communication . . . 2-14
`soclose` - Close a socket . . . 2-16
`connect` - Initiate a connection on a socket . . . 2-17
`shutdown` - shut down part of a full-duplex connection . . . 2-18
`bind` - Bind a socket to an address . . . 2-19
`listen` - Listen for connections on a socket . . . 2-20
`accept` - Accept a connection on a socket . . . 2-21
2.3 Data transfer . . . 2-23
`recv` - Receive a message from a socket . . . 2-24
`recvfrom` - Receive datagram . . . 2-25
`send` - Send a message from a socket . . . 2-27
`sendto` - Send message . . . 2-28
2.4 Byte order conversion . . . 2-29
`htonl` - Convert byte order . . . 2-30
`htons` - Convert byte order . . . 2-31
`ntohl` - Convert byte order . . . 2-32
`ntohs` - Convert byte order . . . 2-33
`bswap` - Swap bytes of a 16 bit value . . . 2-34
`lswap` - Swap bytes of a 32 bit value . . . 2-35
2.5 Socket utility functions . . . 2-37
`getpeername` - Get address of connected peer . . . 2-38
`getsockname` - Get socket address . . . 2-39
`getsockopt` - Get options on sockets . . . 2-40
`setsockopt` - Set options on sockets . . . 2-44
`soioctl` - Set I/O mode for socket . . . 2-45
2.6 Internet address conversion . . . 2-47
`inet_addr` - Convert text to Internet address . . . 2-48
`inet_aton` - Convert text to Internet address . . . 2-49

inet_lnaof - Return local network address part 2-50
inet_netof - Return network part of address 2-51
inet_makeaddr - Construct Internet address 2-52
inet_network - Convert text to network address 2-53
inet_ntoa - Convert Internet address to text 2-54
inet_ntoa_r - Convert Internet address to text. 2-55
2.7 Resolver functions. 2-57
res_init - Initialize resolver 2-58
herror - Print text for current h_errno 2-59
gethostbyname - Resolve host name 2-60
gethostbyaddr - Resolve host address. 2-61
res_mkquery - Prepare query. 2-62
res_send - Send query 2-63
dn_comp - Compress domain name 2-64
dn_expand - Expand compressed domain name 2-65
2.8 BOOTP functions 2-67
BootRequest - Request IP address with BOOTP 2-68
2.9 errno values 2-69

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.

1.1.3 Attaching network interfaces to the network component

A network interface is a special EUROS driver (port driver channel or resource manager unit) that can be attached to the Network Manager. When the driver is attached to the Network Manager, it can no longer be accessed by the I/O System. When attaching the driver, the resulting interface is assigned a name. This name must be used to refer to that interface when using the `netctl` call. The name is internal to the Network Manager. It can *not* be used as input for the `ObjName` function. 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 Socket programming

1.2.1 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.

1.2.2 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 500 bytes of stack.

In addition, in order to be able to call some socket functions (especially `inet_aton`) the calling tasks must be created with the `TDP_USE_NET` flag set in their Task Definition Parameters.

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.

1.2.3 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.
`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`.

1.2.4 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);

...
```

1.2.5 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);
...
```

1.3 Technical information

1.3.1 Supported features

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
- configurable TTL
- configurable TOS
- fragmentation and reassembly

ICMP

ARP

Link layer protocols:

Point-to-point interfaces (PPP)

Broadcast interfaces (Ethernet, IEEE 802.2)

1.3.2 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 `protocols.h` and `socket.h` are provided containing symbolic definitions for protocols and services.
- IP multicasting is not supported
- Timeout values are specified with an EUROS standard TimeLimit value in an `uint32`.

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`.

`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()` call, one of these blocks is used.

`NumClusters` Number of available clusters (large buffers). These are used to buffer large amounts of protocol data.

`NumBuffers` Number of available buffers. These are used to buffer small amounts of protocol data.

`NumPcb` Number of available protocol control blocks. For each connection one of these blocks is required.

`NumUtilBlocks` Number of available utility blocks. These are used to store other information of the network component like routes, interface definitions, interface addresses etc.

Return values

`OK` Network component successfully initialized

`FAIL` Initialization failed

See also:

-

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 this data.

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

`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 below
`pData`	Pointer to option data
`Size`	Size of option data

Return values:

`OK`	Option successfully set
`FAIL`	Option not set

See also:

-

Remarks:

For every Option value, `pData` and `Size` have different meanings. The following table lists available options:

Option	Meaning	pData	Size
`SETHOSTNAME`	set host name	pointer to new host name	length of host name (including \0)
`GETHOSTNAME`	get host name	pointer to host name buffer	length of buffer
`DUMPSOCKETS`	print socket information on console	ignored	ignored
`SETDEFTTL`	set default TTL	pointer to `int` containing new TTL. The default value is 64.	`sizeof(int)`
`GETDEFTTL`	get default TTL	pointer to `int`	`sizeof(int)`
`SETFORWARDING`	set forwarding flag	pointer to int containing new flag. 0 means that no IP datagrams are forwarded. !=0 means that IP datagrams are forwarded if they are addressed to a different host. The default is to not forward datagrams.	`sizeof(int)`
`GETFORWARDING`	get forwarding flag	pointer to `int`	`sizeof(int)`

Option	Meaning	pData	Size
SETREDIR	set redirection flag	pointer to int containing new flag. 0 means that no ICMP redirect messages are sent. !=0 means that ICMP redirect messages are sent if necessary. The default is 0.	sizeof(int)
GETREDIR	get redirection flag	pointer to int	sizeof(int)
GETIPSTATS	get IP statistics	pointer to struct ipstat (see ip_var.h)	sizeof(struct ipstat)
GETROUTESTATS	get routing statistics	pointer to struct rtstat (see route.h)	sizeof(struct rtstat)
SETREASSTTL	set reassembly TTL. The default value is 60.	pointer to int containing new reassembly TTL	sizeof(int)
GETREASSTTL	get reassembly TTL	pointer to int	sizeof(int)
GETTCPSTATS	get TCP statistics	pointer to struct tcpstat (see tcp_var.h)	sizeof(struct tcpstat)
DUMPTCPPCBS	print TCP PCB information on console	ignored	ignored
GETTCPCONNS	get TCP connections	Pointer to buffer (see struct tcpconnlist in tcp_var.h)	When = sizeof(int), the call returns the number of entries in the list. When >sizeof(int), connection entries are returned, up to the buffer size.
GETUDPSTATS	get UDP statistics	pointer to struct udpstat (see udp_var.h)	sizeof(struct udpstat)
SETUDPCHECK	set UDP checksum flag.	pointer to int containing new flag. 0 means that the UDP checksum is not calculated for generated UDP datagrams. !=0 means that the checksum is calculated. The default is 1.	sizeof(int)
GETUDPCHECK	get UDP checksum flag	pointer to int	sizeof(int)

Option	Meaning	pData	Size
GETUDPCONNS	get UDP connections	Pointer to buffer (see struct udp-connlist in udp_var.h)	When = sizeof(int), the call returns the number of entries in the list. When >sizeof(int), connection entries are returned, up to the buffer size.
SETICMPMASKR	set ICMP address mask reply flag	pointer to int containing new flag. 0 means to not send replies to ICMP address mask requests. !=0 means to send replies. The default is 0.	sizeof(int)
GETICMPMASKR	get ICMP address mask reply flag	pointer to int	sizeof(int)
GETICMPSTATS	get ICMP statistics	pointer to struct icmpstat (see icmp_var.h)	sizeof(struct icmpstat)
SETARPENTRY	set ARP entry	pointer to struct arpreq (see if_arp.h)	sizeof(struct arpreq)
GETARPENTRY	get ARP entry	pointer to struct arpreq (see if_arp.h)	sizeof(struct arpreq)
DELARPENTRY	delete ARP entry	pointer to struct arpreq (see if_arp.h)	sizeof(struct arpreq)
FLUSHARP	Flush ARP cache	ignored	ignored
DUMPARPENTRIES	get all ARP entries	Pointer to buffer (see struct arpdump in if_arp.h)	When = sizeof(int), the call returns the number of entries in the cache. When >sizeof(int), cache entries are returned, up to the buffer size.
SIOCSIFADDR	set interface address	pointer to struct ifreq (see if.h). The component ifr_name contains the name of the interface. The component ifr_addr must contain the new address.	sizeof(struct ifreq)
SIOCGIFADDR	get interface address	pointer to struct ifreq (see if.h). The component ifr_name contains the name of the interface. The address is returned in the component ifr_addr.	sizeof(struct ifreq)
SIOCSIFDSTADDR	set destination address of point-to-point interface	pointer to struct ifreq (see if.h). The component ifr_name contains the name of the interface. The component ifr_dstaddr must contain the new address.	sizeof(struct ifreq)

Option	Meaning	pData	Size
SIOCGIFDSTADDR	get destination address of point-to-point interface	pointer to struct ifreq (see if.h). The component ifr_name contains the name of the interface. The address is returned in the component ifr_dstaddr.	sizeof(struct ifreq)
SIOCSIFFLAGS	set interface flags	pointer to struct ifreq (see if.h). The component ifr_name contains the name of the interface. The component ifr_flags must contain the new flags. Some interface flags can not be changed.	sizeof(struct ifreq)
SIOCGIFFLAGS	get interface flags	pointer to struct ifreq (see if.h). The component ifr_name contains the name of the interface. The flags are returned in the component ifr_flags.	sizeof(struct ifreq)
SIOCGIFBRDADDR	get broadcast address	pointer to struct ifreq (see if.h). The component ifr_name contains the name of the interface. The address is returned in the component ifr_broadaddr.	sizeof(struct ifreq)
SIOCSIFBRDADDR	set broadcast address	pointer to struct ifreq (see if.h). The component ifr_name contains the name of the interface. The component ifr_broadaddr must contain the new address.	sizeof(struct ifreq)
SIOCGIFCONF	get interface list	pointer to struct ifconf (see if.h). On input the component ifc_len 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_addr valid. ifc_len contains the size of the unused portion of the data buffer.	sizeof(struct ifconf)
SIOCGIFNETMASK	get network mask of interface	pointer to struct ifreq (see if.h). The component ifr_name contains the name of the interface. The network mask is returned in the component ifr_addr.	sizeof(struct ifreq)
SIOCSIFNETMASK	set network mask of interface	pointer to struct ifreq (see if.h). The component ifr_name contains the name of the interface. The component ifr_addr must contain the new network mask.	sizeof(struct ifreq)
SIOCGIFMETRIC	get interface metric	pointer to struct ifreq (see if.h). The component ifr_name contains the name of the interface. The metric is returned in the component ifr_metric.	sizeof(struct ifreq)
SIOCSIFMETRIC	set interface metric	pointer to struct ifreq (see if.h). The component ifr_name contains the name of the interface. The component ifr_metric must contain the new metric.	sizeof(struct ifreq)

Option	Meaning	pData	Size
`ATTACHINTERFACE`	Attach IP interface	pointer to `struct ifattach` (see `if.h`)	`sizeof(struct ifattach)`
`DETACHINTERFACE`	Detach IP interface		
`ADDROUTE`	add route	pointer to `struct rtreq` (see `route.h`)	`sizeof(struct rtreq)`
`DELROUTE`	delete route	pointer to `struct rtreq` (see `route.h`)	`sizeof(struct rtreq)`
`GETROUTES`	get all routes	Pointer to buffer (see `struct routelist` in `route.h`)	When = `sizeof(int)`, the call returns the number of entries in the list. When >`sizeof(int)`, route entries are returned, up to the buffer size.

`gethostname` - Get name of current host

Syntax:

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

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(const char *name, 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

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`.

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.

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

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

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.

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

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

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

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 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.

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

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`.

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

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

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

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.

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

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.

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.

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.

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 `socket.h`.

`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

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.

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

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

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. 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. 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.

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.

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`

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.

`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

See also:

`getsockopt`

Remarks:

See the remarks for `getsockopt` for a description of socket options.

soioctl - Set I/O mode for socket

Syntax:

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

Description:

Set I/O mode for socket Parameters:

Parameters:

`s`	Socket descriptor
`cmd`	Ioctl command
`data`	Pointer to ioctl data

Return values:

`OK`	Operation successful
`FAIL`	Error

See also:

-

Remarks:

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

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`.

`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

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.

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.

`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.

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. Tasks intending to call `inet_ntoa` *must* be created with the `TDP_USE_NET` flag set in their Task Definition Parameters. Otherwise `NULL` is returned for every call to `inet_ntoa`.

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.

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.

See also:

-

Remarks:

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

The structure tResolverConfig contains the following fields:

retrans — Retransmition 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.

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.

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.

See also:

`res_mkquery`

Remarks:

-

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.

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.

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 requests 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.

`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.

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 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.

EPROTONOSUPPORTThe given protocol is not supported

EPROTOTYPE The given protocol type is not supported

ETIMEDOUT Operation timed out

Index

A

accept 2-21
AF_INET 1-6, 2-69
ARP 1-3, 1-9, 2-7, 2-69

B

bind 2-19
BootRequest 2-68
bswap 2-34
buffer 2-4

C

Client 1-5
cluster 2-4
connect 2-14, 2-17, 2-69
Connection 1-5

D

datagram 2-14
descriptor 2-14
dn_comp 2-64
dn_expand 2-65
DNS 2-57

E

EAGAIN 2-24, 2-25
ECONNREFUSED 2-20
EMSGSIZE 2-28
EPIPE 2-15, 2-41
errno 2-15, 2-69
ETIME 2-14

F

fully qualified domain names 2-57

G

gethostbyaddr 2-57, 2-61
gethostbyname 2-57, 2-60
gethostname 2-10
getpeername 2-38
getsockname 2-39
getsockopt 2-15, 2-24, 2-40

H

h_errno 2-57
herror 2-59
host byte 2-30
Host byte order 1-5
HOST_NOT_FOUND 2-57
htonl 2-30
htons 2-31

I

ICMP 1-9, 2-7
INADDR_ANY 1-5
inet_addr 2-48
inet_aton 2-49
inet_lnaof 2-50
inet_makeaddr 2-52
inet_netof 2-51
inet_network 2-53
inet_ntoa 2-54
inet_ntoa_r 2-55
IP 1-9
IP level options 2-42

L

listen 2-20
lswap 2-35

M

main() 2-4
MSG_DONTROUTE 2-27, 2-28
MSG_OOB 2-24, 2-25, 2-27, 2-28
MSG_PEEK 2-24, 2-25
MSG_WAITALL 2-24, 2-25

N

netctl 1-3, 2-5, 2-10, 2-11
NetInit 2-4
Network byte order 1-5
network byte order 2-30
NO_DATA 2-57
NO_RECOVERY 2-57
ntohl 2-32
ntohs 2-33

O

out-of-band 2-14

P

Peer 1-5
PF_INET 2-14, 2-69

R

recv 2-14, 2-24
recvfrom 2-15, 2-25
res_init 2-58
res_mkquery 2-62
res_send 2-63
resolver 1-3

S

send 2-14, 2-27
sendto 2-15, 2-28
Server 1-5
sethostname 2-10, 2-11
setsockopt 2-15, 2-44

shutdown. 2-18
SO_LINGER 2-16, 2-40
SO_RCVLOWAT. 2-24, 2-25
SO_RCVTIMEO 2-24, 2-25, 2-40
SO_SNDTIMEO 2-40
SOCK_DGRAM. 2-14, 2-15, 2-17
SOCK_RAW 2-14, 2-15
SOCK_STREAM 2-14, 2-17, 2-21
socket 2-4, 2-14
Socket level options. .2-41
soclose 2-14, 2-16
soioctl. 1-3, 2-45
SOL_SOCKET. 2-40
stream. 2-14
struct hostent 2-57
struct sockaddr 1-6
struct sockaddr_in 1-6
system memory 2-4

T

TCP 1-6, 1-7, 1-9, 2-14
TCP level options 2-43
thread stack 1-5
tResolverConfig 2-58
TRY_AGAIN 2-57

U

UDP 1-9, 2-14