English

Fujitsu Software BS2000

openNet Server

SOCKETS

User Guide

Valid for:
SOCKETS V21.1A

Edition June 2026

Table of Contents

SOCKETS V21.1 User Guide.

Comments... Suggestions... Corrections...

The User Documentation Department would like to know your opinion on this manual. Your feedback helps us to optimize our documentation to suit your individual needs.

Feel free to send us your comments by e-mail to: bs2000.info@fujitsu.com

Certified documentation according to DIN EN ISO 9001:2015

To ensure a consistently high quality standard and user-friendliness, this documentation was created to meet the regulations of a quality management system which complies with the requirements of the standard DIN EN ISO 9001:2015.

Copyright and Trademarks

Preface

SOCKETS(BS2000) is the name for the socket functions within BS2000. These functions provide the development environment for BS2000 users who want to write socket application programs.

Brief product description

Sockets programming with SOCKETS(BS2000) provides a number of options for developing communication applications. SOCKETS(BS2000) is an interface for network programming within BS2000. It can be used to develop communication applications based on the TCP/IP protocols.

Objectives and target groups of this manual

This manual is intended for programmers who want to use the SOCKETS(BS2000) interface functions to develop communication applications.
Familiarity with the C programming language is required and assumed.

Summary of contents

This manual describes and illustrates the various options available for socket programming using simple examples. The example programs show how socket functions can be used for connection-oriented communication applications using the TCP protocol and the ISO service and for connectionless communication applications using the UDP protocol.

The manual is laid out as follows:

- Chapters 2 to 5 provide an introduction to developing SOCKETS(BS2000) communication applications. Sample programs are used to illustrate basic topics such as address structures, connection setup, data transfer and client /server communications.
- Chapter 6 contains an alphabetically-ordered reference section with the user functions of the SOCKETS (BS2000) interface.
- Chapter 7 describes the additional functions of the Socket interface under BS2000 for the special mode using an external bourse.
- Chapter 8 contains information on the following topics:
  - SOCKETS(BS2000) subsystems
  - SOCKETS(BS2000) user programs
  - Production of a SOCKETS(BS2000) user program with the associated software requirements

Information under BS2000

The `/SHOW-INSTALLATION-PATH INSTALLATION-UNIT=<product>` command shows the user ID under which the product's files are stored.

Additional product information

Current information, version and hardware dependencies, and instructions for installing and using a product version are contained in the associated Release Notice. These Release Notices are available online at http://bs2manuals.ts.fujitsu.com.

Changes since the last edition of the manual

- The function *recvmsg()* now supports the option MSG_ERRQUEUE.
- New section "Error analysis" has been added. The sections describes the new functionalities of MSG_ERRQUEUE.
- New section "CMSG-Macros" has been added. It explains how to access *cmsghdr* structures correctly.
- *getsockopt()* has been supplemented by the subfunctions SO_TIMESTAMP, IP_MTU_DISCOVER, IP_OPTIONS, IP_RECVTTL, IPV6_MTU_DISCOVER, IPV6_RECVHOPLIMIT and IPV6_UNICAST_HOPS.
- *setsockopt()* has been supplemented by the subfunctions SO_TIMESTAMP, IP_OPTIONS, IP_RECVTTL, IPV6_RECVHOPLIMIT and IPV6_UNICAST_HOPS.
- Setting the IP_TTL and IPV6_HOPLIMIT option in *setsockopt()* and *getsockopt()* is now also allowed for IPPROTO_IPV4/IPPROTO_IPV6. Datagram sockets are supported.
- Setting the IP_MTU_DISCOVER and IPV6_MTU_DISCOVER option in *setsockopt()* and *getsockopt()* is now also allowed for IPPROTO_IPV4/IPPROTO_IPV6. Datagram sockets are supported.
- The subfunctions IP_RECVERR and IPV6_RECVERR in *setsockopt()* have been linked to the new MSG_ERRQUEUE.
- The description about the nslookup was moved to the user guide "openNet Server BCAM Volume 1/2".

Additional changes since SOCKETS V21.0A01:

- Both calls *soc_poll()* and *select()* now run with an accuracy of up to +200ms (previously up to +10s).
- *soc_ioctl()* has been supplemented by the subfunctions SIOCGBCPROC und SIOCGCBFQDN.

Additional changes since SOCKETS V21.1A00:

- LWRESD will be replaced completely by the new DNS client GETDNS.
  - New options in the resolv file.
  - The DNS resolver can now be used with DNSSEC.
  - ASTI is no longer required for the DNS resolver.
- *getsockopt()* has been supplemented by the subfunction SO_RESOLVE_BCAM.
- *setsockopt()* has been supplemented by the subfunction SO_RESOLVE_BCAM.

Notational conventions

The following notational conventions are used in this manual:

i	For informative texts
—————— ——————	Syntax definitions are delimited above and below with horizontal lines. Continuation lines within syntax definitions are indented.
`typewritten font`	Program text in examples; syntax illustrations.
italic font	Names of programs, functions, function parameters, files, structures and structure components in descriptive text; syntax variables (e.g. *filename*)
<angle brackets>	Identify header files in descriptive text.
[ ]	Optional entries.The square brackets are metacharacters and must not be entered within statements.
...	Ellipses in syntax definitions mean that the preceding text may be repeated as often as required. In examples, they mean that the remaining parts are not relevant and are not required in order to understand the example.The ellipses are metacharacters and must not be entered within statements.

The notational conventions for describing user functions are explained at the start of the chapter "SOCKETS (BS2000) user functions".

References within this manual include the page concerned in the manual and the section or chapter as necessary. References to topics in other manuals include the brief title of the manual concerned. You will find the full title in the list of related publications at the end of this manual.

Compatibility of SOCKETS(BS2000) V21.1 with earlier versions

SOCKETS(BS2000) V21.1 is compatible to all older SOCKETS(BS2000) versions, i.e. existing Socket user programs can be executed in version 21.1.

If a Socket user program was produced with the user library SYSLIB.SOCKETS.211 then it will not possibly run on systems with subsystem version V21.0 or lower.

SOCKETS(BS2000) basics

This chapter explains the basic terms and functions of socket programming. Program examples for the topics handled in this chapter are summarized in the chapter "Client/server model with SOCKETS(BS2000)". The individual functions of the SOCKETS(BS2000) interface are described in detail in the chapter "SOCKETS(BS2000) user functions".

Network connection via the SOCKETS(BS2000) interface

The SOCKETS(BS2000) interface is one of the interfaces for network programming within BS2000. It can be used to develop communication applications based on the TCP/IP and OSI service definitions.

The SOCKETS(BS2000) interface is defined in separate header files.

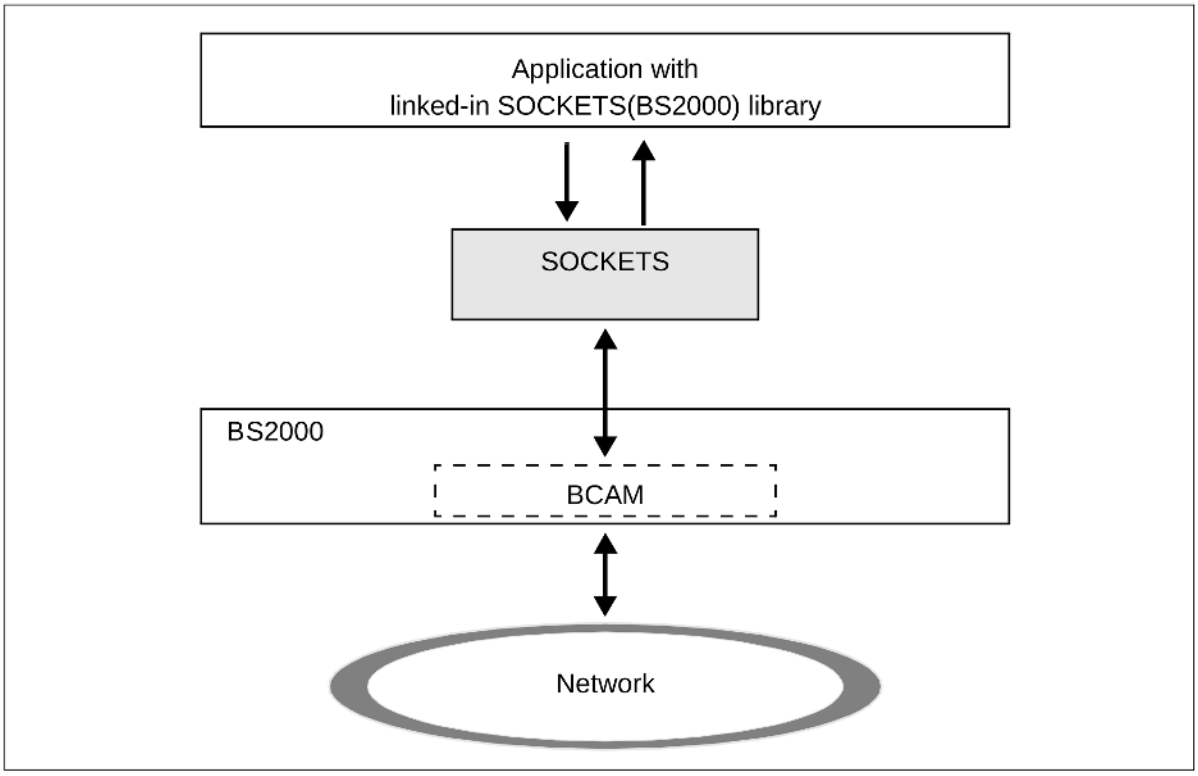


Figure 1: SOCKETS in BS2000

Header files

When SOCKETS(BS2000) is installed, BSD V4.2 and RFC 2553-compliant header files are created. The specific header file(s) to be linked by an application in order to execute a given function can be found under the description of each socket function in the chapter “SOCKETS(BS2000) user functions”.

SOCKETS(BS2000) provides the following header files:

arpa.inet.h

- Defines utility functions and macros for manipulating Internet addresses
- Defines the *in_addr* structure, as defined in <netinet.in.h>

icmp6.h

Structures and constants for the communication through a ICMPv6 protocol

ioctl.h

Defines the socket control functions called by *soc_ioctl()*.

ip.icmp.h

Structures and constants for the communication through a ICMPv4 protocol

iso.h

Defines the address structure for the AF_ISO address family

linux.errqueue.h

Defines the error structure for the error analysis (see chapter "Error analysis")

net.if.h

Structures for the packet switching interface

netdb.h

- Structures and function declarations for address conversion utilities
- Defines the flags for controlling the address conversion utilities
- Defines the error messages for the address conversion utilities

netinet.in.h

- Defines the address structure for the Internet domains (AF_INET, AF_INET6)
- Symbolic constants for protocol types
- Test macros for the AF_INET6 domain

sys.poll.h

Defines the *soc_poll()* function

sys.socket.h

- Defines the socket address structure and other structures for socket system functions
- Declares the socket system calls

- Symbolic constants for socket options and socket types

sys.time.h

timval structure for *select()* and subfunction *linger*

sys.uio.h

Data structure *msghdr* for the transfer of data in individual packets for *sendmsg()* and *recvmsg()*

Socket types

A socket is a basic component for developing communications applications and serves as a communications endpoint. A socket can be assigned a name via which it can then be accessed and addressed.

Each socket has a specific type and belongs to a task. More than one socket may be associated with the same task.

A socket belongs to a specific communications domain. Address and protocol families are collected together into a communications domain. An address family comprises addresses with the same address structure. A protocol family defines a set of protocols which implement the socket types in the domain. Communications domains are used to group the common characteristics of tasks that communicate via sockets. The socket interface in BS2000 supports the Internet communications domains AF_INET and AF_INET6, as well as the ISO communications domain AF_ISO.

There are various socket types with different communications characteristics. Two different socket types are currently supported:

- stream sockets
- datagram sockets
- raw sockets

Stream sockets (connection-oriented)

Stream sockets support connection-oriented communications. A Stream socket provides bidirectional, secured and sequential data flow, thus ensuring that the data is only transferred once and in the correct order.

Connection-oriented transfer in the communications domains AF_INET and AF_INET6

The record boundaries of the data are not retained for connection-oriented communication with stream sockets. Stream sockets are used to develop connection-oriented communications applications based on the TCP protocol.

Connection-oriented transfer in the communications domain AF_ISO

Connection-oriented communication in AF_ISO is record-oriented, i.e. the record boundaries remain intact. The communications applications are based on the ISO transport service.

Datagram sockets (connectionless)

Datagram sockets support connectionless communications in the AF_INET and AF_INET6 address families. A datagram socket provides bidirectional data flow. However, datagram sockets do not ensure either secure or sequential data transfer. It is also possible that the data may be transferred more than once. A task that receives messages on a datagram socket may therefore possibly receive messages more than once and/or in a different order from that transmitted. The application is therefore responsible for checking and saving the received data. One important characteristic of datagram sockets is that the record limits of the transferred data are retained.

Datagram sockets are used to develop connectionless communications applications based on the UDP protocol.

Raw sockets

Raw sockets offer the option of writing and reading data at protocol header level. SOCKETS(BS2000) permits this for the **I**nternet **C**ontrol **M**essage **P**rotocol (ICMP) and for the **I**nternet **C**ontrol **M**essage **P**rotocol for IP**v6** (ICMPv6). As a result, a sockets application with a raw socket is able to generate an ICMP/ICMPv6 echo request and to receive an ICMP/ICMPv6 echo reply.

Socket addressing

A socket is created initially without a name or address. After creating a socket, you will have to use the *bind()* function to assign the socket a name (address) according to its address family (see the section “Assigning a name to a socket”) so that tasks can address it. You can then receive messages via the socket.

Using socket addresses

When the *bind()*, *connect()*, *getpeername()*, *getsockname()*, *recvfrom()* and *sendto()* functions are called, a pointer to a name (address) is passed as the current parameter. Prior to this, the program has to make the name available according to the address structure of the address family used. This address structure will be different depending on the address family used.

If, at first, the address family has to be determined from the address structure in order to continue working specific to an address family, the general *sockaddr* structure is used.

The *sockaddr* structure is defined as follows in the header file <sys.socket.h> :

```
struct sockaddr {
  u_short  sa_family;   /* address family */
  char     sa_data[50]; /* 50 bytes for the longest address (sockaddr_iso) /*
  };
```

The address structures for the AF_INET and AF_INET6 address families, as well as the AF_ISO address family, are described in the following two sections.

Addressing with an Internet address

SOCKETS(BS2000) supports both IPv4 and IPv6 addresses. IPv4 and IPv6 addresses have different lengths and are therefore identified by different address families:

- AF_INET supports the 4-byte IPv4 Internet address.
- AF_INET6 supports the 16-byte IPv6 Internet address.

The structure of these addresses and the form they take are described in the “BCAM Volume 1/2” manual.

sockaddr_in address structure of the AF_INET address family

With the AF_INET address family, a name comprises a 4-byte Internet address and a port number. You use the *sockaddr_in* address structure for the AF_INET address family. This structure has a variant for #define SIN_LEN.

The *sockaddr_in* structure is declared in the <netinet.in.h> header as follows:

```
struct sockaddr_in {
   short            sin_family;      /* address family AF_INET */
   u_short          sin_port;        /* 16-bit port number */
   struct in_addr   sin_addr;        /* 32-bit Internet address */
   char             sin_zero[8];
   };
```

Structure variant of *sockaddr_in* with #define SIN_LEN set to support BSD 4.4 systems:

```
struct sockaddr_in {
   u_char           sin_len;         /* structure length */
   u_char           sin_family;      /* address family AF_INET */
   u_short          sin_port;        /* 16 bit port number */
   struct in_addr   sin_addr;        /* 32 bit Internet address */
   char             sin_zero[8];
   };
```

You can supply a variable *server* of type *struct sockaddr_in* with a name by using the following statements:

```
struct sockaddr_in server;
...
server.sin_family = AF_INET;
server.sin_port = htons(8888);
server.sin_addr.s_addr = htonl(INADDR_ANY);
```

A pointer to the variable *server* can now be passed as the current parameter, e.g. with a *bind()* call, to bind the name to a socket:

```
bind(..., &server, ...) /* bind() call with type conversion */
```

The structures for host, protocol and service names are described in the chapter “Address conversion with SOCKETS(BS2000)”.

sockaddr_in6 address structure of the AF_INET6 address family

With the AF_INET6 address family, a name comprises a 16-byte Internet address and a port number. You use the *sockaddr_in6* address structure for the AF_INET6 address family. This structure has additional variants for #define SCOPE_ID and #define SIN6_LEN.

The *sockaddr_in6* structure is declared in the <netinet.in.h> header as follows:

```
struct sockaddr_in6 {
   short            sin6_family;            /* address family AF_INET6 */
   u_short          sin6_port;              /* 16-bit port number */
   u_int            sin6_flowinfo
   struct in6_addr  sin6_addr;              /* IPv6 address */
   char             sin6_zero[8];
   };
```

Structure variant of *sockaddr_in6* with #define SCOPE_ID set to support Open Source:

```
struct sockaddr_in6 {
   short            sin6_family;            /* address family AF_INET6 */
   u_short          sin6_port;              /* 16 bit port number */
   u_int            sin6_flowinfo
   struct in6_addr  sin6_addr;              /* IPv6 address*/
   u_int32_t        sin6_scope_id;
   };
```

Structure variant of *sockaddr_in6* with #define SIN6_LEN set to support BSD 4.4 systems:

```
struct sockaddr_in6 {
   u_int8_t            sin6_len;            /* structure length */
   sa_family_t         sin6_family;         /* address family AF_INET6 */
   in_port_t           sin6_port;           /* 16 bit port number */
   u_int32_t           sin6_flowinfo
   struct in6_addr     sin6_addr;           /* IPv6 address */
   u_int32_t           sin6_scope_id;
   };
```

You can supply a variable *server* of type *struct sockaddr_in6* with a name by using the following statements:

```
struct sockaddr_in6 server;
struct in6_addr in6addr_any = IN6ADDR_ANY_INIT;
...
server.sin6_family = AF_INET6;
server.sin6_port = htons(8888);
   memcpy(server.sin6_addr.s6_addr, in6addr_any.s6_addr, 16);
```

A pointer to the variable *server* can now be passed as the current parameter, e.g. with a *bind()* call, to bind the name to a socket:

```
bind(..., &server, ...)   /* bind() call with type conversion */
```

Memory space allocation

Memory space allocation with associated initialization for the *in6addr_any* variable must take place in the code of the application. The following declaration is made available in the include file <netinet.in.h>:

```
extern const struct in6_addr in6addr_any;
```

in6addr_any has the value ::0. A corresponding constant IN6ADDR_ANY_INIT is defined in <netinet.in.h>.

sockaddr_iso address structure for the AF_ISO address family

With the AF_ISO address family, the name comprises a network selector NSEL and a transport selector TSEL. You use the *sockaddr_iso* address structure for the AF_ISO address family.

The *sockaddr_iso* structure is declared in the <iso.h> header as follows:

```
struct sockaddr_iso {
   u_char siso_len;            /* length of this sockaddr_iso structure*/
   u_char siso_family;         /* AF_ISO address family */
   u_char siso_plen;           /* length of presentation selector */
                               /* (is not supported; default: 0) */
   u_char siso_slen;           /* length of session selector */
                               /* (is not supported; default: 0) */
   u_char siso_tlen;           /* length of transport selector */
   struct iso_addr siso_addr;  /* ISO application address */
   u_char siso_pad[6];         /* is not supported*/
};

struct iso_addr {
   u_char isoa_len;            /* is not supported*/
   char isoa_genaddr[40];      /* complete address ( NSEL/TSEL ) */
};
```

The communications system for BS2000 expects a BCAM host name as the NSEL. The BCAM host name has a fixed length of 8 characters. Blanks are permitted at the end of the name, i.e. the name can be padded with blanks in order to achieve a length of 8 characters for NSEL. The transport selector TSEL can have a maximum length of 32 bytes. Because of the fixed length specifications for NSEL, you can use the transport selector length *siso_tlen* to select the transport selector from *isoa_genaddr*.

BCAM host name:

> The name is eight characters in length. Alphanumeric characters and the special characters #, @, $ or blanks can be used at the end of the name. As a rule, uppercase characters should be used, but the name is case-sensitive. Names comprising numeric characters only are not permitted.

Creating a socket

A socket is created with the *socket()* function:

```
int s;
...
s = socket(domain, type, protocol);
```

The *socket()* call creates a socket of type *type* in the domain *domain* and returns a descriptor (integer value). The new socket can be identified in all further socket function calls via this descriptor.

The domains are defined as fixed constants in the <sys.socket.h> header file. The following domains are supported:

- Internet communications domain AF_INET
- Internet communications domain AF_INET6
- ISO communications domain AF_ISO

You must therefore specify AF_INET, AF_INET6 or AF_ISO as the *domain*.

The socket types *type* are also defined in the <sys.socket.h> file:

- Specify SOCK_STREAM for *type,* if you want to set up connection-oriented communications via a stream socket.
- Specify SOCK_DGRAM for *type*, if you want to set up connectionless communications via a datagram socket.
- Specify SOCK_RAW for *type*, if you want to send an ICMP message via a raw socket.

The *protocol* parameter is not supported and should have the value 0.

Creating a socket in the AF_INET domain

The following call creates a stream socket in the AF_INET Internet domain:

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

In this case, the underlying communications support is provided by the TCP protocol.

The following call creates a datagram socket in the AF_INET Internet domain:

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

The UDP protocol used in this case transfers the datagrams without any further communications support to the underlying network services.

Creating a socket in the AF_INET6 domain

The following call creates a stream socket in the IPv6 Internet domain AF_INET6:

```
s = socket(AF_INET6, SOCK_STREAM, 0);
```

In this case, the underlying communications support is provided by the TCP protocol.

The following call creates a datagram socket in the IPv6 Internet domain AF_INET6:

```
s = socket(AF_INET6, SOCK_DGRAM, 0);
```

The UDP protocol used in this case transfers the datagrams without any further communications support to the underlying network services.

Creating a socket in the AF_ISO domain

The following call creates a socket in the ISO domain for using the ISO transport service:

```
s = socket(AF_ISO, SOCK_STREAM, 0);
```

Assigning a name to a socket

A socket created with *s=socket()* initially has no name. The socket must therefore be assigned a name, i.e. a local address. Not until this has been done can partners address the socket and socket users set up connections and send and/or receive data. You bind a name to the socket, i.e. you assign the socket a local address, with the *bind()* function.

Assigning an address explicitly

In this case, you call *bind()* as follows:

```
bind(s, name, namelen);
```

In the communications domain AF_INET, *name* comprises a 4-byte IPv4 address and a port number. *name* is passed in a variable of the type *struct sockaddr_in* (see section "sockaddr_in address structure of the AF_INET address family"). *namelen* contains the length of the data structure that defines the name.

In the communications domain AF_INET6, *name* comprises a 16-byte IPv6 address and a port number. *name* is passed in a variable of the type *struct sockaddr_in6* (see section "sockaddr_in6 address structure of the AF_INET6 address family"). *namelen* contains the length of the data structure that defines the name.

In the communications domain AF_ISO, *name* comprises a network selector and a transport selector. *name* is passed in a variable of the type *struct sockaddr_iso* (see section "sockaddr_iso address structure for the AF_ISO address family"). *namelen* contains the length of the data structure that defines the name.

Assigning an address explicitly in the domains AF_INET and AF_INET6

Assigning an address explicitly in AF_INET

The following program extract illustrates how a name is assigned to a socket in the AF_INET domain.

```
#include <sys.types.h>
#include <netinet.in.h>
 ...
struct sockaddr_in sin;
int s;
...
sin.sin_family = AF_INET;
sin.sin_port = 0;
sin.sin_addr.s_addr = INADDR_ANY;
 ...
bind(s, &sin, sizeof sin);
```

Assigning an address explicitly in AF_INET6

```
#include <sys.types.h>
#include <netinet.in.h>
 ...
struct sockaddr_in6 sin6;
struct in6_addr in6addr_any = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
int s;
...
sin6.sin6_family = AF_INET6;
sin6.sin6_port = 0;
memcpy(sin6.sin6_addr.s6_addr,in6addr_any.s6_addr,16);
 ...
bind(s, &sin6, sizeof sin6);
```

You must note the following when selecting the port number:

- Port numbers lower than PRIVPORT# (see the “BCAM Volume 1/2” manual) are reserved for privileged users (default: 2050).
- Certain port numbers are reserved for some standard applications:
  - Port number 3161 is used by the SNMP Basic Agent BS2000 is used for internal communications between the master agent and subagents (see the “SNMP Management for BS2000” manual).
  - Port number 1235 is required by the Domain Name Service (DNS) (see the “interNet Services” administration manual).
  - Note should be made of other well-known, registered, dynamic and/or private port numbers, which are documented on the IANA website at “http://www.iana.org/assignments/port-numbers”.

Assigning an address explicitly in the AF_ISO domain

The following program section illustrates how a name is assigned to a socket in the AF_ISO domain.

```
#include <sys.types.h>
#include <iso.h>
... ..
struct sockaddr_iso sin;
int s;
... ..
/* The statements which supply sin with a network selector
   and a transport selector must be inserted here.*/
... ..
bind(s, &sin, sizeof sin);
```

Assigning addresses with wildcards (AF_INET, AF_INET6)

Wildcard addresses simplify local address assignment in the Internet domains AF_INET and AF_INET6.

Assigning an Internet address with a wildcard

You use the *bind()* function to assign a local name (address) to a socket. Instead of a concrete Internet address, you can also specify INADDR_ANY (for AF_INET) or IN6ADDR_ANY (for AF_INET6) as the Internet address. INADDR_ANY and IN6ADDR_ANY are defined as a fixed constants in <netinet.in.h>.

When you use *bind()* to assign a socket *s* a name whose Internet address is specified as INADDR_ANY or IN6ADDR_ANY, this means:

- The socket *s* bound to INADDR_ANY can receive messages via all the IPv4 network interfaces of its host. This allows socket *s* to receive all messages addressed to the port number of *s* and any valid IPv4 address of the host on which socket *s* lies. For example, if the host has IPv4 addresses 128.32.0.4 and 10.0.0.78, a task to which socket *s* is assigned can accept connection requests which are addressed to 128.32.0.4 and 10.0.0.78.
- The socket *s* bound to IN6ADDR_ANY can receive messages via all the IPv4 and IPv6 network interfaces of its host. This allows socket *s* to receive all messages addressed to the port number of *s* and any valid IPv4 or IPV6 address of the host on which socket *s* lies. For example, if the host has IPv4 or IPv6 address 128.32.0.4 or 3FFE:1:1000:1000:52C1:D5FF:FE0E:2B01, a task to which socket *s* is assigned can accept connection requests which are addressed to 128.32.0.4 and 3FFE:1:1000:1000:52C1:D5FF:FE0E:2B01.

The following examples show how a task can bind a local name to a socket without an Internet address being specified. The task only has to specify the port number:

For AF_INET:

```
#include <sys.types.h>
#include <netinet.in.h>
#define MYPORT 2222
 ...
struct sockaddr_in sin;
int s;
 ...
s = socket(AF_INET, SOCK_STREAM, 0);
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, &sin, sizeof sin);
```

For AF_INET6:

```
#include <sys.types.h>
#include <netinet.in.h>
#define MYPORT 2222
 ...
struct in6_addr inaddr_any = IN6ADDR_ANY_INIT;
struct sockaddr_in6 sin6;
int s;
 ...
s = socket(AF_INET6, SOCK_STREAM, 0);
sin6.sin6_family = AF_INET6;
memcpy(sin6.sin6_addr.s6_addr, in6addr_any.s6_addr, 16);
sin6.sin6_port = htons(MYPORT);
bind(s, &sin6, sizeof sin6);
```

Assigning a port number with a wildcard

A local port can remain unspecified (0 specified). In this case, the system selects a suitable port number for it. The following examples show how a task assigns a socket a local address without specifying the local port number:

For AF_INET:

```
struct sockaddr_in sin;
...
s=socket(AF_INET, SOCK_STREAM,0);
sin.sin_family=AF_INET;
sin.sin_addr.s_addr=htonl(INADDR_ANY);
sin.sin_port = htons(0);
bind(s, &sin, sizeof sin);
```

For AF_INET6:

```
struct sockaddr_in6 sin6;
struct in6_addr in6addr_any = IN6ADDR_ANY_INIT;
 ...
s = socket(AF_INET6, SOCK_STREAM, 0);
sin6.sin6_family = AF_INET6;
memcpy(sin6.sin6_addr.s6_addr,in6addr_any.s6_addr, 16);
sin6.sin6_port = htons(0);
bind(s, &sin6, sizeof sin6);
```

Automatic address assignment by the system

You can still call a function for a socket which actually requires a bound socket (e.g. *connect()*, *sendto()*, etc.) even if the socket has no address assigned to it. In this case, the system executes an implicit *bind()* call with wildcards for the Internet address and port number, i.e. the socket is bound with INADDR_ANY to all IPv4 addresses and with IN6ADDR_ANY to all IPv6 addresses and IPv4 addresses of the host and receives a port number from a free range.

Direct address assignment in the domains AF_INET and AF_INET6

As of Version 2.2, it is possible to selectively bind a socket to a selected interface (multihoming support). When you do this, you should note that the required address must be present at the host in question and the IP address/port number tuple must not be occupied.

In this way, it is possible for a listen socket to listen in at a specific interface address and a specific port. In addition, it is possible for multiple listen sockets to be bound to one interface address each at a port

To make it possible to switch from single addressing to Anyaddr addressing and back again, the functionality of the *setsockopt()* subfunction *SO_REUSEADDR* has been extended. If the socket is marked using this subfunction before the *bind()* then, if the transport system permits it, it is possible to bind a socket to an interface address even though a socket has already been bound to Anyaddr for this port. This also applies in the opposite direction.

Communication in the AF_INET and AF_INET6 domains

A distinction is made between connection-oriented and connectionless communications in the AF_INET and AF_INET6 communications domains.

Connection-oriented communications in AF_INET and AF_INET6

Sockets which communicate with each other are connected via an assignment. An assignment in the Internet domain consists of a local address and local port number and a remote address and remote port number.

```
<local address, local port, foreign address, foreign port>
```

When setting up a socket, you must initially specify both address-pairs. The *bind()* call specifies the local half of the assignment:

```
<local address, local port>
```

The calls of the *connect()* and *accept()* functions described below, complete the socket assignment during connection setup.
The connection setup between two sockets is generally asymmetric, with one socket assuming the role of the client and the other the role of the server.

Connection request by the client

The client requests services from the server by sending a connection request to the socket of the server with the *connect()* function. On the client side, the *connect()* call causes a connection to be set up.

In the Internet domain AF_INET, a connection request progresses as follows:

```
struct sockaddr_in server;
 ...
connect(s, &server, sizeof server);
```

In the Internet domain AF_INET6, a connection request progresses as follows:

```
struct sockaddr_in6 server;
 ...
connect(s, &server, sizeof server);
```

The *server* parameter passes the Internet address and port number of the server with which the client wishes to communicate.
If the client's socket has no name assigned at the time of the *connect()* call, the system selects a name automatically and assigns it to the socket.

If connection setup is unsuccessful, an error code is returned. This can occur, e.g. if the server is not ready to accept a connection (see the section "Connection acceptance by the server"). However, all names assigned automatically by the system are retained even if the connection setup fails.

Connection acceptance by the server

If the server is ready to provide its special services, it assigns one of its sockets the name (address) defined for the service concerned. In order to be able to accept the connection request of a client, the server must also execute the following two steps:

1. The server uses the *listen()* function to mark the socket for incoming connection requests as “listening”. The server then monitors the socket, i.e. it waits passively for a connection request for this socket. It is now possible for any partner to take up contact with the server.

   listen() also causes SOCKETS(BS2000) to place connection requests to the socket concerned in a queue. This normally prevents any connection requests being lost while the server processes another one.
2. The server uses *accept()* to accept the connection for the socket marked as “listening”.

After the connection is accepted with *accept()*, the connection is set up between the client and server, and data can be transferred.

The following program extract illustrates connection acceptance by the server in the Internet domain AF_INET:

```
struct sockaddr_in from;
int s, fromlen, newsock;
...
listen(s, 5);
fromlen = sizeof(from);
newsock = accept(s, &from, &fromlen);
```

The following program extract illustrates connection acceptance by the server in the Internet domain AF_INET6:

```
struct sockaddr_in6 from;
int s, fromlen, newsock;
 ...
listen(s, 5);
fromlen = sizeof(from);
newsock = accept(s, &from, &fromlen);
```

The first parameter passed when *listen()* is called is the descriptor *s* of the socket on which the connection is to be set up. The second parameter defines the maximum number of connection requests which may be placed in the queue for acceptance by the server task.

Note, however, that SOCKET(BS2000) does not evaluate this parameter at present and continues to accept connection requests until the maximum number of available sockets have been used.

The first parameter passed when *accept()* is called is the descriptor *s* of the socket on which the connection is to be set up. After *accept()* is executed, the *from* parameter contains the address of the partner application, and *fromlen* contains the length of this address. When a connection is accepted with *accept()*, a descriptor is created for a new socket. This descriptor returns *accept()* as its result. Data can now be exchanged on the new socket. The server can accept additional connections on socket *s*.

An *accept()* call normally blocks because the *accept()* function does not return until a connection is accepted. When *accept()* is called, the server task also has no way of indicating that it only wants to accept connection requests from one or more specific partners. The server task must therefore note where the connection comes from and terminate it if it does not want to communicate with the client concerned.

The following points are described in detail in the chapter "Extended SOCKETS(BS2000) functions":

- how a server task can accept connections on more than one socket
- how a server task can prevent the *accept()* call from blocking

Data transfer with connection-oriented communications

Data can be transferred as soon as a connection is set up. If the communications endpoints of both communication partners are hard-bound with each other via the addressing-pair, a user task can send and receive messages without having to specify the addressing-pair every time.

There are several functions for sending and receiving data:

```
recv(s, buf, sizeof buf, flags);
send(s, buf, sizeof buf, flags);
soc_getc(c, s);
soc_gets(s, n, d);
soc_putc(c, s);
soc_puts(s, d);
soc_read(s, buf, sizeof buf);
soc_write(s, buf, sizeof buf);
recvmsg(s,msg,flags);
sendmsg(s,msg,flags);
```

The socket functions are described in detail in the section "Description of functions".

Examples of connection-oriented client/server communications

The following two program examples illustrate how a streams connection in the Internet domain is initialized by the client and accepted by the server:

Example 1: Initialization of a streams connection by the client

```
#include <sys.types.h>
#include <sys.socket.h>
#include <netinet.in.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#define DATA "Half a league, half a league . . ."

/*
 * This program creates a socket and initializes a connection with the
 * Internet address passed in the command line.
 * A message is sent via the connection.
 * The socket is then closed and the connection shut down.
 * The client program expects the entry of a host name and
 * port number. It is the host on which the server program runs and
 * the port number of the list socket of the server program (in the example
 * the port number 2222).
 */

main(argc, argv)
      int argc;
      char *argv[];
{
      int sock;
      struct sockaddr_in server;
      struct hostent *hp;
      /* Create a socket */
      sock = socket(AF_INET, SOCK_STREAM, 0);
      if (sock < 0) {
              perror("opening stream socket");
              exit(1);
      }

      /* Connection setup using the name specified in the
       * command line.
       */
      server.sin_family = AF_INET;
      hp = gethostbyname(argv[1]);
      if (hp == 0) {
              fprintf(stderr, "%s: unknown host\n", argv[1]);
              exit(2);
      }
      memcpy((char *)&server.sin_addr, (char *)hp->h_addr,
        hp->h_length);
      server.sin_port = htons(atoi(argv[2]));
      if (connect(sock, (struct sockaddr*)&server, sizeof server) < 0) {
              perror("connecting stream socket");
              exit(1);
      }
      if (send(sock, DATA, sizeof DATA, 0) < 0)
              perror("writing on stream socket");
      soc_close(sock);
      exit(0);
}
```

Example 2: Acceptance of the streams connection by the server

```
#include <sys.types.h>
#include <sys.socket.h>
#include <netinet.in.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#define TRUE 1
#define TESTPORT 2222
/*
 * This program creates a socket and then goes into an endless loop.
 * With each loop run, it accepts a connection and sends messages.
 * If the connection is interrupted or a termination message is passed,
 * the program accepts a new connection.
 */

main()
{
      int sock, length;
      struct sockaddr_in server, client;
      int msgsock;
      char buf[1024];
      int rval;
      /* Create socket. */
      sock = socket(AF_INET, SOCK_STREAM, 0);
      if (sock < 0) {
              perror("opening stream socket");
              exit(1);
      }
/* The socket is assigned a name using wildcards. */
      server.sin_family = AF_INET;
      server.sin_addr.s_addr = htonl(INADDR_ANY);
      server.sin_port = htons(TESTPORT);
      if (bind(sock, (struct sockaddr*)&server, sizeof server ) < 0) {
              perror("binding stream socket");
              exit(1);
      }
      /* Start acceptance of connection requests. */
      listen(sock, 5);
      do {
              length = sizeof client;
              msgsock = accept(sock, (struct sockaddr*)&client, &length);
              if (msgsock == -1)
                      perror("accept");
              else do {
                      memset(buf, 0, sizeof buf );
                      if ((rval = recv(msgsock, buf, 1024, 0)) < 0)
                              perror("reading stream message");
                 else if (rval == 0)
                              printf("Ending connection\n");
                      else
                              printf("-->%s\n", buf); }
              while (rval > 0);
              soc_close(msgsock);
      } while (TRUE);

      /*
       * As this program runs in an endless loop, the socket "sock" is
       * never explicitly closed.
```

```
        * However, all sockets are closed automatically if a task is
        * terminated or reaches its normal conclusion.
        */

        exit(0);
}
```

Connectionless communications in AF_INET and AF_INET6

In addition to the connection-oriented communications described in the previous section, connectionless communication via the UDP protocol is also supported in the AF_INET and AF_INET6 domains.

Connectionless communications are executed via datagram sockets (SOCK_DGRAM). In contrast to connection-oriented tasks, where the client and server communicate with each other via a fixed connection, no connection is set up for datagram transfers. Each message contains the destination address instead.

In the section "Creating a socket", you will find a description of how datagram sockets are created. If a specific local address is required, the *bind()* function must be called before the first data transfer (see section "Assigning a name to a socket"). Otherwise, the system assigns the local Internet address and/or port number the first time data is sent (see section "Assigning addresses with wildcards (AF_INET, AF_INET6)").

Data transfer with connectionless communications

You use the *sendto()* function to send data from one socket to another socket:

```
sendto(s, buf, buflen, flags, &to, tolen);
```

You use the *s*, *buf*, *buflen* and *flags* parameters in exactly the same way as with connectionoriented sockets. You pass the destination address with *to* and the length of the address with *tolen*.

Please note that reliable data transfer cannot be guaranteed when using a datagram interface. This means that a *sendto()* call can only return error information if the local system is aware of the fact that a message could not be transferred.

You use the *recvfrom()* function to receive a message on a datagram socket:

```
recvfrom(s, buf, buflen, flags, &from, &fromlen);
```

The *fromlen* parameter initially contains the size of the *from* buffer. On return from the *recvfrom()* function, *fromlen* specifies the size of the address of the socket from which the datagram was received.

If you wish, you can define a specific destination address for a datagram socket before a *sendto()* or *recvfrom()* call with *connect()*. In this case, calling *sendto()* or *recvfrom()* results in the following behavior:

- Data which the task sends with *sendto()* without explicitly specifying a destination address is sent automatically to the partner with the destination address specified in the *connect()* call.
- A user task only receives data with *recvfrom()* from the partner with the address specified in the *connect()* call.

For a datagram socket, only *one* target address can be specified with *connect()* at any one time. However, you can define a different destination address for the socket with an additional *connect()* call.

A *connect()* call for a datagram socket returns immediately, and the system only stores the address of the communications partner.

Examples of connectionless communications

The following two program examples illustrate how datagrams are received and sent with connectionless communications:

Example 1: Receiving datagrams

```
#include <sys.types.h>
#include <sys.socket.h>
#include <netinet.in.h>
#include <stdio.h>
#define TESTPORT 2222

/*
 * The <netinet.in.h> header file declares sockaddr_in as follows:
 *
 * struct sockaddr_in {
 *    short   sin_family;
 *    u_short sin_port;
 *    struct in_addr sin_addr;
 *    char    sin_zero[8];
 * };
 *
 * This program creates a socket, assigns it a name and then reads from
 * the socket.
 */

main()
{
      int sock, length, peerlen;
      struct sockaddr_in name, peer;
      char buf[1024];

      /* Create the socket to be read from. */
      sock = socket(AF_INET, SOCK_DGRAM, 0);
      if (sock < 0) {
              perror("opening datagram socket");
              exit(1);
      }

      /* Assign the socket a name using wildcards */
      name.sin_family = AF_INET;
      name.sin_addr.s_addr = INADDR_ANY;
      name.sin_port = htons(TESTPORT);

      if (bind(sock, &name, sizeof name ) < 0) {
              perror("binding datagram socket");
              exit(1);
      }

       /* Read from socket. */
       peerlen=sizeof peer;
       if (recvfrom(sock, buf, 1024, &peer, &peerlen) < 0)
               perror("receiving datagram packet");
       else
       printf("-->%s\n", buf);
       soc_close(sock);
       exit(0);
}
```

Example 2: Sending datagrams

```
#include <sys.types.h>
#include <sys.socket.h>
#include <netinet.in.h>
#include <netdb.h>
#include <stdio.h>
#define DATA "The sea is calm, the tide is full . . ."

/*
 * This program sends a datagram to a receiver whose name is passed via
 * the arguments in the command line.
 */

main(argc, argv)
      int argc;
      char *argv[];
{
      int sock;
      struct sockaddr_in name;
      struct hostent *hp;
      /* Create socket on which data is to be sent */
      sock = socket(AF_INET, SOCK_DGRAM, 0);
      if (sock < 0) {
              perror("opening datagram socket");
              exit(1);
      }

      /*
       * Construct the name of the socket on which data is to be sent
       * without using wildcards. gethostbyname()  returns a structure
       * containing the Internet address of the specified host. The
       * port number is taken over from the command line.
       */

      hp = gethostbyname(argv[1]);
      if (hp == 0) {
              fprintf(stderr, "%s: unknown host\n", argv[1]);
              exit(2);
      }
      memcpy( (char *)&name.sin_addr, (char *)hp->h_addr,
        hp->h_length);
      name.sin_family = AF_INET;
      name.sin_port = htons(atoi(argv[2]));

      /* Send message. */

      if (sendto(sock, DATA, sizeof DATA , 0, &name, sizeof name) < 0)
              perror("sending datagram message");
      soc_close(sock);
      exit(0);
}
```

Communications in the AF_ISO domain

Only connection-oriented communications are supported in the AF_ISO domain. Sockets which communicate with each other are connected via an assignment. An assignment in AF_ISO consists of a local network selector and a local transport selector, and a remote network selector and a remote transport selector:

```
<local nsel, local tsel, foreign nsel, foreign tsel>
```

When setting up a socket, you need not initially specify both address pairs.The *bind()* call specifies one half of the assignment:

```
<local nsel, local tsel>
```

The calls of the *connect()* and *accept()* functions described below complete the name assignment during connection setup.

The connection setup between two sockets is generally asymmetric, with one socket assuming the role of the client and the other the role of the server.

You will find examples of communications in the AF_ISO domain see section "Connection-oriented server for AF_ISO" (server example) and section "Connection-oriented client for AF_ISO" (client example).

Connection request by the client

The client requests services from the server by sending a connection request to the socket of the server with the *connect()* function. On the client side, the *connect()* call causes a connection to be set up. In the ISO domain (AF_ISO) a connection request progresses as follows:

```
struct sockaddr_iso name;
struct sockaddr_iso server;
...
bind(s, &name, sizeof name);
connect(s, &server, sizeof server);
```

The *server* parameter passes the network and transport selectors of the server with which the client wishes to communicate. The socket of the client must be assigned a name before *connect()* is called, i.e. *bind()* must have been called for the socket beforehand.

If connection setup is unsuccessful, an error code is returned. This can occur, for example, if the server is not ready to accept a connection (see the section “Connection acceptance by the server”). However, all names assigned by *bind()* are retained even if the connection setup fails.

Connection acceptance by the server

If the server is ready to provide its special services, it assigns one of its sockets the name (address) defined for the service concerned. In order to be able to accept the connection request of a client, the server must also execute the following two steps:

1. The server uses the *listen()* function to mark the socket for incoming connection requests as “listening”. The server then monitors the socket, i.e. it waits passively for a connection request for this socket. It is now possible for any partner to take up contact with the server. *listen()* also causes SOCKETS(BS2000) to place connection requests to the socket concerned in a queue. This normally prevents any connection requests being lost while the server processes another one.

   Exception: BCAM connection timer has elapsed:
   This timer must be taken into greater consideration when using the ISO transport service since here, unlike AF_INET and AF_INET6, the connection setup acknowledgment is not generated and sent to the partner until a send action is initiated (for example with *send()*).
2. The server uses *accept()* to accept the connection for the socket marked as “listening”. You can use the function *getsockopt()* or *recvmsg()* to evaluate the connection data transferred earlier for the connection request. Unlike the Internet domain, the connection is not completely set up after *cept()*. The connection to the partner is not set up in its entirety until
   - user data is sent or
   - CFRM data (confirm) is sent with the *sendmsg()* function.

The following program extract illustrates connection acceptance by the server in the AF_ISO domain:

```
struct sockaddr_iso from;
... ..
listen(s, 5);
fromlen = sizeof(from);
newsock = accept(s, &from, &fromlen);
send(newsock, msg, len, flags);
```

The first parameter passed when *listen()* is called is the descriptor *s* of the socket on which the connection is to be set up. The second parameter defines the maximum number of connection requests which may be placed in the queue for acceptance by the server task. Note, however, that SOCKET(BS2000) does not evaluate this parameter at present and continues to accept connection requests until the maximum number of available sockets have been used.

The first parameter passed when *accept()* is called is the descriptor *s* of the socket on which the connection is to be set up. After *accept()* is executed, the *from* parameter contains the address of the partner application, and *fromlen* contains the length of this address. When a connection is accepted with *accept()*, a descriptor is created for a new socket. This descriptor returns *accept()* as its result. Once the execution of *send()* has set up the connection completely, data can be exchanged on the new socket. The server can accept additional connections on socket *s*.

An *accept()* call normally blocks because the *accept()* function does not return until a connection is accepted. When *accept()* is called, the server task also has no way of indicating that it only wants to accept connection requests from one or more specific partners. The server task must therefore note where the connection comes from and terminate it if it does not want to communicate with the client concerned.

The following points are described in detail in the chapter "Extended SOCKETS(BS2000) functions":

- how a server task can accept connections on more than one socket
- how a server task can prevent the *accept()* call from blocking

Data transfer with connection-oriented communications

Data can be transferred as soon as a connection is set up. If the communications endpoints of both communication partners are hard-bound with each other via the addressing-pair, a user task can send and receive messages without having to specify the addressing-pair every time.

There are several functions for sending and receiving data:

```
recv(s, buf, sizeof buf, flags);
send(s, buf, sizeof buf, flags);
soc_read(s, buf, sizeof buf);
soc_write(s, buf, sizeof buf);
soc_readv(s, iov, iovcnt);
soc_writev(s, iov, iovcnt);
recvmsg(s, msg, flags);
sendmsg(s, msg, flags);
```

The socket functions are described in detail in the section “Description of functions”.

Terminating a connection and closing a socket

The way in which a connection is teminated and a socket is closed differs depending on the communication domain used (AF_INET/AF_INET6 or AF_ISO).

Terminating a connection in the AF_INET and AF_INET6 domains

In the AF_INET and AF_INET6 domains, a connection can be terminated using *soc_close()* or *shutdown()*. A socket can only be closed with *soc_close()*, but not with *shutdown()*.

When terminating a connection a distinction is made between a “graceful disconnect” and an “abortive disconnect”. This is handled by the transport system or rather the TCP protocol machine. One of the two options is selected using the *soc_close()* and *shutdown()* functions.

The following explanations of terminating a connection using *soc_close()* and *shutdown()* are based on the situation illustrated in figure 2. A client/server connection is to be terminated via which data has been transferred in both directions.

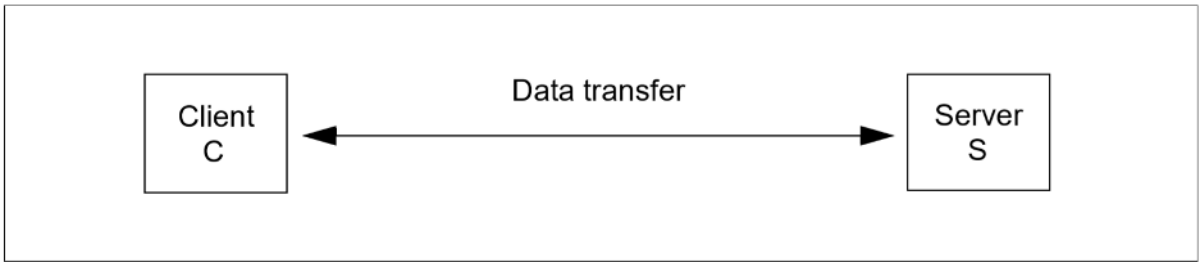


Figure 2: Client/server connection with bidirectional data transfer

Terminating a connection (“graceful”) using `soc_close()`

The following steps are executed:

1. Once the last data has been sent, server S initiates the termination of a connection on the socket in S using *soc_close()*. This disables writing for the socket in server S and the partner socket in client C is informed that the socket in S will no longer send data. This is a “graceful disconnect”. Following a “graceful disconnect”, the connection is still maintained, however data transmission from S to C is disabled.
2. Once the signal for “graceful disconnect” has been received, the client C user program can read all the data that has not yet been fetched until the end of the data flow is indicated with EOF.
3. Client C calls *soc_close()* for the socket in C. This sends a “graceful disconnect” to S and the connection is completely terminated. The socket is closed in C. The termination of the connection is reported in S and the socket is closed.

> ℹ If C answers the “graceful disconnect” event by calling *soc_close()* before attempting to read any existing data, the connection is completely terminated immediately and data is lost.

Terminating a connection (“graceful”) using `shutdown()`

The following steps are carried out:

1. Once the last data has been sent, server S initiates the termination of the connection on the socket in Server S using *shutdown(... , SHUT_WR)*. This disables writing for the socket in server S and the partner socket in client C is informed that the socket in S can no longer send data. This is a “graceful disconnect”. Following a “graceful disconnect”, the connection is still maintained, however data transmission from S to C is disabled.

2. Once the signal for “graceful disconnect” has been received, the client C user program can read all the data that has not yet been fetched until the end of the data flow is indicated with EOF.
3. Client C calls *shutdown(... , SHUT_WR)* for the socket in C. This sends a “graceful disconnect” to S and the connection is completely terminated.
4. The sockets in C and S are both closed with *soc_close()*.

> If C answers the “graceful disconnect” event by calling *shutdown(... ,SHUT_WR)* before attempting to read any existing data, the connection is completely terminated immediately and data is lost.

Terminating a connection (“abortive”) using `soc_close()`

The following steps are carried out:

1. Server S marks its socket interface with the SO_LINGER option of the *setsockopt()* function and sets the *l_linger* delay interval in the *linger* structure to 0.
2. When the server calls the *soc_close()* function, the “abortive” termination of the connection is initiated.There is no read or write access to the socket in Server S. The partner socket in client C is informed of the “abortive disconnect” and the socket in server S is closed.
3. Once the signal for “abortive disconnect” has been received, the client C user program can no longer read data. Any existing data that has not yet been fetched from the transport system is lost.
4. Client C can therefore only respond to the socket in server S with *soc_close()* and thus close the C socket.

Terminating a connection (“abortive”) using `shutdown()`

The following steps are executed:

1. Server S initiates the termination of the connection using *shutdown(..., SHUT_RDWR)*. There is no read or write access to the socket in server S now, and the partner socket in client C is informed of the “abortive disconnect”.
2. If the client C application program has not fetched any existing data from the transport system before receiving “abortive disconnect”, this data is lost.
3. It is therefore only meaningful to answer with *shutdown(...,SHUT_RDWR)* in client C and to close both sockets in server S and client C with *soc_close()*.

Terminating a connection in the AF_ISO domain

In the AF_ISO domain, only the *soc_close()* function is available for terminating a connection. The connection is completely aborted upon the first call of *soc_close()* for the socket of a connection end point. Data not yet fetched on the partner side is lost. Connection termination data, which was previously entered in the socket (see *getsockopt()*, *setsockopt()* in section "getsockopt(), setsockopt() - get and set socket options") can however be transmitted with the *soc_close()* function.

Terminating a connection (“abortive”) using *soc_close()*

The following steps are carried out:

1. If required, server S writes connection termination data to the socket using the TPOPT_DISC_DATA option of the *setsockopt()* function.
2. If the server calls the *soc_close()* function, the “abortive disconnect” is initiated. Here is no read or write access to the socket in server S now. The partner socket in client C is informed of the “abortive disconnect”. If available, the data referring to the termination of the connection is transmitted. The socket in server S is closed.
3. Once the signal for “abortive disconnect” has been received, the client C application program can read the connection abort data, if this has been transmitted from the server. The user program can no longer read any other data. Any existing data that has not yet been fetched from the transport system is lost.
4. Client C can therefore only respond to the signal of S with *soc_close()* and thus close the socket in C.

Multiplexing input/output

It is often useful to distribute inputs and outputs over several sockets. You can use either the *select()* or the *soc_poll()* function for this type of input/output multiplexing. However, it is recommended that you use the *select()* function.

Multiplexing input/output with the select() function

select() enables a program to monitor several connections simultaneously.

The following program section illustrates the use of *select()*.

```
#include <sys.time.h>
#include <sys.types.h>
...
char *readmask, *writemask, *exceptmask;
struct timeval timeout;
int nfds:
 ...
select(nfds, readmask, writemask, exceptmask, &timeout);
```

The parameters required by *select()* are three pointers to one bit mask each, which represents a set of socket descriptors:

- *select()* uses the bit mask passed with *readmask* to test from which sockets data can be read.
- *select()* uses the bit mask passed with *writemask* to test to which sockets data can be written.
- *select()* uses the bit mask passed with *exceptmask* to test which sockets have an exception pending.
  The *exceptmask* parameter is not evaluated by SOCKETS(BS2000) at present.

The bit masks for the individual descriptor sets are stored as bit fields in integer strings. The maximum required size of the bit fields can be determined via the *getdtablesize()* function (see section "getdtablesize() - get size of descriptor table"). The required memory should be requested from the system dynamically.

The *nfds* parameter specifies how many bits or descriptors are to be tested:
select() tests bits 0 to *nfds*-1 in each bit mask.
If you are not interested in one of the pieces of information (read, write or pending exceptions), you should pass the null pointer with the *select()* call for the parameter concerned.

You can modify the bit masks with macros. You should, in particular, set the bit masks to 0 before modifying them. The bit mask manipulation macros are described in section "select() - multiplex input/output" under the functional description of *select()*.

You can use the *timeout* parameter to define a timeout value if the selection process is to be limited to a predefined time. If you pass the null pointer with *timeout*, the execution of *select()* blocks for an unspecified time.
You can set polling by passing *timeout* a pointer to a *timeval* variable whose components are all set to 0.

After successful execution, the value returned by *select()* specifies the number of selected descriptors. The bit masks then indicate:

- which descriptors are ready for reading
- which descriptors are ready for writing

If *select()* terminates with a timeout, it returns the value 0. The bit masks are then unchanged.
If *select()* terminates with an error, it returns the value “-1” and the appropriate error code in *errno*. The bit masks are then unchanged.

After *select()* has been executed successfully, use the FD_ISSET(*fd*, &*mask*) macro call to check the status of a descriptor *fd*. The macro returns a value not equal to 0 if *fd* is a member of bit mask *mask*; otherwise, the value 0.

You can determine whether connection requests to a socket *fd* are waiting for acceptance by *accept()* by checking the “read” readiness of socket *fd*. To do this, you call *select()* and then the FD_ISSET (*fd*, *&mask*) macro. If FD_ISSET returns a value not equal to 0, this indicates “read” readiness of socket *fd*: which means that a connection request is pending on socket *fd*.

Example: Using select() to test for pending connection requests

The program code (for AF_INET) below results in a connection request being waited for. When it arrives, it is accepted and the program is terminated.

```
#include <stdlib.h>
#include <sys.types.h>
#include <sys.socket.h>
#include <sys.time.h>
#include <netinet.in.h>
#include <netdb.h>
#include <stdio.h>
#define TRUE 1
#define TESTPORT 5555
/*
 * This program uses select() to test whether someone is trying to set up
 * a connection and then calls accept().
 */

main()
{
   int sock;
   struct sockaddr_in server;
   struct sockaddr_in client;
   int clientlen;
   int msgsock;
   int fdsize;
   char * ready;
   struct timeval to;
   memset(&server,'\0',sizeof(server));
   memset(&client,'\0',sizeof(client));
   clientlen = sizeof(client);

    /* Request memory for testing the socket descriptors using
    soc_select() */
   if ((fdsize = getdtablesize()) < 0) {
             perror("get fd_size");
             exit(1);
}
      if (ready = ((fd_set *) memalloc(fdsize/8) == NULL))  {
         perror("no memory space");
         exit(1);
}
      /* Create socket. */
      sock = socket(AF_INET, SOCK_STREAM, 0);
      if (sock < 0) {
           perror("opening stream socket");
           exit(1);
   }
   /* Assign the socket a name using wildcards */
   server.sin_family = AF_INET;
```

```
   server.sin_addr.s_addr = htonl(INADDR_ANY);
   server.sin_port = htons(TESTPORT);
   if (bind(sock, (struct sockaddr *)&server, sizeof server) < 0) {
           perror("binding stream socket");
           exit(1);
   }
      /* Start acceptance of connections. */
      listen(sock, 5);
      do {
              memset(ready, 0, fdsize/8);
              FD_SET(sock, (fd_set *)&ready);
              to.tv_sec = 5;
              to.tv_usec=0;
              if (select(sock + 1, (fd_set *)ready, (fd_set *)0,
                (fd_set *)0, &to) < 0) {
                     perror("select");
                     continue;
              }
         if (FD_ISSET(sock, (fd_set *)ready)) {
         msgsock = accept(sock, (struct sockaddr *)&client, &clientlen);

         if (msgsock >= 0)
           {
            /* Successful acceptance of request to establish connection*/
            /* Follow-up processing of the data which is transferred */
            /* via this connection */
            printf("End of program after successful conection setup\n");
            break;
           }
         else
           {
            /* An error has occurred */
            /* Error message and possibly renewed waiting for a request */
            /* to establish a connection */
            printf("End of program: An error occurred during connection
            setup\n");
            break;
           }
       }
    } while (TRUE);
 exit(0);
}
```

Multiplexing input/output with the soc_poll() function

soc_poll() also enables a program to monitor several connections simultaneously.

The following program section illustrates the use of *soc_poll()*:

```
#include <sys.socket.h>
#include <sys.poll.h>
...
struct pollfd fds[3];
int timeout = 0;
unsigned long nfds = 3;

fds[0].events = POLLIN;
fds[1].events = POLLOUT

fds[2].events = POLLIN;
 ...
soc_poll(fds, nfds, timeout);
```

The socket descriptors and events to be tested are transmitted in an array of *pollfd* structure elements. *fds* is a pointer to this array. *nfds* specifies the number of structure elements.

In the example shown in this section these are descriptors 0...2 and the POLLIN and POLLOUT events. POLLIN indicates the “read” readiness and POLLOUT the “write” readiness of the socket.

The *timeout* parameter specifies how the *soc_poll()* function should behave if no event is to be tested:

- If *timeout* = 0, *soc_poll()* tests all specified descriptors of the event to be tested only once. *soc_poll()* is then reset, regardless of whether the test was successful or not.
- If *timeout* > 0 a waiting time is specified in seconds. During this waiting time *soc_poll()* is blocked as long as none of the events to be tested occur.
- If *timeout* = -1 *soc_poll()* is blocked until one of the events to be tested occurs.

The return value of *soc_poll()* indicates the frequency of the occurrence, i.e., at least one bit is set in the *revents* return field of the corresponding *pollfd* structure element.

pollfd structure as declared in <sys.poll.h>:

```
struct pollfd {
   int      fd;              /* socket file descriptor to poll*/
   short    events;          /* events on interest on fd*/
   short    revents;         /* events that occurred on fd */
};
```

Example: Using soc_poll() to test for pending connection requests

The following program code is the same as the previous example except that the *select()* function has been replaced with the *soc_poll()* function.

```
#include <sys.types.h>
#include <stdlib.h>
#include <sys.socket.h>
#include <sys.poll.h>
#include <netinet.in.h>
#include <netdb.h>
#include <stdio.h>
#define TRUE 1
#define TESTPORT 5555
/*
 * This program uses soc_poll() to check whether someone is attempting to
 * establish a connection, and then calls accept().
 */

main()
{
    int sock;
    struct sockaddr_in server;
    struct sockaddr_in client;
    int clientlen;
    int msgsock;
    struct pollfd fds[1];
    unsigned long nfds = 1;
    int timeout = 5;
    memset(&server,'\0',sizeof(server)); memset(&client,'\0',sizeof(client));

   clientlen = sizeof(client);

/* Initialize the fds structure arrays to request the "read" readiness of the
listen socket */
fds[0].fd = 0;
fds[0].events = POLLIN;
fds[0].revents = 0;

/* Create socket. */
      sock = socket(AF_INET, SOCK_STREAM, 0);
      if (sock < 0) {
              perror("opening stream socket");
              exit(1);
      }
   /* Assign the socket a name using wildcards */
   server.sin_family = AF_INET;
   server.sin_addr.s_addr = htonl(INADDR_ANY);
   server.sin_port = htons(TESTPORT);
   if (bind(sock, (struct sockaddr *)&server, sizeof server) < 0) {
         perror("binding stream socket");
         exit(1);
   }
      /* Start acceptance of connections. */
      listen(sock, 5);
      do {
```

```
              fds[0].fd = sock;
              if (soc_poll(fds, nfds, timeout)) <= 0){
                      perror("soc_poll");
                      continue;
     }
     else
     {
     if (fds[0].revents & POLLIN) {
         fds[0].revents = 0;
         msgsock = accept(sock, (struct sockaddr *)&client, &clientlen);
         if (msgsock >= 0)
           {
            /* Successful acceptance of request to establish connection*/
            /* Follow-up processing of the data which is transferred */
            /* via this connection */
            printf("End of program after successful conection setup\n");
            break;
           }
         else
           {
            /* An error has occurred */
            /* Error message and possibly renewed waiting for a request */
            /* to establish a connection */
            printf("End of program: An error occurred during connection
            setup\n");
            break;
           }
       }
     }
    } while (TRUE);
exit(0);
}
```

Interaction of the SOCKETS interface functions

The following figures illustrate the interaction between the functions of the SOCKETS(BS2000) interface. The individual functions are described in detail in the chapter "SOCKETS(BS2000) user functions".

Interaction between functions for connection-oriented communications

The way in which connection-oriented communications are performed differ depending on the communications domain used (AF_INET or AF_INET6, or AF_ISO).

Connection-oriented communication in AF_INET and AF_INET6

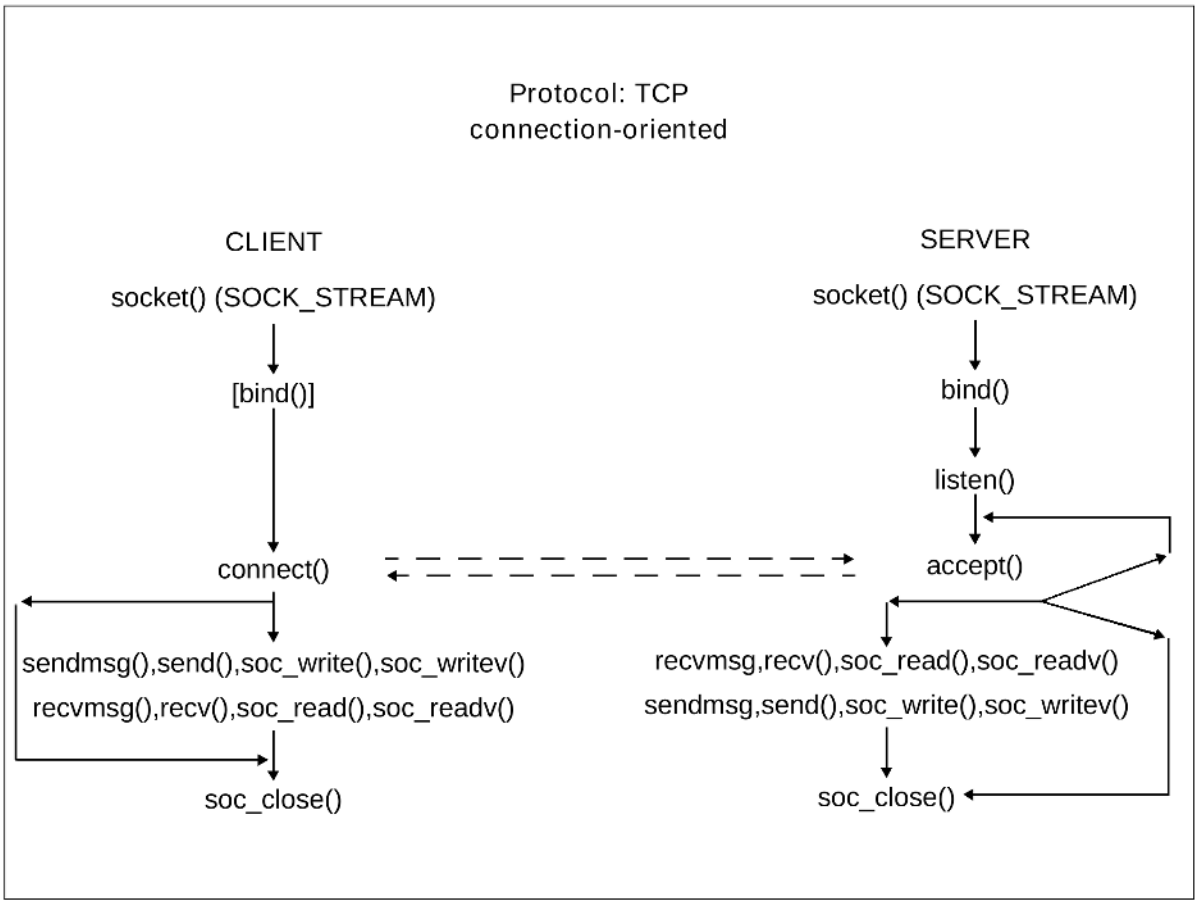


Figure 3: Interaction of the SOCKETS(BS2000) interface functions with stream sockets (AF_INET, AF_INET6)

Connection-oriented communication AF_ISO

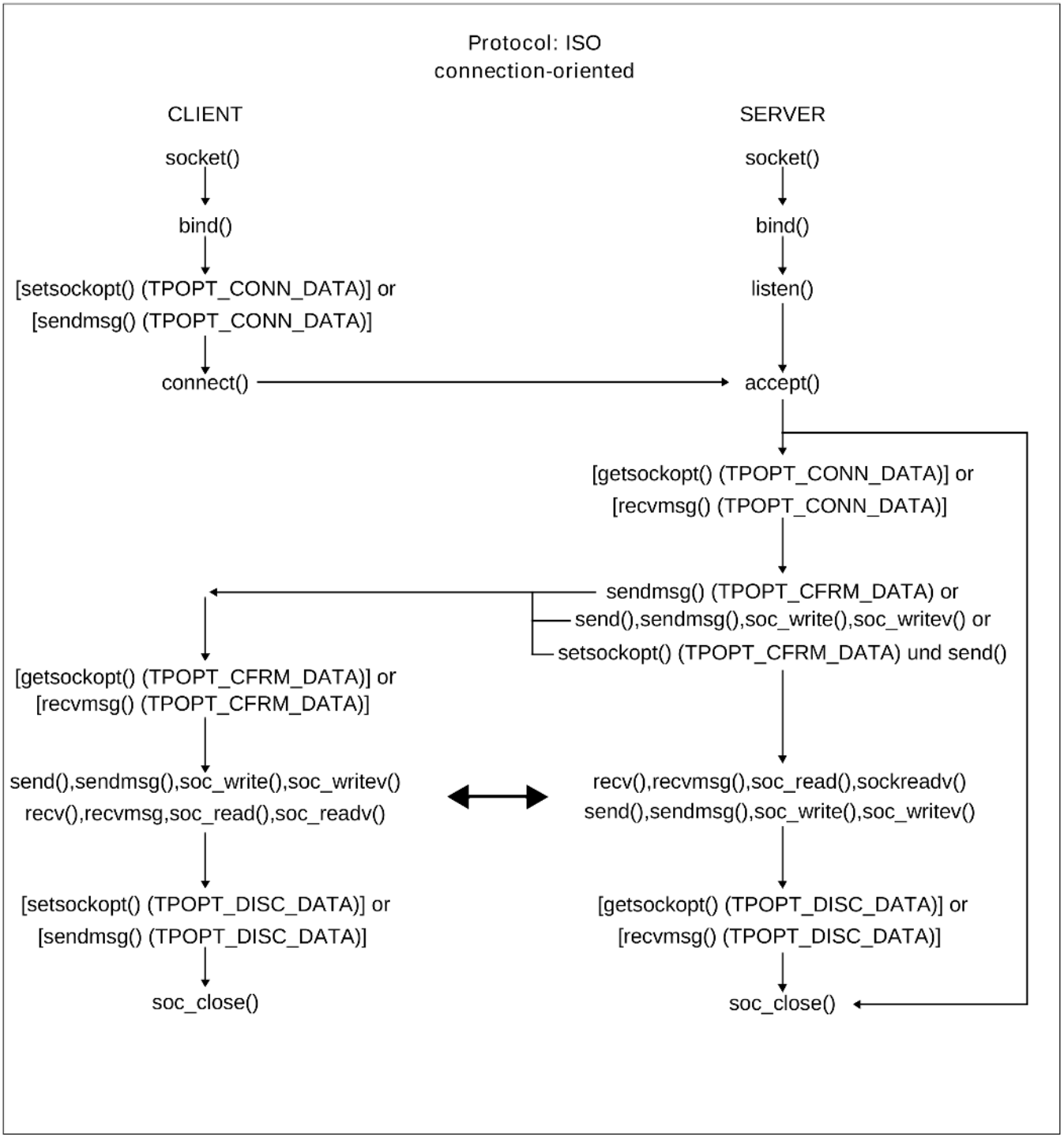


Figure 4: Interaction of the SOCKETS(BS2000) interface functions with stream sockets (AF_ISO)

Interaction between functions for connectionless communications

The figure below illustrates the interaction of the SOCKETS(BS2000) interface functions with datagram sockets (SOCK_DGRAM).

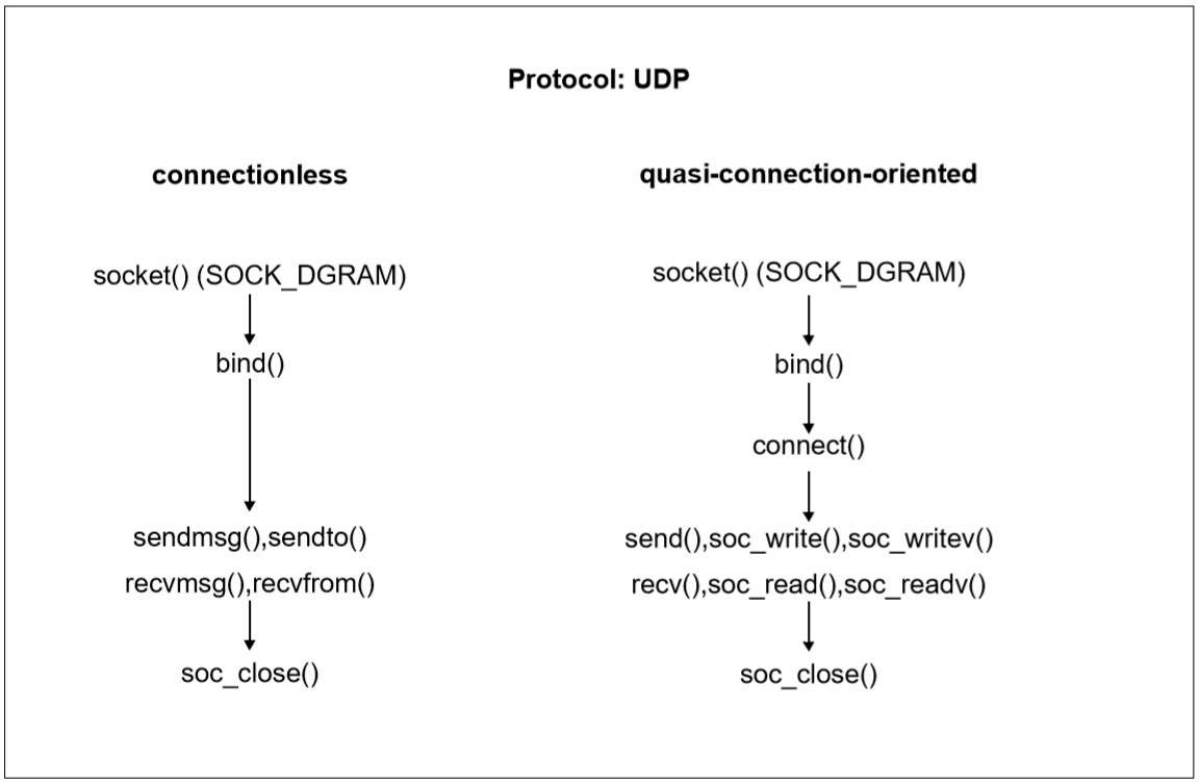


Figure 5: Interaction of the SOCKETS(BS2000) interface functions with datagram sockets

Address conversion with SOCKETS(BS2000)

In order to allow processes to communicate with one another on sockets, network addresses need to be determined and created. The SOCKETS(BS2000) library provides many different utility routines and macros for this purpose for the communications domains AF_INET and AF_INET6. These utility routines and macros are presented briefly in this chapter.
All utility routines are described in detail in the chapter “SOCKETS(BS2000) user functions”.

Before a client and server can communicate with each other, the client has to determine the service on the remote host. The following address conversion stages are required to determine the service concerned:

1. A service and a host are each assigned names for better legibility at the user program level, e.g. the service *login* on host *Monet*.
2. The system converts a service name to a service number (port number) and a host name to a network address (IPv4 or IPv6 address).

The following conversion functions are available:

- host names to network addresses, and vice versa
- network names to network numbers
- protocol names to protocol numbers
- service names to port numbers and the relevant protocol for communicating with the server

If you want to use one of these functions, you will need to include the <netdb.h> file. Program examples which use the conversion functions described below can be found in chapter “Client/server model with SOCKETS(BS2000)”.

Converting host names to network addresses and vice versa

There are special socket functions for converting host names to network addresses and vice versa in the AF_INET and AF_INET6 address families.

Socket functions for converting addresses in the AF_INET and AF_INET6 address families

The *getipnodebyname()* function converts a host name to an IPv4 or IPv6 address. A host name is passed when *getipnodebyname()* is called.

The *getipnodebyaddr()* function converts an IPv4 or IPv6 address to a host name. An IPv4 or IPv6 address is passed when *getipnodebyaddr()* is called.

The *inet_ntop()* function converts an Internet host name to a character string. This character string is returned as follows:

- in hexadecimal colon notation for AF_INET6
- in decimal dotted notation for AF_INET

The *inet_pton()* function converts an Internet host address in printable representation

- from a character string in decimal dotted notation to a binary IPv4 address (AF_INET).
- from a character string in hexadecimal colon notation to a binary IPv6 address (AF_INET6).

Abbreviated notation using two consecutive colons “::” is not supported for AF_INET6.

Socket functions address conversion which are only supported in AF_INET

The *gethostbyname()* function converts a host name to an IPv4 address. A host name is passed when *gethostbyname()* is called.

The *gethostbyaddr()* function converts an IPv4 address to a host name. An IPv4 address is passed when *gethostbyaddr()* is called.

gethostbyname() and *gethostbyaddr()*, as well as *getipnodebyname()* and *getipnodebyaddr(),* return a pointer to an object of data type *struct hostent* as their result.

The *hostent* structure is declared in <netdb.h> as follows:

```
struct hostent {
     char *h_name;          /* official host name */
     char **h_aliases;      /* alias list */
     int  h_addrtype;       /* address type */
     int  h_length;         /* length of the address (in bytes) */
     char **h_addr_list;    /* list of addresses for the host, */
                            /* terminated with the null pointer*/
};
#define h_addr h_addr_list[0]  /* first address, network byte order */
```

The *hostent* object returned by *gethostbyname()* and *gethostbyaddr()* and by *getipnodebyname()* and *getipnodebyaddr()* always contains the following information, if this is made available by the database:

- the official name of the host
- a list of the host aliases

- address type (domain)
- a list of addresses of variable length, terminated with the null pointer

The address list is required because a host normally has several addresses which are all assigned to the same host name. *h_addr* ensures backward compatibility and is defined as the first address in the address list of the *hostent* structure.

The *inet_ntoa()* function converts an IPv4 host address to a character string in accordance with the normal Internet dotted notation.

Converting protocol names to protocol numbers

The *getprotobyname()* function converts a protocol name to a protocol number. The protocol name is passed when *getprotobyname()* is called.

getprotobyname() returns a pointer to an object of type *struct protoent* as its result.

The *protoent* structure is declared in <netdb.h> as follows:

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

Converting service names to port numbers and vice versa

A service is expected to be on a specific port and use just one communications protocol. This view is consistent within the Internet domain but does not apply in some other network architectures. A service may also be available on several ports, in which case higher-level library functions have to be forwarded or extended.

The *getservbyname()* function converts a service name to a port number. The service name and, optionally, the name of a qualifying protocol are passed when *getservbyname()* is called.The *getservbyport()* function converts a port number to a service name. The port number and, optionally, the name of a qualifying protocol are passed when *getservbyport()* is called.

getservbyname() and *getservbyport()* return a pointer to an object of data type *struct servent* as their result.

The *servent* structure is declared in <netdb.h> as follows:

```
struct servent {
     char *s_name;          /* official name of the service */
     char **s_aliases;      /* alias list */
     int  s_port;           /* number of the port on which the service lies*/
     char *s_proto;         /* protocol used */
};
```

> Up to openNet Server V3.4 with SOCKETS(BS2000) V2.5, conversion took place on the basis of a static list contained in SOCKETS(BS2000).
> In openNet Server V3.5 and higher with SOCKETS(BS2000) V2.6, a services file with the default name SYSDAT.BCAM.ETC.SERVICES is offered which is managed by BCAM (see the “BCAM Volume 1/2” manual). This file is supplied with the default assignment of ports 1-1023. If you have appropriate user rights, you can modify this file. You can then modify default port assignments and add port assignments.

Example

The following program code returns the port number of the *telnet* service, which uses the TCP protocol:

```
struct servent *sp;
...
sp = getservbyname("telnet", "tcp");
```

Converting the byte order

If you use the address conversion functions described above, you will seldom have to directly handle addresses in an Internet user program. You can then develop services that are independent of networks to a large extent. However, some network dependency still remains, since the IP address has to be specified in a user program if a name is assigned to a service or socket.

Besides the library functions for converting names to addresses, there are also some macros which simplify the handling of names and addresses.

The host byte order and network byte order differ in some architectures. Because of this, programs sometimes have to change the byte order. The macros summarized in the table below convert bytes and integers from host byte order to network byte order, and vice versa.

Library macros for converting byte orders

Call	Meaning
htonl(*val*)	Convert 32-bit fields from host byte order to network byte order
htons(*val*)	Convert 16-bit fields from host to network byte order
ntohl(*val*)	Convert 32-bit fields from network to host byte order
ntohs(*val*)	Convert 16-bit fields from network to host byte order

The byte order conversion macros are needed because the operating system expects the IPv4 addresses in network byte order. The library functions which return network addresses supply them in network byte order, allowing them to be simply copied into the structures available to the system. You should therefore only encounter byte order problems when interpreting network addresses.

The host and network byte orders are identical in BS2000. The macros listed in the table are therefore defined as null macros (macros without contents). However, it is strongly recommended that you use the macros if you want to create portable programs.

In IPv6 implementation, network addresses are always expected in network byte order, i.e. there is no definition of a difference between host byte order and network byte order, and there is therefore no corresponding conversion function.
If necessary, only the port number has to be converted.

Example of address conversion

The client program code of the *remote login* shown below demonstrates the address conversion discussed in the preceding sections.

```
#include <sys.types.h>
#include <sys.socket.h>
#include <netinet.in.h>
#include <stdio.h>
#include <netdb.h>

main(argc, argv)
      int argc;
      char *argv[];
{
      struct sockaddr_in server;
      struct servent *sp;
      struct hostent *hp;
      int s;
      sp = getservbyname("login", "tcp");
      if (sp == NULL) {
              fprintf(stderr, "rlogin: tcp/login: unknown service\n");
              exit(1);
      }
      hp = gethostbyname(argv[1]);
      if (hp == NULL) {
              fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
              exit(2);
      }
      memset((char *)&server, 0, sizeof server);
      memcpy((char *)&server.sin_addr, hp->h_addr, hp->h_length);
      server.sin_family = hp->h_addrtype;
      server.sin_port = sp->s_port;
      s = socket(AF_INET, SOCK_STREAM, 0);
      if (s < 0) {
              perror("rlogin: socket");
              exit(3);
      }

      /* Connect does the bind for us */
      if (connect(s, &server, sizeof server) < 0) {
              perror("rlogin: connect");
              exit(5);
      }
exit(0);
}
```

Extended SOCKETS(BS2000) functions

The procedures described in the preceding chapters will suffice in most cases for developing distributed applications. However, it may sometimes be necessary to make additional use of the following SOCKETS(BS2000) features:

- non-blocking sockets
- multicast messages
- socket options
- support of virtual hosts
- Handoff (move an accept socket)
- raw sockets

Non-blocking sockets

With non-blocking sockets, the *accept()*, *connect()* and all input/output functions are terminated if they cannot be executed immediately. The function concerned then returns an error code. In other words, in contrast to normal sockets, non-blocking sockets prevent a process from being interrupted because it has to wait for the termination of *accept()*, *connect()* or I/O functions. You can mark a socket created with *s=socket()* as non-blocking with the *soc_ioctl()* function (see section "soc_ioctl() (ioctl) - control sockets") as follows:

```
#include <ioctl.h>
...
int s;
 ...
int block;
s = socket(AF_INET, SOCK_STREAM, 0);
 ...
block = 1;
...
if (soc_ioctl(s, FIONBIO, &block) < 0) {
      perror("soc_ioctl(s, FIONBIO, block) <0");
      exit(1);
}
 ...
```

You should particularly watch out for the EWOULDBLOCK error when executing the *accept()*, *connect()* or I/O functions on non-blocking sockets. EWOULDBLOCK is stored in the global *errno* variable and occurs if a function which normally blocks is executed on a non-blocking socket.

The *accept()* and *connect()* functions as well as all read and write operations can return the EWOULDBLOCK error code. Processes should therefore be prepared to handle such return values: for example, even if the *send()* function is not executed completely, it may still be meaningful with stream sockets to execute at least part of the write operations. In this case, *send()* only considers the data that can be sent immediately. The return value indicates the amount of data already sent.

> The “non-blocking” property of a listen socket is not passed onto a socket created with *accept()*.

Multicast messages (AF_INET, AF_INET6)

In contrast to unicast messages, a sender can use multicast messages to reach more than one receiver. However, unlike with broadcast messages, a selection takes place because each recipient must join a multicast group to receive such messages. A sender does not log in, but it is possible to receive locally at the same time.

Multicast messages save on system resources and bandwidth in the network, especially when an application is involved for which there is only one send direction. Practical application scenarios for multicast messages are file streams, e.g. for music or video, video conferences or news or stock exchange tickers.

Multicast messages are transferred using datagram packets, in other words using an insecure service. The application must therefore guarantee that the data reaches the receiver with its integrity ensured. And it must make sure that the data is only supplied to authorized receivers.

Prerequisites

Separate areas are used for multicast message transfer in both the IPv4 address space and the IPv6 address space. The communications systems used, such as BCAM in BS2000, must permit and support multicast operation.

Multicast operation with the default settings is permitted in BCAM. If there is any doubt, the configuration should be checked, and if necessary intervention should take place on an administrative level. Please refer to the “BCAM Volume 1/2” manual for details.

If the messages are to leave the local area, multicast-capable routers are also required which must be configured accordingly.

SOCKETS functions for multicast support

SOCKETS(BS2000) offers functions for transferring and receiving multicast messages and for logging onto or logging off from multicast groups.

The address range 224.0.0.0 through 239.255.255.255 is provided for IPv4; addresses 224.0.0.0 through 224.0.0.255 are reserved for local applications and are not routed.

The multicast address range in IPv6 begins with the prefix FF, followed by 4-bit flags and a 4-bit scope. The precise assignment is described in the RFC “IP Version 6 Addressing Architecture,” currently RFC 4291.

Reserved multicast addresses in IPv4 and IPv6 can be viewed at the Internet Assigned Numbers Authority (IANA).

Socket options for AF_INET

In the AF_INET address family, the transfer of multicast messages is supported by the following socket options:

- IP_ADD_MEMBERSHIP: log on to a multicast group.
  After logging on, data of this group is delivered.
- IP_DROP_MEMBERSHIP: log off from a multicast group
- IP_MULTICAST_IF: display or define the sender interface
- IP_MULTICAST_TTL: display or define the multicast hop limit
- IP_MULTICAST_LOOP: reception is possible on the local sending host

Socket options for AF_INET6

In the AF_INET6 address family, the transfer of multicast messages is supported by the following socket options:

- IPV6_JOIN_GROUP: log on to a multicast group.
  After logging on, data of this group is delivered.
- IPV6_LEAVE_GROUP: log off from a multicast group
- IPV6_MULTICAST_IF: display or define the index of the sender interface
- IPV6_MULTICAST_HOPS: display or define the multicast hop limit
- IPV6_MULTICAST_LOOP: reception is possible on the local sending host

Socket options

You can use the *setsockopt()* and *getsockopt()* functions to set or query the current value of various options for sockets.

For example, you set options to activate the *keepalive* monitoring for a socket connection or to modify the time interval for monitoring.

For example, you can set options to identify a socket for sending broadcast messages.

The general format of the calls is as follows:

```
setsockopt(s, level, optname, optval, optlen);
```

```
getsockopt(s, level, optname, optval, optlen);
```

s designates the socket for which the option is to be set or queried.
level defines the protocol level to which the option belongs. This is normally the socket level that is indicated by the SOL_SOCKET symbolic constant (for AF_INET and AF_INET6) or SOL_TRANSPORT (for AF_ISO). SOL_SOCKET and SOL_TRANSPORT are defined in <sys.socket.h>.

Other *level*s are SOL_GLOBAL, IP_PROTO_TCP, IP_PROTO_IPv4, IP_PROTO_IPv6, IPPROTO_ICMP und IPPROTO_ICMPv6. For reasons of compatibility, both IP_PROTO_IP and IP_PROTO_IPv4 are supported. You will find a description of these *level*s in the description of the *getsockopt()* and *setsockopt()* functions in section "getsockopt(), setsockopt() - get and set socket options".

The socket option is specified in *optname* and is also a symbolic constant defined in <sys.socket.h>.

optval is a pointer to the option value. You use *optval* with *setsockopt()* to enable/disable the *optname* option for socket *s*. With *getsockopt()*, *optval* informs you as to whether the *optname* option is enabled or disabled for socket *s*.

With *setsockopt()*, *optlen* defines the length of the option value *optval*, With *getsockopt()*, *optlen* is a pointer which defines the size of the memory area to which *optval* points. On returning from *getsockopt()*, *optlen* points to an integer value that indicates the current length of the option value returned in *optval*.

Support of virtual hosts

It is possible to define a number of virtual hosts in addition to a real host (standard host). The real host and the virtual host are created using the static or dynamic generation which BCAM offers. Additional steps must be taken in order to ensure that the applications can be addressed. It is possible for a virtual host to have a number of IP addresses.

This functionality has no impact on existing or new standard applications. The functionality is made available with the new subfunctions *soc_ioctl()* and *getsockopt()*, *setsockopt()*, which allow the sockets user to obtain the necessary information on the configuration with virtual hosts and to use this information appropriately in the applications.
The decision as to the host on which the application will run is taken when the *bind()* function is executed. At this time, the socket must be informed of the host to be addressed.
In the event of single addressing, this is done automatically using the specified IP address, and in the event of ANYADDR or LOOPBACK addressing, it is necessary to specify the relevant BCAM host name. Where required, this name must be entered in the socket using the new *setsockopt()* subfunction SO_VHOSTANY before *bind()* is executed. This is necessary because it is not possible to uniquely assign ANYADDR or LOOPBACKADDR to a host.
The new *soc_ioctl()* subfunctions SIOCGLVHNUM and SIOCGLVHCONF can be used to determine the number of virtual hosts and the associated BCAM host names and socket host names.
Note that it is, of course, still possible to assign sockets applications to a virtual host using the application table in BCAM.
This is why it is also possible to address the real host using the new functionality.

BCAM host name:

> The name is eight characters in length. Alphanumeric characters and the special characters #, @, $ or blanks can be used at the end of the name. As a rule, uppercase characters should be used, but the name is case-sensitive. Names comprising numeric characters only are not permitted.

> **i** By default, the HOST-ALIASING functionality is active in the BCAM transport system. This can lead to undesired side-effects if the functionality for supporting virtual hosts is used.
>
> HOST-ALIASING means that a request to establish a connection to a virtual host is forwarded to a real host if the relevant port number is only open in the real host.
>
> HOST-ALIASING can be suppressed at the listen socket of the real host using *setsockopt(fd, SOL_SOCKET, SO_DISHALIAS, 1, 4)*. If this flag is set in the socket prior to *bind()*, the subsequent *bind()* deactivates HOST-ALIASING for this port number in the BCAM transport system.
>
> The result of this is that a request to establish a connection using this port number on a virtual host can only be successful if the port with the corresponding address is actually open on the virtual host. Requests are then not forwarded to the real host for this application.

Handoff (move an accept socket)

General description

The handoff functionality makes it possible to move the endpoint of a socket connection without the need to interrupt establishment of the connection. In other words, the application that is actively establishing the connection does not have to repeat the process. This is made possible by an extended functionality taking into account the ISO service functionality in the AF_ISO domain. In order to achieve this, it is necessary to establish a local AF_ISO connection for internal communication.

One practical example is when connection requests are accepted by a central listener and the endpoint is moved to an assigned server.

Execution of the function

New subfunctions for *sendmsg()*, *recvmsg()*, *setsockopt()* and *getsockopt()* are provided for this functionality.

To achieve this, the following structures are required in the header file *sys.socket.h*:

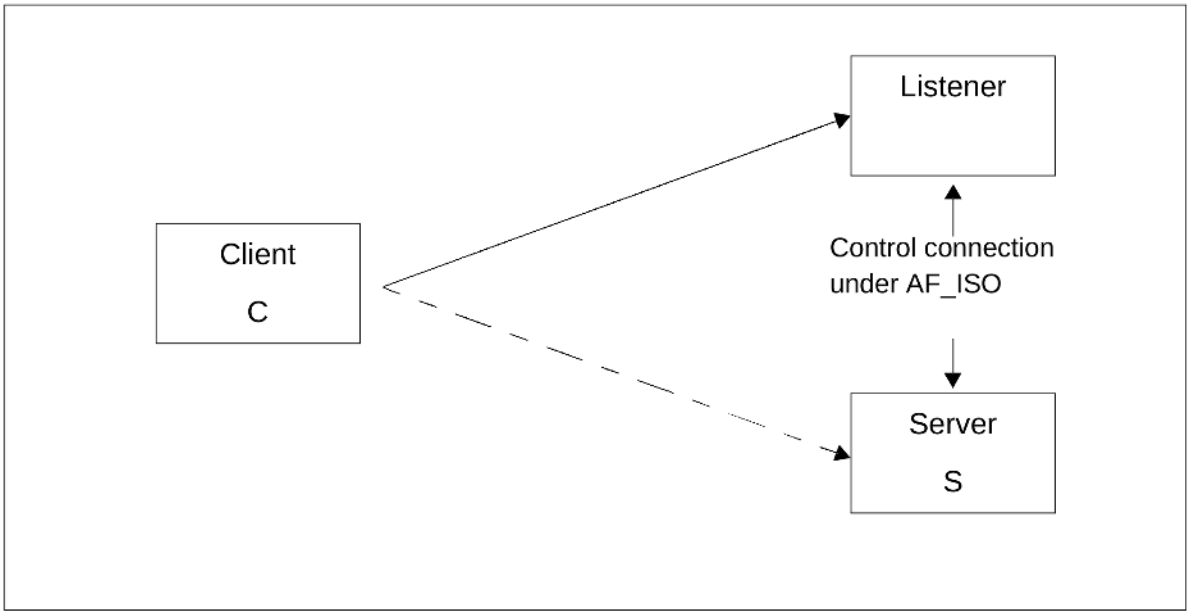
```
/*
 * struct instead of cmsghdr in case of Handoff-Handling
*/

struct red_info_tcp {
        short   fd;                         /* file descriptor  (listener)          */
        short   port;                       /* port number                          */
        short   domain;                     /* address family                       */
        short   flags;                      /* flags of success                     */
        int     cid;                        /* cid                                  */
        int     if_index;                   /* interface index listener process     */
        int     rwindow;                    /* max read  window                     */
        int     wwindow;                    /* max write window                     */
};
struct red_info_iso {
        short   fd;                         /* file descriptor  (listener)          */
        short   domain;                     /* address family                       */
        short   flags;                      /* flags of success                     */
        short   tsellen;                    /* length of TSEL                       */
        int     rwindow;                    /* max read  window                     */
        int     wwindow;                    /* max write window                     */
        char    tsel[32];                   /* TSEL application                     */
        char    tesn[8];                    /* TESN hostname                        */
};
struct red_info_svrs {
        short   domain;                     /* domain    (server[accept])           */
        short   fd_server;                  /* file descriptor(server[accept])      */
        int     tsor_server;                /* tsap_open_reference 1. server_socket */
        int     cref_server;                /* cref server[accept]_socket           */
};
struct cmsg_redhdr {
        u_int   cmsg_len;                   /* data byte count, including hdr       */
        int     cmsg_level;                 /* originating protocol                 */
        int     cmsg_type;                  /* protocol-specific type of operation  */
        union {
        char    tsap_name[TSAPNAMMAXLEN];   /* needed tsap_name for shared tsap     */
        struct  red_info_iso red_liso;      /* needed tsap_name for iso shared tsap */
        struct  red_info_tcp red_ltcp;      /* Info of listen socket                */
        struct  red_info_tcp red_ctcp;      /* Info of client                       */
        struct  red_info_svrs red_svrs;     /* Info of server socket ("accept")     */
       } cmsg_redhdr_info;
        short   bind_ok;                    /* open shared TSAP successfull         */
        short   handoff_ok;                 /* handoff successfull                  */
        short   tsap_name_len;              /* length of tsap_name                  */
        short   fd_server;                  /* file descriptor redirected socket    */
        short   domain;                     /* address family                       */
        int     tsn;                        /* tsn server-process                   */
#define  redhdr_tsap_name          cmsg_redhdr_info.tsap_name
#define  redhdr_red_liso           cmsg_redhdr_info.red_liso
#define  redhdr_red_ltcp           cmsg_redhdr_info.red_ltcp
#define  redhdr_red_ctcp           cmsg_redhdr_info.red_ctcp
#define  redhdr_red_svrs           cmsg_redhdr_info.red_svrs
};
```

Execution sequence

Acceptance: client = C, listener = L, server = S
The listen socket of listener L has the file descriptor *fd* = 1.

A connection under the AF_ISO domain is established between listener L and server S using*fd* 0 on the listener side and *fd* 0 on the server side. This connection can be established either in blocking mode or in non-blocking mode.

In non-blocking mode, it is mandatory for pending events to be polled using *select()* or *soc_poll()*.



The AF_ISO listen socket on the server side has the *fd* 0. The endpoint is moved in the following stages:

1. C establishes a connection to L.
2. The connection is accepted by L and moved to S.
3. In L, *sendmsg()* is used to forward to S information on the domain of the connection established by C and the *fd* of the accept socket for this connection over the local AF_ISO connection.

   *int sendmsg(int s, struct msghdr * msg, int flags)*;

   msg.msg_control is a pointer to a structure of the type *cmsg_redhdr*.

   The length of *struct cmsg_redhdr* is entered in *msg.msg_control_len*.

   cmsg_redhdr.cmsg_len = *sizeof(cmsg_redhdr)*
   cmsg_redhdr.cmsg_level = SOL_TRANSPORT
   cmsg_redhdr.cmsg_type = TPOPT_REDI_DATA

   The address family of the endpoint to be moved must be entered in *cmsg_redhdr.domain* and depending on this address family, the *fd* of the accept socket must be entered in *cmsg_redhdr.redhdr_red_liso* (for AF_ISO) or *cmsg_redhdr.redhdr_red_ltcp* (for AF_INET or AF_INET6) in the element *fd*.
4. The S side reads from the AF_ISO connection using *recvmsg()*.

   *int recvmsg(int s, struct msghdr * msg, int flags)*;

   msg.msg_control is a pointer to a structure of the type *cmsg_redhdr*.

cmsg_redhdr.cmsg_len = sizeof(cmsg_redhdr)
cmsg_redhdr.cmsg_level = SOL_TRANSPORT
cmsg_redhdr.cmsg_type = TPOPT_REDI_DATA

Action on S:

On the basis of the information from L contained in the structure of the type *cmsg_redhdr* that has been passed, a new connection endpoint is generated using an internal *bind()*.
The *fd* of this endpoint is returned in *cmsg_redhdr.fd_server* together with the address family in *cmsg_redhdr.domain*.

5. The S side passes the information that the new connection endpoint has been created to the L side.

   *int sendmsg(int s, struct msghdr * msg, int flags)*;

   msg.msg_control is a pointer to a structure of the type *cmgs_redhdr*.

   cmsg_redhdr.cmsg_ len = sizeof(cmsg_redhdr)
   cmsg_redhdr.cmsg_level = SOL_TRANSPORT
   cmsg_redhdr.cmsg_type = TPOPT_REDI_BDOK

   The address family of the socket for the new endpoint must additionally be entered in *cmsg_redhdr.domain* and the *fd* of this socket must be entered in *cmsg_redhdr.fd_server*.

6. The L side must wait for the information that the new endpoint is available before the end point can actually be moved.

   *int recvmsg(int s, struct msghdr * msg, int flags)*;

   msg.msg_control is a pointer to a structure of the type *cmsg_redhdr*.

   cmsg_redhdr.cmsg.len = sizeof(struct cmsg_redhdr)
   cmsg_redhdr.cmsg_level = SOL_TRANSPORT
   cmsg_redhdr.cmsg_type = TPOPT_REDI_BDOK

   The address family of the endpoint to be moved must additionally be entered in *cmsg_redhdr.domain* and depending on this address family, the *fd* of the accept socket must be entered in *cmsg_redhdr. redhdr_red_liso* (for AF_ISO) or *cmsg_redhdr.redhdr_red_ltcp* (for AF_INET or AF_INET6) in the element *fd*.

   The *cmsg_redhdr.bind_ok* field can be used to check whether successful creation of the new endpoint on the S side has been acknowledged with REDBIND_OK.

   A data stop is then triggered internally for this connection endpoint. This means that data can be sent from the client, but it is no longer delivered to the old connection end point.

7. *setsockopt()* is then issued on the L side to move the functionality of the endpoint. This means that the partner information entered in the accept socket is transferred to the socket of the new end point in the server.

   *int setsockopt(int s, int level, int optname, char * optval, int optlen)*;

   optval is a pointer to a structure of the type *cmsg_redhdr.*
   level = SOL_TRANSPORT
   optname = TPOPT_REDI_CALL

   cmsg_redhdr.cmsg_len = sizeof(struct cmsg_redhdr)
   cmsg_redhdr.cmsg_level = SOL_TRANSPORT
   cmsg_redhdr.cmsg_type = TPOPT_REDI_CALL*getsockopt()*

8. is issued on the S side to wait for data from the accept socket on the L side.

   *int getsockopt(int s, int level, int optname, char * optval, int* optlen)*;

optval is a pointer to a structure of the type *cmsg_redhdr.*
level = SOL_TRANSPORT
optname = TPOPT_REDI_CALL

cmsg_redhdr.cmsg_len = sizeof(struct cmsg_redhdr)
cmsg_redhdr.cmsg_level = SOL_TRANSPORT
cmsg_redhdr.cmsg_type = TPOPT_REDI_CALL

The address family must be entered in *cmsg_redhdr.domain* and the *fd* of the socket for the new endpoint must be entered in *cmsg_redhdr.fd_server*.

Once the event has been received and picked up, the connection environment is finally established and the data stop for the connection is canceled. This means that data is now delivered to the new connection endpoint.

The accept socket of the original endpoint can now be closed with *soc_close()*, as can the AF_ISO connection for handoff communication.

Raw sockets

A raw socket enables both an ICMP protocol header, e.g. for an ICMP echo request, and an ICMPv6 protocol header, e.g. for an ICMPv6 echo request, to be written.

ICMP

The ICMP protocol (which must always be viewed in conjunction with IPv4) enables you to test whether a data packet reaches an end system (host) and whether it is acknowledged. A detailed description of ICMP is provided in RFC 792.

Please note the following two special features in the format of the protocol and of the data:

- The ICMP header checksum must be generated by the application.
- The socket's port number is expected as the identifier. Before sending the message, you should therefore execute a *bind()* on the raw socket.

Before you call the *bind()* function to reserve a port, you must enable the delivery of possible ICMP error messages for this socket (see section "getsockopt(), setsockopt() - get and set socket options"):
setsockopt(...,IPPROTO_IPV4, IP_RECVERR,...,...)

The ICMP header is four bytes long. The length of the following data is variable. The following applies for the ECHO-REQUEST and ECHO-REPLY types: The first word contains the identifier (port number) and the sequence number. The next two words contain a timestamp. The time is contained in seconds in the first word and in microseconds in the second word:

00	Type	Code	Checksum
04	Identifier		Sequence#
08	Data (Timestamp struct timeval/tv_s)		
0C	Data (Timestamp struct timeval/tv_us)		
10	Data (Testpattern)		
14	Data (Testpattern)		

The associated IPv4 header is generated by the transport system. However, the application has the option of determining the hop limit. For this purpose you must set the raw socket appropriately before you send the data packet (ICMP message).

To set the hop limit specifically, use the function *setsockopt(..., IPPROTO_ICMP, IP_TTL,...,...)* (see section "getsockopt(), setsockopt() - get and set socket options").

The ICMP echo request message is sent using *sendmsg()*. The end system's response is received as an ICMP echo reply message using *recvmsg()*.

ICMPv6

The ICMPv6 protocol (which must always be viewed in conjunction with IPv6) enables you to test whether a data packet reaches an end system (host) and whether it is acknowledged.
A detailed description of ICMPv6 is provided in RFC 4443.

Please note the following two special features in the format of the protocol and of the data (as with ICMP):

- The ICMPv6 header checksum must be generated by the application.
- The socket's port number is expected as the identifier. Before sending the message, you should therefore execute a *bind()* on the raw socket.

Before you call the *bind()* function to reserve a port, you must enable the delivery of possible ICMPv6 error messages for this socket (see section "getsockopt(), setsockopt() - get and set socket options"):
setsockopt(...,IPPROTO_IPV6, IPV6_RECVERR,...,...)

The ICMPv6 header is four bytes long. The length of the following data is variable. The following applies for the ECHO-REQUEST and ECHO-REPLY types: The first word contains the identifier (port number) and the sequence number. The next two words contain a timestamp. The time is contained in seconds in the first word and in microseconds in the second word:

00	Type	Code	Checksum
04	Identifier		Sequence#
08	Data (Timestamp struct timeval/tv_s)		
0C	Data (Timestamp struct timeval/tv_us)		
10	Data (Testpattern)		
14	Data (Testpattern)		

The associated IPv6 header is generated by the transport system. However, the application has the option of determining the hop limit. For this purpose you must set the raw socket appropriately before you send the data packet (ICMPv6 message).

To set the hop limit specifically, use the function *setsockopt(..., IPPROTO_ICMPV6, IPV6_HOPLIMIT,...,...)* (see section "getsockopt(), setsockopt() - get and set socket options").

The ICMPv6 echo request message is sent using *sendmsg()*. The end system's response is received as an ICMPv6 echo reply message using *recvmsg()*.

Error analysis

During communication between sockets a few different problems can arise. Detailed error reports are necessary to be able to analyze them. By setting the flag MSG_ERRQUEUE additional information can be obtained for connectionsless communications in case of an error.

> ¡ This functionality requires BCAM V25 or newer.

At this time, MSG_ERRQUEUE is only supported for the function *recvmsg()*.

The following program section illustrates the use of *recvmsg()*:

```
#include <sys.socket.h>
#include <sys.uio.h>
...
int s;
struct msghdr *msg;
int flags = MSG_ERRQUEUE;
 ...
recvmsg(s, msg, flags);
```

For a successful call of *recvmsg()* the socket, where the message is to be received, an empty *msghdr* structure with sufficient buffer space and MSG_ERRQUEUE as set flag is needed. Detailed information regarding the function *recvmsg()* can be found in recvmsg() - receive a message from a socket.

Errors that arise while receiving messages are saved in a *cmsghdr* structure. The flag MSG_ERRQUEUE is added to *msg->msg_flags* and a pointer is set to the *cmsghdr* structure in the field *msg->msg_control* subsequently.

> ¡ Errors reports can only be received with MSG_ERRQUEUE if the socket option IP_RECVERR (for AF_INET) or IPV6_RECVERR (for AF_INET6) has been set. These error reports will be emptied if the flag IP_RECVERR or IPV6_RECVERR is unset.

msghdr structure

The *msghdr* structure is declared in <sys.socket.h> as follows:

```
struct msghdr {
        caddr_t       msg_name;        /* optional address */
        int           msg_namelen;     /* length of the address */
        struct iovec  *msg_iov;        /* scatter/gather fields */
        int           msg_iovlen;      /* number of elements in msg_iov */
        caddr_t       msg_control;     /* auxiliary data */
        int           msg_controllen;  /* length of the buffer for auxiliary data */
        int           msg_flags;       /* flag for received message */
};
struct msghdr *msg;
```

iovec structure
The *iovec* structure is declared in <sys.uio.h> as follows:

```
struct iovec{
        caddr_t  iov_base;  /* buffer for auxiliary data */
        int      iov_len;   /* buffer length */
};
```

cmsghdr structure
The c*msghdr* structure is declared in <sys.socket.h> as follows:

```
struct cmsghdr {
        u_int   cmsg_len;      /* number of data bytes incl. header */
        int     cmsg_level;    /* generating protocol */
        int     cmsg_type;     /* protocol-specific type */
        /* followed by u_char cmsg_data[] */
};
struct cmsghdr *cmsg;
```

The fields of the iovec structure are used to store the received L4-header (TCP, UDP or ICMP).

The fields *cmsg->cmsg_level* and *cmsg->cmsg_type* describe, which data is written into the data area of the *cmsghdr* structure. The error information itself is stored there via a *sock_extended_err* structure. The data can be accessed with the CMSG_DATA macro.

sock_extended_err structure
The *sock_extended_err* structure is declared in <linux.errqueue.h> as follows:

```
struct sock_extended_err {
        u_int32_t  ee_errno;   /* copy of errno number */
        u_int8_t   ee_origin;  /* where the error originated */
        u_int8_t   ee_type;    /* type of error */
        u_int8_t   ee_code;    /* error code, specifies the error */
        u_int8_t   ee_pad;     /* padding */
        u_int32_t  ee_info;    /* additional information */
        u_int32_t  ee_data;    /* other data */
};
struct sock_extended_err *ext_err;
```

More detailed information about the error is contained in the fields *ext_err->ee_type* and *ext_err->ee_code*. A list of all possible error codes can be found in the header files <ip.icmp.h> and <icmp6.h>.

Example: Obtaining detailed error information with MSG_ERRQUEUE and recvmsg()

The program code (for AF_INET) below shows the reception and handling of errors with MSG_ERRQUEUE for connectionless (in this case an ICMP protocol) communications.

```
#include <stdlib.h>
#include <sys.socket.h>
#include <sys.uio.h>
#include <sys.types.h>
#include <netinet.in.h>
#include <netdb.h>
#include <stdio.h>
#include <ip.icmp.h>
#include <linux.errqueue.h>
main(){
        int sock;
        int type, code, recv_id, recv_seq; /* Integer for the data of the icmp protocol */
        int val = 1;
        struct msghdr msg;
        struct iovec iov;
        struct cmsghdr *cmsg;
        struct sock_extended_err *ext_err = NULL;
        struct sockaddr_in from;
        struct sockaddr_in recv;
        struct icmp *icmp;
        char buffer[1280];
        char *buffer_ptr = buffer;
        char control[1024];
        int error_type;
        int error_code;
        ...
        /* For preparation a socket has to be created and bound (chapter 2.5 and chapter
2.6). */
        ...
        /* Setting IP_RECVERR to be able to receive error reports */
        setsockopt (sock, IPPROTO_IPV4, IP_RECVERR, &val, sizeof (val));
        ...
        /* Afterwards a packet is sent via connectionless communication (chapter
2.7.2).
        Now the reply needs to be received.
        Before that can happen, the socket must wait with select() or poll() (chapter 2.10)
*/
        ...
        /* Filling out the fields of the msghdr structure */
        msg.msg_name = &from; /* Contains the destination address */
        msg.msg_namelen = sizeof (from);
        msg.msg_control = control; /* Memory for the control message structure (cmsg) */
        msg.msg_controllen = sizeof (control);
        iov.iov_base = buffer; /* Protocol header and data */
        iov.iov_len = sizeof (buffer);
        msg.msg_iov = &iov;
        msg.msg_iovlen = 1;
        recvmsg(sock, &msg, MSG_ERRQUEUE);
        icmp = (struct icmp *) buffer_ptr; /* Reading from msg.msg_iov (in case of an ICMP
packet) */
        type = icmp->icmp_type; /* Type of the received ICMP packet */
        code = icmp->icmp_code; /* Contains more detailed information about the packet type
```

```
*/
        recv_id = ntohs (icmp->icmp_id); /* Only for type Echo Reply: Contains ID of the
Echo Request */
        recv_seq = ntohs (icmp->icmp_seq); /* Only for type Echo Reply: Contains sequence of
the Echo Request */
        /* recv_id and recv_seq specify to which (previously sent) package the reply belongs
*/
        for (cmsg = CMSG_FIRSTHDR (&msg); cmsg; cmsg = CMSG_NEXTHDR (&msg, cmsg)) {
                void *ptr = CMSG_DATA (cmsg);
                if (cmsg->cmsg_type == IP_RECVERR) {
                        ext_err = (struct sock_extended_err *) ptr; /* Reading from the data
area */
                        error_type = ext_err->ee_type; /* Classifies the error of the
received error packet */
                        error_code = ext_err->ee_code; /* Contains more detailed information
about the error packet */
                        memcpy (&recv, SO_EE_OFFENDER (ext_err), sizeof (struct
sockaddr_in));
                        /* Saving the address of the sender of the error package in a
sockaddr_in structure */
                }
        }
}
```

¡ This example code does not check the functions return values.

¡ Important: When IP_RECVERR or IPV6_RECVERR is set, the error packages have to be picked up in a timely manner (with the flag MSG_ERRQUEUE)! A maximum of 32 error packages will be stored in the buffer. Packaged not picked up will be deleted as soon as this count is reached.

More information regarding the *cmsghdr* structure and its macros can be found at CMSG-Macros.

CMSG-Macros

There are some macros that can help gaining various information from the cmsghdr structure. This is necessary to obtain data from certain socket options (getsockopt(), setsockopt() - get and set socket options) and detailed error reports (Error analysis).

> This functionality requires BCAM V25 or newer.

The macros are defined as follows:

```
#include <sys/socket.h>
struct cmsghdr  *CMSG_FIRSTHDR(struct msghdr *msg);
struct cmsghdr  *CMSG_NXTHDR(struct msghdr *msg, struct cmsghdr *cmsg);
int             CMSG_ALIGN(int length);
int             CMSG_SPACE(int length);
int             CMSG_LEN(int length);
unsigned char   *CMSG_DATA(struct cmsghdr *cmsg);
```

cmsghdr structure
The c*msghdr* structure is declared in <sys.socket.h> as follows:

```
struct cmsghdr {
        u_int   cmsg_len;      /* number of data bytes incl. header */
        int     cmsg_level;    /* generating protocol */
        int     cmsg_type;     /* protocol-specific type */
        /* followed by u_char cmsg_data[] */
};
struct cmsghdr *cmsg;
```

The sequence of *cmsghdr* structures should not be accessed directly. Instead, it is recommended to use the following macros.

CMSG_FIRSTHDR(struct msghdr *msg)

> returns a pointer to the first *cmsghdr* associated with the passed *msghdr*. It returns NULL if there is not enough space in the buffer.

CMSG_NXTHDR(struct msghdr *msg, struct cmsghdr *cmsg)

> returns the next valid *cmsghdr* after the passed *cmsghdr*. It returns NULL if there is not enough space in the buffer. CMSG_NEXTHDR can be used instead of CMSG_FIRSTHDR, wenn 0 is passed instead of a valid *cmsghdr* structure.

CMSG_ALIGN(int length)

> given a length, returns it including the required alignment.

CMSG_SPACE(int length)

returns the total length of the *cmsghdr* under the assumption that the data portion of the *cmsghdr* requires space equal to *length*. The return value is only affected by *length*. The macro also takes account for potential padding and buffer space between two *cmsghdr*.

CMSG_DATA(struct cmsghdr *cmsg)

returns a pointer to the data area. The pointer returned cannot be assumend to be suitably alligned for accessing arbitrary payload data types. Applications should use *memcpy()* to copy data to or from a suitably declared object.

CMSG_LEN(int length)

returns the value to store in *cmsg->cmsg_len*. The length matches the size of the complete cmsghdr under the assumption that the data portion of the *cmsghdr* requires space equal to *length*. The macro takes account for potential padding between two *cmsghdr*, but ignores potential buffer space.

Client/server model with SOCKETS(BS2000)

The client/server model is the most commonly used model for developing distributed applications. In the client /server model, client applications request services from a server. The present chapter uses examples to describe the interaction between the client and server in more detail and also illustrates some problems which may occur when developing client/server applications, together with their solutions.

Before a service can be granted and accepted, the communication between client and server needs a set of agreements known to both ends. These agreements are defined in a protocol that must be implemented on both ends of a connection. The protocol can be symmetric or asymmetric, depending on the conditions. In a symmetric protocol, both ends can take on the role of either server or client. With an asymmetric protocol, one end is fixed as the server and the other end as the client.

Regardless of whether a symmetric or asymmetric protocol is used for a service, whenever a service is accessed, there is always a client and a server.

The following are described in the sections below:

- Connection-oriented server
- Connection-oriented client
- Connectionless server
- Connectionless client

Connection-oriented server

The server normally waits on a known address for service requests. The server remains inactive until a client sends a connection request to the address of the server. The server then “wakes up” and serves the client by executing the relevant actions for the client request.
The server is accessed via a known Internet address.

You will find an example of a connection-oriented server for both AF_INET and AF_INET6, and for AF_ISO, below.

Connection-oriented server for AF_INET / AF_INET6

Programming of the main program loop is shown in the following examples.
The server uses the following socket interface functions in the example programs:

- *socket()*: create socket
- *bind()*: assign a socket a name
- *listen()*: “listen” to a socket for connection requests
- *accept()*: accept a connection on a socket
- *recv()*: read data from a socket
- *soc_close()*: close socket

Example: Connection-oriented server for AF_INET

```
#include <stdio.h>
#include <sys.types.h>
#include <sys.socket.h>
#include <netinet.in.h>
#include <netdb.h>

main(argc, argv)
    int argc;
    char *argv[];
{
#define TESTPORT 2222

    int sock, length;
    struct sockaddr_in server;
    struct sockaddr_in client;
    int clientlen;
    int msgsock;
    char buf[1024];
int rval;
    memset(&server,'\0',sizeof(server));
    memset(&client,'\0',sizeof(client));
    clientlen = sizeof(client);

/* Create socket */
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0)
       { perror("Create stream socket");
         exit(1);
       }

    /* Assign the socket a name */
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = htonl(INADDR_ANY);
    server.sin_port = htons(TESTPORT);

if (bind(sock, (struct sockaddr *)&server, sizeof (server) ) < 0)
       { perror("Bind stream socket");
         exit(1);
       }
```

```
/* Start acceptance of connection requests */
    listen(sock, 5);

msgsock = accept(sock, (struct sockaddr *)&client, &clientlen);
if (msgsock == -1)
       { perror("Accept connection");
         exit(1);
       }
    else do {
            memset(buf, 0, sizeof buf);
            if ((rval = recv(msgsock, buf, 1024, 0)) < 0)
               { perror("Reading stream message");
                 exit(1);
               }
             if (rval == 0 )
                 fprintf(stderr, "Ending connection\n");
             else
                 fprintf(stdout, "->%s\n", buf);
            } while (rval != 0);

  soc_close(msgsock);
  soc_close(sock);
}
```

Example: Connection-oriented server for AF_INET6

```
#include <stdio.h>
#include <sys.types.h>
#include <sys.socket.h>
#include <netinet.in.h>
#include <netdb.h>

main(argc, argv)
     int argc;
     char *argv[];
{
     #define TESTPORT 2222
     int sock, length;
     struct sockaddr_in6 server;
     struct sockaddr_in6 client;
     int clientlen;
     struct in6_addr in6addr_any = IN6ADDR_ANY_INIT;
     int msgsock;
     char buf[1024];
     int rval;
     memset(&server,'\0',sizeof(server));
     memset(&client,'\0',sizeof(client));
     clientlen = sizeof(client);

     /* Create socket */
     sock = socket(AF_INET6, SOCK_STREAM, 0);
     if (sock < 0)
        {
        perror("Create stream socket");
        exit(1);
        }
```

```
    /* Assign the socket a name */

    server.sin6_family = AF_INET6;
    memcpy(server.sin6_addr.s6_addr, in6addr_any,16) ;
    server.sin6_port = htons(TESTPORT);
    if (bind(sock, (struct sockaddr *)&server, sizeof (server) ) < 0)
       {
       perror("Bind stream socket");
       exit(1);
       }

    /* Start acceptance of connection requests */

    listen(sock, 5);
    msgsock = accept(sock, (struct sockaddr *)&client, &clientlen);

    if (msgsock == -1)
       {
       perror("Accept connection");
       exit(1);
       }

    else do
       {
       memset(buf, 0, sizeof buf);
       if ((rval = recv(msgsock, buf, 1024, 0)) < 0)
          {
          perror("Reading stream message");
          exit(1);
          }
       else if (rval == 0 )
          fprintf(stderr, "Ending connection\n");
       else
          fprintf(stdout, "->%s\n", buf);
       }

    while (rval != 0);
    soc_close(msgsock);
    soc_close(sock);
}
```

The following steps are executed in the program examples for AF_INET and AF_INET6:

1. The server uses the *socket()* function to create a communications endpoint (socket) and the corresponding descriptor.
2. The server socket is assigned a defined port number with the *bind()* function. It can then be addressed in the network via this port number.
3. The server uses the *listen()* function to determine whether connection requests are pending.
4. The server can accept connection requests with *accept()*. The value returned by *accept()* is tested to ensure that the connection was successfully set up.
5. As soon as the connection is set up, data is read from the socket with the *recv()* function.
6. The server closes the socket with the *soc_close()* function.

Connection-oriented server for AF_ISO

The server uses the following socket interface functions in the example program:

- *getbcamhost()*: get the host name entry
- *socket()*: create socket
- *bind()*: assign a socket a name
- *listen()*: “listen” to a socket for connection requests
- *accept()*: accept a connection on a socket
- *sendmsg()*: send a message from socket to socket / confirm connection
- *recv()*: read data from a socket
- *soc_close()*: close socket

Example: Connection-oriented server for AF_ISO

```
/*
 * Example: ISO SERVER
 *
 * DESCRIPTION
 *   1. getbcamhost - socket - bind - listen - accept - sendmsg
 *   2. recv
 *   3. soc_close
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <iso.h>
#include <netinet/in.h>
#include <netdb.h>

#define INT 5
#define MAXREC 1000000
#define MAXTSEL 32
#define MAXNSEL 9

main(argc, argv)
int argc;
char *argv[];
{
  void error_exit();
  int sockfd, newfd, clilen, ret;
  int tsellen, nsellen;
  char tsel[MAXTSEL];
  char nsel[MAXNSEL];
  char buffer [MAXREC];
  struct sockaddr_iso cli_addr, serv_addr;
  struct msghdr message;
  struct cmsghdr cmessage;

  strcpy (tsel,"SERVER");
  tsellen = strlen(tsel);
  nsel[8] = '\0';
```

```
  /* Get BCAM host name */
  errno = 0;
  if(getbcamhost(nsel,sizeof(nsel)) < 0)
     error_exit("ISO_svr: getbcamhost failed ",errno);
  else
     printf ("getbcamhost OK! (%s)\n",nsel);
  nsellen = strlen(nsel);

  /* Create socket*/
  errno = 0;
  if((sockfd = socket(AF_ISO, SOCK_STREAM, 0)) < 0)
     error_exit("ISO_svr: Socket Creation failed ",errno);
  else
     printf("socket OK!\n");

  /* Assign the socket a name */
  memset ((char *) &serv_addr, 0, sizeof(serv_addr));
  serv_addr.siso_len = sizeof (struct sockaddr_iso);
  serv_addr.siso_family = AF_ISO;
  serv_addr.siso_plen = 0;
  serv_addr.siso_slen = 0;
  serv_addr.siso_tlen = tsellen;
  serv_addr.siso_addr.isoa_len = tsellen + nsellen;
  memcpy (serv_addr.siso_addr.isoa_genaddr,nsel,nsellen);
  memcpy (serv_addr.siso_addr.isoa_genaddr + nsellen,tsel,tsellen);

  errno = 0;
if(bind(sockfd, (struct sockaddr_iso *) &serv_addr, sizeof(serv_addr)) < 0)
     error_exit("ISO_svr: Bind failed ",errno);
  else
     printf("bind OK!\n");

/* Start acceptance of connection requests */
errno = 0;
if (listen(sockfd, INT) < 0)
   error_exit("ISO_svr: Listen failed ",errno);
else
   printf("listen OK!\n");

errno = 0;
clilen = sizeof(cli_addr);
newfd = accept(sockfd, (struct sockaddr_iso *) &cli_addr, &clilen);
if(newfd < 0)
   error_exit("ISO_svr: New Socket Creation failed",errno);
else
   printf("accept OK!\n");

/* Confirm connection request (CONNECTION CONFIRM)
   No actual transfer of data takes place */
memset ((char *) &message, 0, sizeof(message));
memset ((char *) &cmessage, 0, sizeof(cmessage));
message.msg_control = (char *) &cmessage;
message.msg_controllen = sizeof (struct cmsghdr);
cmessage.cmsg_level = SOL_TRANSPORT;
cmessage.cmsg_type = TPOPT_CFRM_DATA;
cmessage.cmsg_len = sizeof (struct cmsghdr);
```

```
errno = 0;
ret = sendmsg (newfd, (struct msghdr *) &message, 0);
if (ret == -1)
   error_exit("ISO_svr: Sendmsg in Error", errno);
else
   printf("sendmsg OK!(%d)\n",ret);

/* Read data from a socket */
if ((ret = recv (newfd, buffer, MAXREC, 0)) < 0)
{
   if (errno != EPIPE)  /* Broken Pipe */
     error_exit("ISO_svr: Receive in Error", errno);
}
else
   printf("recv OK!(%d)\n",ret);

/* Close socket */
errno = 0;
if (soc_close (newfd) < 0)
  error_exit("ISO_svr: soc_close failed ",errno);
else
   printf("soc_close (newfd) OK!\n");
  if (soc_close (sockfd) < 0)
    error_exit("ISO_svr: soc_close failed ",errno);
  else
     printf("soc_close (sockfd) OK!\n");

} /* END MAIN */


void
error_exit(estring,erno)
  char *estring;
  int erno;
{
  fprintf(stderr,"%s errno=%d\n",estring,erno);
  perror (estring);
  exit(erno);
}
```

The server determines the BCAM host name with the *getbcamhost()* function.The following steps are executed in the program example for AF_ISO:

1. The server creates a communications endpoint (server socket) and the corresponding descriptor with the *socket()* function.
2. The server assigns the newly created socket a name with *bind()*.
3. The server (socket) is prepared for accepting connection requests with *listen()*.
4. The (server) socket accepts a connection request with *accept()*.
5. The server confirms the connection request (CFRM) with *sendmsg()*, i.e. the connection has now been set up. *sendmsg()* does not transfer any user data.
6. The server socket receives user data from the partner socket (client socket) with *recv()*.
7. The (server) socket is closed with the function *soc_close()*.

Connection-oriented client

You will find an example of a connection-oriented client for AF_INET, AF_INET6 and AF_ISO below.

Connection-oriented client for AF_INET / AF_INET6

The client side was shown in the example in section "Connection-oriented server for AF_INET / AF_INET6". You can clearly see the separate, asymmetric roles of the client and server in the program code. The server waits as a passive instance for connection requests from the client, whereas the client initiates a connection as the active instance.

The steps executed by the *remote login* client process are looked at more closely in the following sections. In the example programs, the client uses the following socket interface functions:

- *socket()*: create socket
- *gethostbyname()* / *getipnodebyname()*: get the host name entry
- *connect()*: request a connection on the socket
- *send()*: write data to the socket
- *soc_close()*: close socket

Example: Connection-oriented client for AF_INET

```
#include <stdio.h>
#include <sys.types.h>
#include <sys.socket.h>
#include <netinet.in.h>
#include <netdb.h>
#include <sys.uio.h>

main(argc, argv)
    int argc;
    char *argv[];
{
#define TESTPORT 2222
#define DATA "Here's the message ..."

    int sock, length;
    struct sockaddr_in client;
    struct hostent *hp;
    char buf[1024];
/*  Create socket */
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0)
       { perror("Create stream socket");
         exit(1);
       }

/* Fill in the address structure */
    client.sin_family = AF_INET;
    client.sin_port = htons(TESTPORT);
    hp = gethostbyname(argv[1]);
    if (hp == NULL)
       { fprintf(stderr,"%s: unknown host\n", argv[1]);
         exit(1);
       }
    memcpy((char *) &server.sin_addr, (char *)hp->h_addr,
                hp->h_length);

    /* Start the connection */
    if ( connect(sock, &server, sizeof(client) ) < 0 )
       { perror("Connect stream socket");
         exit(1);
       }

    /* Write to the socket */
    if ( send(sock, DATA, sizeof DATA, 0) < 0)
       { perror("Write on stream socket");
         exit(1);
       }

     soc_close(sock);
}
```

Example: Connection-oriented client for AF_INET6

```
#include <stdio.h>
#include <sys.types.h>
#include <sys.socket.h>
#include <netinet.in.h>
#include <netdb.h>
#include <sys.uio.h>

main(argc, argv)
    int argc;
    char *argv[];
{
    #define TESTPORT 2222
    #define DATA "Here's the message ..."
    int sock, length;
    int error_num;
    struct sockaddr_in6 client;
    struct hostent *hp;
    char buf[1024];

    /* Create socket */
    sock = socket(AF_INET6, SOCK_STREAM, 0);
    if (sock < 0)
       {
       perror("Create stream socket");
       exit(1);
       }

    /* Fill in the address structure */

    client.sin6_family = AF_INET6;
    client.sin6_port = htons(TESTPORT);
    hp = getipnodebyname(argv[1], AF_INET6, 0, &error_num);
    if ((hp == NULL) || (error_num != NETDB_SUCCESS))
       {
       fprintf(stderr,"%s: unknown host\n", argv[1]);
       exit(1);
       }
    memcpy((char *) &CLIENT.sin6_addr, (char *)hp->h_addr,
    hp->h_length);

    /* Release the dynamic memory of hostent */
    freehostent (hp);
    /* Start connection */

    if ( connect(sock, &client, sizeof(client) ) < 0 )
       {
       perror("Connect stream socket");
       exit(1);
       }

    /* Write to the socket */
    if ( send(sock, DATA, sizeof DATA, 0) < 0)
       {
       perror("Write on stream socket");
       exit(1);
       }
    soc_close(sock);
}
```

The client creates a communications endpoint (socket) and the corresponding descriptor with the *socket()* function. The following steps are executed in the program examples for AF_INET and AF_INET6:

1. The client queries the address of the host with *gethostbyname()* (only for AF_INET).The host name is passed as a parameter.
   The client determines the IPv6 address of the host name passed as a parameter with *getipnodebyname()*. This new function could also be used for the AF_INET example. A connection must then be set up to the server for the desired host. The client initializes the address structure for this purpose.
2. The connection is set up with *connect()*.
3. After connection setup, data is written to the socket with the *send()* function.
4. The created socket is closed with the *soc_close()* function.

Connection-oriented client for AF_ISO

In the example program, the client uses the following socket interface functions:

- *getbcamhost()*: get BCAM host name
- *socket()*: create socket
- *bind()*: assign a name to the socket
- *connect()*: request a connection on the socket
- *send()*: write data to the socket
- *soc_close()*: close socket

Example: Connection-oriented client for AF_ISO

```
/*
 * Example: ISO CLIENT
 *
 * DESCRIPTION
 *   1. getbcamhost - socket - bind - connect
 *   2. send
 *   3. soc_close
*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <iso.h>
#include <netinet/in.h>
#include <netdb.h>

#define INT 5
#define MAXREC 1000000
#define MAXTSEL 32
#define MAXNSEL 9

main(argc, argv)
int argc;
char *argv[];
{
  void error_exit();
  int sockfd, ret, lng;
  int tsellen, nsellen, par_tsellen, par_nsellen;
  char tsel[MAXTSEL];
  char par_tsel[MAXTSEL];
  char nsel[MAXNSEL];
  char par_nsel[MAXNSEL];
  char buffer [MAXREC];
  struct sockaddr_iso cli_addr, serv_addr;
lng = 1024 ;
strcpy (tsel,"CLIENT");
tsellen = strlen(tsel);
strcpy (par_tsel,"SERVER");
par_tsellen = strlen(par_tsel);
nsel[MAXNSEL-1] = '\0';

/* Get partner host name */
```

```
if (argc > 1)
{
  strcpy (par_nsel,argv[1]);
  if ((par_nsellen = strlen(par_nsel)) != MAXNSEL - 1)
  {
     printf ("Error: Invalid host name !!\n");
     exit (-1);
  }
}
else
{
   printf ("Partner host name was not passed as an argument in the
            command line !\n");
   exit (-1);
}

/* Get BCAM host name*/
errno = 0;
if (getbcamhost(nsel,sizeof(nsel)) < 0)
    error_exit("ISO_cli: getbcamhost failed ",errno);
else
    printf ("getbcamhost OK! (%s)\n",nsel);
nsellen = strlen(nsel);

/* Create socket*/
errno = 0;
if ((sockfd = socket(AF_ISO, SOCK_STREAM, 0)) < 0)
    error_exit("ISO_cli: Socket Creation failed ",errno);
else
    printf ("socket OK!\n");

/* Assign a name to the socket */
memset ((char *) &cli_addr, 0, sizeof(cli_addr));
cli_addr.siso_len = sizeof (struct sockaddr_iso);
cli_addr.siso_family = AF_ISO;
cli_addr.siso_plen = 0;
cli_addr.siso_slen = 0;
cli_addr.siso_tlen = tsellen;
cli_addr.siso_addr.isoa_len = tsellen + nsellen;
  memcpy (cli_addr.siso_addr.isoa_genaddr,nsel,nsellen);
  memcpy (cli_addr.siso_addr.isoa_genaddr + nsellen,tsel,tsellen);

  memset ((char *) &serv_addr, 0, sizeof(serv_addr));
  serv_addr.siso_len = sizeof (struct sockaddr_iso);
  serv_addr.siso_family = AF_ISO;
  serv_addr.siso_plen = 0;
  serv_addr.siso_slen = 0;
  serv_addr.siso_tlen = par_tsellen;
  serv_addr.siso_addr.isoa_len = par_tsellen + par_nsellen;
  memcpy (serv_addr.siso_addr.isoa_genaddr,par_nsel,par_nsellen);
  memcpy (serv_addr.siso_addr.isoa_genaddr +
              par_nsellen,par_tsel,par_tsellen);

  errno = 0;
  if (bind (sockfd, (struct sockaddr_iso *) &cli_addr, sizeof(cli_addr)) < 0)
      error_exit("ISO_cli: Bind failed ",errno);
  else
      printf ("bind OK!\n");
```

```
  /* Start connection */
  errno = 0;
  if (connect (sockfd, (struct sockaddr_iso *) &serv_addr,
              sizeof(serv_addr)) < 0)
      error_exit("ISO_cli: Connect failed ",errno);
  else
      printf ("connect OK!\n");

  sleep(2);

  /* Write data to the socket */
  ret = send (sockfd, buffer, lng, 0);
  if (ret == -1)
      error_exit("ISO_cli: Send in Error", errno);
  else
      printf ("send OK!(%d)\n",ret);

  /* Close socket*/
  sleep (2);
  errno = 0;
  if (soc_close (sockfd) <0)
      error_exit("Tcp_svr: soc_close failed ",errno);
  else
      printf ("soc_close OK!\n");

} /* END MAIN */
void
error_exit(estring,erno)
  char *estring;
  int erno;
{
  fprintf(stderr,"%s errno=%d\n",estring,erno);
  perror (estring);
  exit(erno);
}
```

The client takes the name of the partner host from the command line argument *argc* of the *main()* function.The following steps are executed in the program:

1. The client determines the BCAM host name with the function *getbcamhost()*.
2. The client creates a communications endpoint (client socket) and the corresponding descriptor with the function *socket()*.
3. The client assigns a name to the newly created socket with *bind()*.
4. The client sets up the connection to the communications partner (server socket) with *connect()*.
5. The client send user data to the partner socket (server socket) with *send()*.
6. The function *soc_close()* closes the (client) socket.

Connectionless server

Most servers operate on a connection-oriented basis, but some services are based on using datagram sockets and are thus connectionless.

The following socket interface functions are used by the server in the example programs:

- *socket()*: create socket
- *bind()*: assign a socket a name
- *recvfrom()*: read a message from a socket
- *soc_close()*: close socket

The program is shown in two variants:

- In the first variant (examples 1 and 3), the program is terminated when a message arrives (*read()*).
- In the second variant (examples 2 and 4), the program waits in an endless loop for further messages after a message has been read.

Example 1: Connectionless server without a program loop for AF_INET

```
#include <stdio.h>
#include <sys.types.h>
#include <sys.socket.h>
#include <ioctl.h>
#include <signal.h>
#include <netinet.in.h>
#include <netdb.h>

#define TESTPORT 2222

/*
 * This program creates a datagram socket, assigns it a defined
 * port and then reads data from the socket.
 */

main()
{
   int sock;
   int length;
    struct sockaddr_in server;
    char buf[1024];
/* Create the socket to be read from. */
    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock < 0 )
       { perror("Socket datagram");
         exit(1);
       }

    /* Assign the server "server" a name, using wildcards
    */
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = htonl(INADDR_ANY);
    server.sin_port = htons(TESTPORT);

    if (bind(sock, &server, sizeof server ) < 0)
       { perror("Bind datagram socket");
         exit(1);
       }

    /* Start reading from the server */
    length = sizeof(server);
    memset(buf,0,sizeof(buf));
    if ( recvfrom(sock, buf, 1024,0, &server, &length) < 0 )
           { perror("recvfrom");
             exit(1);
           }
    else
          printf("->%s\n",buf);

    soc_close(sock);
}
```

Example 2: Connectionless server with a program loop for AF_INET

```
#include <sys.types.h>
#include <sys.socket.h>
#include <ioctl.h>
#include <signal.h>
#include <netinet.in.h>
#include <netdb.h>
#include <stdio.h>
#define TESTPORT 2222

/* This program creates a datagram socket, assigns it a defined
 * port and then reads data from the socket. */
main()
{
    int sock;
    int length;
    struct sockaddr_in server;
    char buf[1024];

    /* Create the socket to be read from. */
    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock < 0 )
       { perror("Socket datagram");
         exit(1);
       }

    /* Assign the server "server" a name using wildcards */
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = htonl(INADDR_ANY);
    server.sin_port = htons(TESTPORT);

    if (bind(sock, &server, sizeof server ) < 0)
       { perror("Bind datagram socket");
         exit(1);
       }

    /* Start reading from the server */
    length = sizeof(server);
    for (;;)
    {

        memset(buf,0,sizeof(buf));
        if ( recvfrom(sock, buf, sizeof(buf),0, &server, &length) < 0 )
              {  perror("recvfrom");
                 exit(1);
              }
else
              printf("->%s\n",buf);
    }

    /* Since this program runs in an endless loop, the socket
     * "sock" is never explicitly closed. However, all sockets
     * are closed automatically if the process is aborted.
     */
}
```

Example 3: Connectionless server without a program loop for AF_INET6

```
#include <stdio.h>
#include <sys.types.h>
#include <sys.socket.h>
#include <ioctl.h>
#include <signal.h>
#include <netinet.in.h>
#include <netdb.h>
#define TESTPORT 2222

/*
* This program creates a datagram socket, assigns it a defined
* port and then reads data from the socket.
*/

main()
{
    int sock;
    int length;
    struct sockaddr_in6 server;
    struct in6_addr in6addr_any = IN6ADDR_ANY_INIT;
    char buf[1024];
    /* Create the socket to be read from. */
    sock = socket(AF_INET6, SOCK_DGRAM, 0);
    if (sock < 0 )
       { perror("Socket datagram");
         exit(1);
       }

    /* Assign the server "server" a name using wildcards */
    server.sin6_family = AF_INET6;
    memcpy(server.sin6_addr.s6_addr, in6addr_any.s6_addr, 16) ;
    server.sin6_port = htons(TESTPORT);
    if (bind(sock, &server, sizeof server ) < 0)
       { perror("Bind datagram socket");
         exit(1);
       }

    /* Start reading from the server */
    length = sizeof(server);
    memset(buf,0,sizeof(buf));
    if ( recvfrom(sock, buf, 1024,0, &server, &length) < 0 )
       { perror("recvfrom");
         exit(1);
       }
    else
       printf("->%s\n",buf);
    soc_close(sock);
}
```

Example 4: Connectionless server with a program loop for AF_INET6

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

```
#include <ioctl.h>
#include <signal.h>
#include <netinet.in.h>
#include <netdb.h>
#include <stdio.h>

#define TESTPORT 2222

/* This program creates a datagram socket, assigns it a defined
 * port and then reads data from the socket. */

main()
{
    int sock;
    int length;
    struct sockaddr_in6 server;
    struct in6_addr in6addr_any = IN6ADDR_ANY_INIT;
    char buf[1024];

    /* Create the socket to be read from. */
    sock = socket(AF_INET6, SOCK_DGRAM, 0);
    if (sock < 0 )
       {
         perror("Socket datagram");
         exit(1);
       }

    /* Assign the server "server" a name using wildcards */

    server.sin6_family = AF_INET6;
    memcpy(server.sin6_addr.s6_addr ,in6addr_any.s6_addr,16);
    server.sin6_port = htons(TESTPORT);
    if (bind(sock, &server, sizeof server ) < 0)

         perror("Bind datagram socket");
         exit(1);
       }

    /* Start reading from the server */

    length = sizeof(server);
    for (;;)
       {
         memset(buf,0,sizeof(buf));
         if ( recvfrom(sock, buf, sizeof(buf),0, &server, &length) < 0 )
           {
             perror("recvfrom");
             exit(1);
         else
             printf("->%s\n",buf);
       }

    /* Since this program runs in an endless loop, the socket
     * "sock" is never explicitly closed. However, all sockets
     * are closed automatically if the process is aborted.
     */
}
```

The following steps are executed in the program examples for AF_INET and AF_INET6:

1. The server creates a communications endpoint (socket) and corresponding descriptor with the *socket()* function.
2. The server socket is assigned a defined port number with the *bind()* function so that it can be addressed from the network via this port number.
3. The *recvfrom()* function can be used to read from a socket of type SOCK_DGRAM.
4. The length of the read message is returned as the result. If no message is available, the process is blocked until a message arrives.

Connectionless client

The following socket interface functions are used by the client in these program examples:

- *socket()*: create socket
- *gethostbyname()* / *getipnodebyname()*: get the host name entry
- *sendto()*: send a message to a socket
- *soc_close()*: close socket

Example: Connectionless client for AF_INET

```
#include <stdio.h>
#include <sys.types.h>
#include <sys.socket.h>
#include <ioctl.h>
#include <signal.h>
#include <netinet.in.h>
#include <netdb.h>

#define DATA " The sea is calm, the tide is full ..."
#define TESTPORT 2222

/*
 * This program sends a datagram to a receiver whose name is passed
 * as an argument in the command line.
 */
main(argc,argv)
    int argc;
    char *argv[];
{
    int sock;
    struct sockaddr_in to;
    struct hostent *hp, *gethostbyname();

    /* Create the socket to be sent on. */
    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock < 0 )
       { perror("Socket datagram");
         exit(1);
       }

    /* Construct the name of the socket to be sent on, without using
     * wildcards. gethostbyname returns a structure which contains the
     * network address of the specified host.
     * The port number is taken from the TESTPORT constant.
     */
hp =gethostbyname(argv[1]);
    if (hp == 0) {
        fprintf(stderr, "%s:unknown host\n", argv[1]);
        exit(1);
    }
     memcpy((char *)&to.sin_addr, (char *)hp->h_addr,hp->h_length);
    to.sin_family = AF_INET;
    to.sin_port = htons(TESTPORT);

    /* Send message. */
    if (sendto(sock, DATA, sizeof DATA, 0, &to, sizeof to) < 0) {
         perror("Sending datagram message");
         exit(1);
    }
    soc_close(sock);
}
```

Example: Connectionless client for AF_INET6

```
#include <stdio.h>
#include <sys.types.h>
#include <sys.socket.h>
#include <ioctl.h>
#include <signal.h>
#include <netinet.in.h>
#include <netdb.h>
#define DATA " The sea is calm, the tide is full ..."
#define TESTPORT 2222

/*
* This program sends a datagram to a receiver whose name is passed
* as an argument in the command line. */
main(argc,argv)
    int argc;
    char *argv[];
{
    int sock;
    int error_num;
    struct sockaddr_in6 to;
    struct hostent *hp;

    /* Create the socket to be sent on. */
    sock = socket(AF_INET6, SOCK_DGRAM, 0);
    if (sock < 0 )
       {
         perror("Socket datagram");
         exit(1);
       }

    /* Construct the name of the socket to be sent on, without using
    * wildcards. gethostbyname returns a structure which contains the
    * network address of the specified host.
    * The port number is taken from the TESTPORT constant.
    */
    hp =getipnodebyname(argv[1], AF_INET6, 0, &error_num);
    if ((hp = = 0) || (error_num != NETDB_SUCCESS))
       {
         fprintf(stderr, "%s:unknown host\n", argv[1]);
         exit(1);
       }
    memcpy((char *)&to.sin6_addr, (char *)hp->h_addr,hp->h_length);
    to.sin6_family = AF_INET6;
    to.sin6_port = htons(TESTPORT);

    /* Release the dynamic memory of hostent */
    freehostent(hp);

    /* Send message. */
    if (sendto(sock, DATA, sizeof DATA, 0, &to, sizeof to) < 0)
       {
         perror("Sending datagram message");
         exit(1);
       }
    soc_close(sock);
}
```

The client creates a communications endpoint (socket) and corresponding descriptor with *socket()*.The following steps are executed in the program examples for AF_INET and AF_INET6:

1. The client queries the address of the host with *gethostbyname()* (for AF_INET); the host name is passed as a parameter.
   The client queries the IPv6 address of the host name passed as a parameter with *getipnodebyname()*. This new function could also be used for the AF_INET example. The address structure is then initialized.
2. The client sends a datagram with *sendto()*, which returns the number of transferred characters.
3. The client closes the socket with *soc_close()*.

SOCKETS(BS2000) user functions

This chapter describes the socket interface functions for BS2000. The first section describes the format in which the individual functions are explained. The subsequent overview collects functions together into task-oriented groups. Finally, all socket interface functions are described in alphabetical order.

Description format

The SOCKETS(BS2000) user functions are described in a uniform format. The function descriptions have the same format as shown in section "Function name - brief description of the functionality".

> Ensure the type conversion (Cast) is correct in ANSI mode to prevent compiler warnings. This applies in particular to the different socket structures (*sockaddr*, *sockaddr_in*, *sockaddr_in6*, *sockaddr_iso*):
>
> To call a function use the general *sockaddr* structure.

Use the specific structure of the relevant address family to enter and read socket addresses.

Function name - brief description of the functionality

```
#include < ... >
#include < ... >
      ...
Function syntax
```

Description

Detailed description of the functionality and parameters.

Return value

List and description of all possible function return values.
Some functions have no return value. The “Return value” section is omitted in such cases and in the descriptions of external variables.

Errors indicated by *errno*

List and description of the error codes in the external variable *errno* that can occur with an invalid call or function. This section may be omitted.

Note

Description of terms or information on interaction with other functions, or tips for use. This section may be omitted.

See also

Cross references to function descriptions. This section may be omitted.

Overview of functions

The following overview of the SOCKETS(BS2000) interface functions collects several functions together into task-oriented groups.

The three columns on the right, “INET”, “INET6”, and “ISO”, indicate the address family (AF_INET, AF_INET6, AF_ISO) in which the function involved is supported.

Setting up and shutting down connections on sockets

Function	Description	See	INET	INET6	ISO
socket()	Create socket	"socket() - create socket"	x	x	x
bind()	Assign a name to a socket	"bind() - assign a socket a name"	x	x	x
connect()	Initiate communication on a socket (e.g. by a client)	"connect() - initiate a connection on a socket"	x	x	x
listen()	Test socket for pending connections (e.g. by a server)	"listen() - test a socket for pending connections"	x	x	x
accept()	Accept connection on a socket (e.g. by a server)	"accept() - accept a connection on a socket"	x	x	x
shutdown()	Shut down connection in read and/or write direction	"shutdown() - terminate full-duplex connection"	x	x	
soc_close()	Close socket	"soc_close() (close) - close socket"	x	x	x

Transferring data between two sockets

Function	Description	See	INET	INET6	ISO
soc_read(), soc_readv()	Receive a message from a socket via an established connection	"soc_read(), soc_readv() (read, readv) - receive a message from a socket"	x	x	x
recv()	Receive a message from a socket via an established connection	"recv(), recvfrom() - receive a message from a socket"	x	x	x
recvfrom()	Receive a message from a socket	"recv(), recvfrom() - receive a message from a socket"	x	x	x
recvmsg()	Receive a message from a socket. AF_ISO: Receive messages (user data or connection data) via an established connection	"recvmsg() - receive a message from a socket"	x	x	x
send()	Send a message from socket to socket via an established connection	"send(), sendto() - send a message from socket to socket"	x	x	x

sendto()	Send a message from socket to socket	"send(), sendto() - send a message from socket to socket"	x	x	x
sendmsg()	Send a message from socket to socket. AF_ISO: Send messages (user data or connection data) via an established connection	"sendmsg() - send a message from socket to socket"	x	x	x
soc_write(), soc_writev()	Send a message from socket to socket via an established connection	"soc_write(), soc_writev() (write, writev) - send a message from socket to socket"	x	x	x
select()	Multiplex input/output	"select() - multiplex input/output"	x	x	x
soc_poll()	Multiplex input/output	"soc_poll() - multiplex input/output"	x	x	x

Transmitting data from/to the socket buffer

Function	Description	See	INET	INET6	ISO
soc_getc()	Get character from socket buffer	"soc_getc() (getc) - get character from input buffer"	x	x	
soc_gets()	Get string from socket buffer	"soc_gets() (gets) - get string from input buffer"	x	x	
soc_putc()	Put character in socket buffer	"soc_putc() (putc) - put character in output buffer"	x	x	
soc_puts()	Put string in socket buffer	"soc_puts() (puts) - put string in output buffer"	x	x	
soc_flush()	Flush socket buffer	"soc_flush () (flush) - flush data from output buffer"	x	x	

Receiving and setting information about sockets

Function	Description	See	INET	INET6	ISO
getdtablesize()	Get size of descriptor table	"getdtablesize() - get size of descriptor table"	x	x	x
getsockopt()	Get socket options	"getsockopt(), setsockopt() - get and set socket options"	x	x	x
getpeername()	Get name of communications partner	"getpeername() - get the remote address of the socket connection"	x	x	x
getsockname()	Get name of socket	"getsockname() - get local address of the socket connection"	x	x	x
setsockopt()	Set socket options	"getsockopt(), setsockopt() - get and set socket options"	x	x	x

Testing configuration values

Function	Description	See	INET	INET6	ISO
getaddrinfo()	Get IP address and port number corresponding to a host and/or service name	"getaddrinfo() - get information about host names, host addresses and services regardless of protocol"	x	x	
gai_strerror()	Get description of a *getaddrinfo()* error code	"gai_strerror() - output text for the error code of getaddrinfo()"	x	x	
getbcamhost()	Get name of BCAM host	"getbcamhost() - get BCAM host name"			x
gethostname()	Get socket host name of current host	"gethostname() - get the name of the current host"	x	x	x
gethostbyaddr()	Get host name belonging to an IPv4 address	"gethostbyaddr(), gethostbyname() - get information about host names and addresses"	x		
gethostbyname()	Get IPv4 address belonging to a host name	"gethostbyaddr(), gethostbyname() - get information about host names and addresses"	x		
getipnodebyaddr()	Get host name belonging to an IPv4 or IPv6 address	"getipnodebyaddr(), getipnodebyname() - get information about host names and addresses"	x	x	
getipnodebyname()	Get IPv4 or IPv6 address belonging to a host name	"getipnodebyaddr(), getipnodebyname() - get information about host names and addresses"	x	x	

getnameinfo()	Get host and service name corresponding to IP address and port number	"getnameinfo() - get the name of the communications partner"	x	x	
getservbyport()	Get name of a service	"getservbyname(), getservbyport() - get information about services"	x	x	
getservbyname()	Get port number of a service	"getservbyname(), getservbyport() - get information about services"	x	x	
getprotobyname()	Get number of a protocol	"getprotobyname() - get the number of the protocol"	x	x	
if_nameindex()	List with interface names and index of local host	"if_nameindex() - list of interface names with the associated interface indexes"	x	x	

Manipulating Internet address

Function	Description	See	INET	INET6	ISO
inet_addr()	Convert character string from dotted notation to integer value (Internet address)	"inet_addr(), inet_lnaof(), inet_makeaddr(), inet_netof(), inet_network(), inet_ntoa() - manipulate IPv4 Internet address"	x		
inet_network()	Convert character string from dotted notation to integer value (subnetwork section)	"inet_addr(), inet_lnaof(), inet_makeaddr(), inet_netof(), inet_network(), inet_ntoa() - manipulate IPv4 Internet address"	x		
inet_makeaddr()	Create Internet address from subnetwork section and subnetwork local address section	"inet_addr(), inet_lnaof(), inet_makeaddr(), inet_netof(), inet_network(), inet_ntoa() - manipulate IPv4 Internet address"	x		
inet_lnaof()	Extract local network address in byte order of host from Internet host address	"inet_addr(), inet_lnaof(), inet_makeaddr(), inet_netof(), inet_network(), inet_ntoa() - manipulate IPv4 Internet address"	x		
inet_netof()	Extract network number in byte order of host from Internet host address	"inet_addr(), inet_lnaof(), inet_makeaddr(), inet_netof(), inet_network(), inet_ntoa() - manipulate IPv4 Internet address"	x		
inet_ntoa()	Convert Internet host address into a string conforming to normal Internet dotted notation	"inet_addr(), inet_lnaof(), inet_makeaddr(), inet_netof(), inet_network(), inet_ntoa() - manipulate IPv4 Internet address"	x		

inet_pton()	Converts • an IPv4 address in decimal dotted notation or • and IPv6 address in hexadecimal colon notation to the corresponding binary address.	"inet_ntop(), inet_pton() - manipulate Internet addresses"	x	x	
inet_ntop()	Converts a binary IPv4 or IPv6 address to the corresponding • IPv4 address in decimal dotted notation or • IPv6 address in hexadecimal colon notation.	"inet_ntop(), inet_pton() - manipulate Internet addresses"	x	x	

Utility functions

Function	Description	See	INET	INET6	ISO
freeaddrinfo()	Release memory area for *addrinfo* structure requested by the *getaddrinfo()* function.	"freeaddrinfo() - release memory for addrinfo structure"	x	x	
freehostent()	Release memory area for *hostent* structure requested by the *getipnodebyaddr()* and *getipnodebyname()* functions.	"freehostent() - release memory for hostent structure"	x	x	
if_freenameindex()	Release memory area for array with *if_nameindex()* structure(s) requested by the *if_nameindex()* function.	"if_freenameindex() - release the dynamic storage occupied with if_nameindex()"	x	x	
htonl()	32 bit fields convert from host to network byte order	"Byte order macros - convert byte order"	x		
htons()	16 bit fields convert from host to network byte order	"Byte order macros - convert byte order"	x	x	
if_indextoname()	Determine name corresponding to the index	"if_indextoname() - convert interface index to interface name"	x	x	
if_nametoindex()	Determine name corresponding to the index	"if_nametoindex() - convert interface name to interface index"	x	x	
ntohl()	32 bit fields convert from network to host byte order	"Byte order macros - convert byte order"	x		
ntohs()	16 bit fields convert from network to host byte order	"Byte order macros - convert byte order"	x	x	

Control functions

Function	Description	See	INET	INET6	ISO
soc_ioctl()	Control sockets	"soc_ioctl() (ioctl) - control sockets"	x	x	x
soc_wake()	Awaken a task waiting with *select()* or *soc_poll()*	"soc_wake() - awaken a task waiting with select() or soc-poll()"	x	x	x

Test macros for AF_INET6

The following test macros are defined in <netinet.in.h>:

Macro	Test
IN6_IS_ADDR_UNSPECIFIED	address = 0 ?
IN6_IS_ADDR_LOOPBACK	address = loopback ?
IN6_IS_ADDR_LINKLOCAL	address = IPv6 - LINKLOCAL ?
IN6_IS_ADDR_SITELOCAL	address = IPv6 - SITELOCAL ?
IN6_ADDR_V4COMPAT	address = IPv4-compatible ?
IN6_ADDR_V4MAPPED	address = IPv4-mapped
IN6_ARE_ADDR_EQUAL	address1 = address2 ?

Description of functions

This section describes all user functions of the SOCKETS(BS2000) interface inalphabetical order.

accept() - accept a connection on a socket

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

#include <netinet.in.h> /* only for AF_INET and AF_INET 6 */
#include <iso.h> /* only for AF_ISO */

Kernighan-Ritchie-C:
int accept(s, addr, addrlen);

int s;
int *addrlen;

struct sockaddr_in *addr; /* only for AF_INET */
struct sockaddr_in6 *addr; /* only for AF_INET6 */
struct sockaddr_iso *addr; /* only for AF_ISO */

ANSI-C:
int accept(int s, struct sockaddr * addr, int* addrlen);
```

Description

The *accept()* function is used by the server task to accept a connection on socket *s*, as requested by the client with the *connect()* function.

In order to call *accept()* for socket *s*, the following requirements must be satisfied:

- *s* must be s stream socket (SOCK_STREAM) that has assigned a name (address) with *bind()*.
- *s* must be marked with *listen()*, i.e. identified as a socket on which connection requests can be accepted.

On returning from *accept()*, *addr* points to the address of the partner application, as known on the communications level. The exact format of **addr* (i.e. the address) is determined by the domain in which communication takes place.

- The address returned for the AF_INET address family is of type *struct sockaddr_in* (see section "sockaddr_in address structure of the AF_INET address family").
- The address returned for the AF_INET6 address family is of type *struct sockaddr_in6* (see section "sockaddr_in6 address structure of the AF_INET6 address family").
- The address returned for the AF_ISO address family is of type *struct sockaddr_iso* (see section "sockaddr_iso address structure for the AF_ISO address family").

addrlen points to an integer object that holds the size of the memory area referenced by **addr* (in bytes) at the time of the *accept()* call. When the *accept()* function returns, **addrlen* contains the length of the returned address in bytes.

When the queue set up by the *listen()* function contains at least one connection request, *accept()* proceeds as follows:

1. *accept()* selects the first connection from the connection requests in the queue.
2. *accept()* creates a new socket.
3. *accept()* returns the descriptor of the new socket as its result.

Two cases must be considered if there are no connection requests in the queue:

- If the socket is marked as blocking (standard case), *accept()* blocks the calling task until a connection is possible.
- If the socket is marked as non-blocking, *accept()* returns an error message with *errno* = EWOULDBLOCK.

You can call *select()* before calling *accept()* to test the read readiness of the socket concerned and make sure that the *accept()* call will not block.

Once *accept()* has executed successfully, the complete connection will have been set up in the AF_INET and AF_INET6 address families. In the AF_ISO address family, one of the two following steps is also required to set up a complete connection (see also figure 4 in section "Interaction between functions for connection-oriented communications"):

- send user data to the partner who requested the connection
- call *sendmsg()* (see section "sendmsg() - send a message from socket to socket") (with or without sending user data)

Once a connection has been set up successfully, data can be exchanged via the new socket created by *accept()* with the socket that requested the connection. Additional connections cannot be set up on the new socket. The original socket *s* remains open to accept further connections.

Return value

>=0:

If successful. The value is the descriptor for the accepted socket.

-1:

If errors occur. *errno* is set to indicate the error.

Errors indicated by *errno*

EBADF

s is not a valid descriptor.

EFAULT

The length of the range for accepting the address is too small.

EMFILE

The maximum number of open sockets has been reached.

ENETDOWN

The connection to the network is down.

EOPNOTSUPP

The referenced socket is not of type SOCK_STREAM or was not marked with *listen()* as a socket that can accept connection requests.

EWOULDBLOCK

The socket is marked as non-blocking, and no free connections are available.

See also

bind(), connect(), listen(), select(), socket()

bind() - assign a socket a name

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

#include <netinet.in.h> /* only for AF_INET and AF_INET6 */
#include <iso.h> /* only for AF_ISO */

Kernighan-Ritchie-C: int bind(s, name, namelen);
int s;
int namelen;

struct sockaddr_in *name;  /* only for AF_INET  */
struct sockaddr_in6 *name; /* only for AF_INET6 */
struct sockaddr_iso *name; /* only for AF_ISO   */

ANSI-C:
int bind(int s, struct sockaddr* name, int namelen);
```

Description

The *bind()* function assigns a name to a socket created with the *socket()* function that is initially nameless. After a socket has been created with the *socket()* function, the socket exists within a name area (address family) but it has no name.

The *s* parameter designates the socket to which a name is to be assigned with *bind()*.*namelen* specifies the length of the data structure which describes the name.

Return value

0:

If successful.

-1:

If errors occur. *errno* is set to indicate the error.

Errors indicated by *errno*

EADDRINUSE

The specified name is already in use.

EADDRNOTAVAIL

The specified name cannot be bound to the socket by the local system.

EBADF

s is not a valid descriptor.

EFAULT

The length of the area for accepting the address is too small.

EINVAL

The socket already has a name assigned to it or *namelen* does not have the size of a valid address for the specified address family.

ENETDOWN

The connection to the network is down.

See also

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

Byte order macros - convert byte order

```
#include <sys.types.h>
#include <netinet.in.h>

u_long htonl(u_long hostlong);

u_short htons(u_short hostshort);

u_long ntohl(u_long netlong);

u_short ntohs(u_short netshort);
```

Description

The *htonl()*, *htons()*, *ntohl()* and *ntohs()* macros are only required in the AF_INET and AF_INET6 address families. *htonl()*, *htons()*, *ntohl()* and *ntohs()* convert bytes and integers of the type *integer* or *short* from host byte order to network byte order and vice versa:

- *htonl()* converts 32 bit fields from host to network byte order.
- *htons()* converts 16 bit fields from host to network byte order.
- *ntohl()* converts 32 bit fields from network to host byte order.
- *ntohs()* converts 16 bit fields from network to host byte order.

These macros are mainly used in connection with IPv4 addresses and port numbers, e.g. as returned by the *gethostbyname()* function (see section "gethostbyaddr(), gethostbyname() - get information about host names and addresses").

With regard to IPv6 addresses, a decision was made according to RFC 2553 in favor of the guaranteed network byte order. Therefore only the 16-bit byte order macros are required for the port numbers for the AF_INET6 address family.

The macros are only needed on systems on which the host and network byte orders differ. Since the host and network byte orders are identical in BS2000, the macros are supplied as null macros (macros without a function) in the <netinet.in.h> header file.

Note, however, that the use of byte order macros is strongly recommended if you want to create portable programs.

Return value

htonl() and *htons()* return the input parameter after conversion into network byte order.

ntohl() and *ntohs()* return the input parameter after conversion into host byte order.

See also

gethostbyaddr(), gethostbyname(), getservbyname()

connect() - initiate a connection on a socket

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

#include <netinet.in.h> /* only for AF_INET and AF_INET6 */
#include <iso.h> /* only for AF_ISO */

Kernighan-Ritchie-C:
int connect(s, name, namelen);

int s;
int namelen;

struct sockaddr_in *name; /* only for AF_INET */
struct sockaddr_in6 *name; /* only for AF_INET6 */
struct sockaddr_iso *name; /* only for AF_ISO */

ANSI-C:
int connect(int s, struct sockaddr* name, int namelen);
```

Description

A task uses *connect()* to initiate communications with a partner socket via socket *s* of type SOCK_STREAM. If the partner socket is of type SOCK_DGRAM, the partner information is only saved in socket *s*.

The *s* parameter designates the socket on which the task initiates communications with another socket. *name* is a pointer to the address of the communications partner. The communications partner is a socket which belongs to the same address family. In the AF_ISO address family both sockets must belong to the same address family.

Communication in both directions is possible between the AF_INET and AF_INET6 address families with the help of IPv4-mapped IPv6 addresses, i.e. it is possible to establish a connection between an AF_INET socket on a host, which only has IPv4 addresses, and an AF_INET6 partner socket on a host, which exclusively or partly has IPv6 addresses.

**name* is an address in the address range of the socket to which the connection is to be initiated. Each address range interprets the *name* parameter in its own way. *namelen* contains the length of the address of the communications partner in bytes.

The exact functionality of *connect()* is determined by the address family used.

connect() for AF_INET and AF_INET6

The manner in which *connect()* proceeds differs according to whether the socket type is SOCK_STREAM or SOCK_DGRAM.

- With a socket of type SOCK_STREAM (stream socket), *connect()* sends a connection request to a partner and tries in this way to set up a connection to this partner. The partner is specified with the *name* parameter. For example, a client task uses *connect()* to initiate a connection to a server on a stream socket.
  Stream sockets can generally set up a connection with *connect()* only once.

- With a socket of type SOCK_DGRAM (datagram socket), a task uses *connect()* to define the name of the communications partner with which data is to be exchanged. The task then sends the datagrams to this communications partner. This communications partner is also the only socket from which the task can receive datagrams.

  If both an IP address and a port not equal to 0 are specified, the transport system generates a route to which it assigns a local interface. This local interface can be inquired using *getsockname()*.

  connect() can be used several times with datagram sockets to change the communications partner. The assignment to a specific partner can be terminated by entering a null pointer for the *name* parameter.

connect() for AF_ISO

connect() is used to set up the connection to an ISO partner. The partner must not only accept the connection request with *accept()*, but must also call a transfer function (*send()* or *sendmsg()*) as confirmation. However, no data need be sent with the transfer function. This can, for example, be done with a *sendmsg()* call.

Here again, the connection between two end points can only be set up once by *connect()*.

Return value

0:

If successful.

-1:

If errors occur. *errno* is set to indicate the error.

Errors indicated by *errno*

EADDRINUSE

The specified address is already in use.

EAFNOSUPPORT

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

EBADF

s is not a valid descriptor.

ECONNREFUSED

The connection attempt was rejected, probably because the requested service was not available at the time of the function call.

EFAULT

The length of the area for accepting the address is too small.

EINPROGRESS

Connection setup has not yet been completed successfully.

EISCONN

The socket already has a connection.

ENETDOWN

The connection to the network is down.

Note

If the connection is established with a non-blocking socket of the type SOCK_STREAM (either with *soc_ioctl()* NONBLOCKING being set or by using an external bourse), in the case of an application produced with Sockets >= V2.6 a return value of -1 can occur with *errno* EINPROGRESS. This means that the connection has not been successfully established at the time control is returned to the caller. Consequently, before this socket is used you must use *select()* or *soc_poll()* to check that it can be written to.

When a write/read access takes place before the connection has been fully established, it is rejected with a return value of -1 and the *errno* EWOULDBLOCK.

See also

accept(), getsockname() select(), soc_close(), socket()

freeaddrinfo() - release memory for addrinfo structure

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

Kernighan-Ritchie-C:
int freeaddrinfo(ai);

struct addrinfo *ai;

ANSI-C:
int freeaddrinfo(struct addrinfo* ai);
```

Description

The *freeaddrinfo()* function release memory area for a concatenated list of *struct addrinfo* objects which was requested beforehand with the *getaddrinfo()* function.

The *ai* parameter is a pointer to the first *addrinfo* object in a list of several concatenated*addrinfo* objects.

The *addrinfo* structure is declared as follows:

```
struct addrinfo {
  int              ai_flags;     /* AI_PASSIVE,AI_CANONNAME,AI_NUMERICHOST*/
  int              ai_family;    /* PF_INET,PF_INET6 */
  int              ai_socktype;  /* SOCK_STREAM,SOCK_DGRAM*/
  int              ai_protocol;  /* 0 (not supported in SOCKETS) */
  size_t           ai_addrlen;   /* length of the address */
  char*            ai_canonname; /* canon name of the node */
  struct sockaddr *ai_addr;      /* socket address structure of address */
                                 /* family AF_INET or AF_INET6 */
  struct addrinfo *ai_next;      /* next structure in concatenated list */
};
```

See also

getipnodebyname(), getipnodebyaddr()

freehostent() - release memory for hostent structure

```
#include <netdb.h>

Kernighan-Ritchie-C:
void freehostent(ptr);

struct hostent *ptr;

ANSI-C:
void freehostent(struct hostent* ptr);
```

Description

The *freehostent()* function releases memory for an object of the type *struct hostent* which was requested beforehand with the *getipnodebyname()* or *getipnodebyaddr()* function.

The *ptr* parameter points to an object of the type *struct hostent.*

The *hostent* structure is declared as follows:

```
struct hostent {
   char *h_name;           /* socket host name */
   char **h_aliases;       /* alias list */
   int h_addrtype;         /* address type */
   int h_length;           /* length of address (in bytes) */
   char **h_addr_list;     /* list of addresses for the host */
                           /* terminated with a null pointer */
};
```

gai_strerror() - output text for the error code of getaddrinfo()

```
#include <netdb.h>

Kernighan-Ritchie-C:
char* gai_strerror(ecode);

int ecode;

ANSI-C:
char* gai_strerror(int ecode);
```

Description

The *gai_strerror() function* outputs an explanatory text string for an error code defined in <netdb.h>. The *ecode* parameter specifies an error code defined in <netdb.h>.

Return value

gai_strerror() returns a pointer to the string containing the explanatory text. If the value for *ecode* does not match any of the error codes for *getaddrinfo()* defined in <netdb.h>, the return value is a pointer to a string indicating an unknown error.

getaddrinfo() - get information about host names, host addresses and services regardless of protocol

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

Kernighan-Ritchie-C:
int getaddrinfo(nodename, servname, hints, res);

const char *nodename;
const char *servname;
const struct addrinfo *hints;
const struct addrinfo **res;

ANSI-C:
int getaddrinfo(const char* nodename, const char* servname, const struct addrinfo*
hints, const struct addrinfo** res);
```

Description

The *getaddrinfo()* function allows host information for the AF_INET and AF_INET6 address families to be queried regardless of the protocol involved.

nodename and servname parameters

When *getaddrinfo()* is called, at least one of the parameters *nodename* or *servname* must be not be the null pointer. *nodename a*nd *servname* are either a null pointer or a string terminated with the null byte. The *nodename* parameter can be a name or an IPv4 address in decimal dotted notation or an IPv6 address in hexadecimal colon notation. The *servname* parameter can be either a service name or a decimal port number.

hints parameter

The *hints* parameter can be used to pass an *addrinfo* structure if desired. If not, the *hints* parameter must be the null pointer.

The *addrinfo* structure is declared as follows:

```
struct addrinfo {
  int              ai_flags;     /* AI_PASSIVE,AI_CANONNAME,AI_NUMERICHOST*/
                                 /* AI_NUMERICSERV,AI_V4MAPPED,AI_ALL     */
                                 /* AI_NUMERICHOST*/
  int              ai_family;    /* PF_INET,PF_INET6 */
  int              ai_socktype;  /* SOCK_STREAM,SOCK_DGRAM*/
  int              ai_protocol;  /* 0 (not supported in SOCKETS) */
  size_t           ai_addrlen;   /* length of the address */
  char             ai_canonname; /* canon name of the node */
  struct sockaddr *ai_addr;      /* socket address structure of the address*/
                                 /* family AF_INET or AF_INET6 */
  struct addrinfo  ai_next;      /* next structure in concatenated list */
};
```

All the elements in the object of the type *struct addrinfo* passed with *hints* except *ai_flags*, *ai_family*, *ai_socktype* must have the value 0 or must be the null pointer.

A selection is made with the values for the *addrinfo* components *ai_flags, ai_family a*nd *ai_socktype*:

- *ai_family* = PF_UNSPEC means that any protocol family is desired.
- *ai_socktype* = 0 means that an *addrinfo* structure is to be created for each socket type with the required service.
- *ai_flags* = AI_PASSIVE means that the returned socket address structure is to be used for a *bind()* call. If *nodename* = NULL (see above), the IP address element is set to INADDR_ANY for an IPv4 address and to IN6ADDR_ANY for an IPv6 address.
- If the AI_PASSIVE bit is not set, the returned socket address structure is used
  - for a *connect()* call if *ai_socktype* = SOCK_STREAM
  - for a *connect()-, sendto()-, sendmsg()* call if *ai_socktype* = SOCK_DGRAM

  If, in these cases, *nodename* is the null pointer, the IP address of *sockaddr* is supplied with the value of the loopback address.
- If the AI_CANONNAME bit is set in the *ai_flags* of the *hints* structure and *getaddrinfo()* is executed successfully, at least the first returned *addrinfo* structure in the element *ai_canonname* contains the pointer to the canon name terminated with the null byte of the selected host.

  > The *ai_canonname* is determined by a reverse lookup. If this reverse lookup is not successful, i.e. no name is found for the specified address, no error is reported, but the content of the *nodename* parameter is copied into the *ai_canonname* element if it is not equal to the null pointer. If *nodename* is a null pointer, a null pointer is entered in the *ai_canonname* element.
  > Please note that the content of *nodename* can also be an address! This is then also copied.

- If the AI_NUMERICHOST bit is set in the *ai_flags* of the *hints* structure, a *nodename* which is not the null pointer must be an IPv4 address string in decimal dotted notation or an IPv6 address string in hexadecimal colon notation. Otherwise, the return value is EAI_NONAME. The flag prevents a call that would resolve the name via a DNS service or internal host table.
- If the AI_V4MAPPED bit is set in the *ai_flags* of the *hints* structure together with *ai_family* = PF_INET6 and no IPv6 addresses are supplied for the name, IPv4 addresses contained in the output list are entered in the form of IPv4-mapped IPv6 addresses. If the AI_ALL bit is also set, both IPv6 and the IPv4-mapped IPv6 addresses are entered.
- If the AI_ADDRCONFIG bit is set in the *ai_flags* of the *hints* structure, IPv4 or IPv6 addresses which belong to the name are output only if a corresponding interface address is defined on the local computer. The loopback address does not count as a configured interface address here.
- If the AI_NUMERICSERV bit is set in the *ai_flags* of the *hints* structure, the pointer of *servname* which is not a null pointer must point to a numerical port number string. If this is not the case, an error message (EAI_NONAME) is returned.

hints = NULL has the same effect as an *addrinfo* structure initialized with 0 and *ai_family* = PF_UNSPEC.

res parameter

If *getaddrinfo()* is executed successfully, a pointer to one or more concatenated *addrinfo* structures is passed in *res*, where the element *ai_next* = NULL indicates the last element in the chain. Each of the returned *addrinfo* structures contains a value corresponding to the *socket()* call in the elements *ai_family* and *ai_socktype*. *ai_addr* always points to a socket address structure whose length is specified in *ai_addrlen*.

Return value

0:

If successful.

>0:

If errors occur. Return value is an error code EAI_xxx defined in <netdb.h>.

-1:

If errors occur. *errno* is set to indicate the error.

Error code defined in <netdb.h>:

EAI_ADDRFAMILY

The Internet address families are not supported for the specified host.

EAI_AGAIN

Temporary error while accessing the host name information (e.g. DNS error).
The function should be called again.

EAI_BADFLAGS

Invalid value for the *ai_flags* parameter.

EAI_FAIL

Error while accessing the host name information

EAI_FAMILY

The protocol family is not supported.

EAI_MEMORY

Error when requesting memory.

EAI_NODATA

No address corresponding to the host name was found.

EAI_NONAME

Host or service name is not supported or is unknown.

EAI_SERVICE

Service is not supported for this socket type.

EAI_SOCKTYPE

The socket type is not supported.

EAI_SYSTEM

System error; is specified in more detail in *errno*.

Note

Memory for the *addrinfo* structures returned by the *getaddrinfo()* function is requested dynamically and must be released again with the *freeaddrinfo()* function.

In SOCKETS(BS2000), PF_UNSPEC = AF_UNSPEC.

If you are not using DNS, do not specify a *fully-qualified-domain-name* but rather a host name if you want to be able to use the BCAM processor table (e.g. *host* instead of *host.mydomain.net*).

The GETDNS client which is common to the SOCKETS(BS2000), BCAM and POSIX sockets is used to access the DNS. For more details, see the manual "BCAM Volume 1/2".

getbcamhost() - get BCAM host name

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

Kernighan-Ritchie-C:
int getbcamhost(bcamname, bcamnamelen);

char *bcamname;
int bcamnamelen;

ANSI-C:
int getbcamhost(char* bcamname, int bcamnamelen);
```

Description

Use of the *getbcamhost()* function only makes sense in the AF_ISO address family.

getbcamhost() returns the BCAM host name in the *bcamname* parameter. The BCAM host name is used for the ISO transport service in the AF_ISO address family and corresponds to the local network selector NSEL. The BCAM host name has a fixed length of 8 characters; blanks are permitted at the end of the name.

The length of the *bcamname* string variable must be specified in the *bcamnamelen* parameter in the *getbcamhost()* call.

If the length of the *bcamname* string variable specified by *bcamnamelen* is sufficient to accept the host name, the host name is terminated with a null byte. Otherwise, the excess host name characters are truncated, and it is then undefined whether the host name returned in this way is terminated by a null byte.

Definition of BCAM host name: see *getsockopt()* in section "getsockopt(), setsockopt() - get and set socket options"

Return value

0:

If successful.

-1:

If errors occur. *errno* is not set.

getdtablesize() - get size of descriptor table

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

Kernighan-Ritchie-C:
int getdtablesize();

ANSI-C:
int getdtablesize();
```

Description

The *getdtablesize()* function returns the size of the socket descriptor table in bits. The table is valid for all the supported address families, i.e. it contains all the possible descriptors for all the address families.

Return value

0:

If successful.

-1:

If errors occur. *errno* is not set.

See also

select()

gethostbyaddr(), gethostbyname() - get information about host names and addresses

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

Kernighan-Ritchie-C:
struct hostent *gethostbyaddr(addr, len, type);

char *addr;
int len;
int type;

struct hostent *gethostbyname(name);

char *name;

ANSI-C:
struct hostent* gethostbyaddr(char* addr, int len, int type);
struct hostent* gethostbyname(char* name);
```

Description

Use of the *gethostbyaddr()* and *gethostbyname()* functions only makes sense in the AF_INET address family.

The *gethostbyaddr()* and *gethostbyname()* functions return current information on all known hosts on the network by obtaining the required information (host name and host address) from a DNS server. Otherwise, i.e. only in cases where this is not successful, the information taken from the BCAM processor table (see the "BCAM Volume 1/2" manual for details).

For *gethostbyaddr()*, *addr* is a pointer to the host address. This host address must be available in binary format with the length *len*. The only valid entry for *type* is AF_INET.For *gethostbyname()*, the host name must be specified for *name*.

The *gethostbyaddr()* and *gethostbyname()* functions return a pointer to an object with the *hostent* structure described below.

The *hostent* structure is declared as follows:

```
struct hostent {
   char *h_name;           /* socket host name */
   char **h_aliases;       /* alias list */
   int h_addrtype;         /* address type */
   int h_length;           /* length of the address in bytes */
   char **h_addr_list;     /* list of addresses for the host, */
                           /* terminated by the null pointer */
};
#define h_addr h_addr_list[0]; /* first address, network byte order */
```

Description of *hostent* components:

h_name

Name of the host

h_aliases

A list of alternative names (aliases) for the host, terminated with null.
Aliases are currently not supported.

h_addrtype

Type of the returned address (always AF_INET)

h_length

Length of the address in bytes

**h_addr_list

A pointer to a null-terminated list of network addresses for the host. These addresses of length *h_length* are returned in network byte order.

Return value

The null pointer is returned if errors occur or the end of the file is reached.

Note

The data returned in the *hostent* object is supplied in a static area that is overwritten with each new *gethostby...()* call. It must therefore be copied if it needs to be saved.

The GETDNS client which is common to SOCKETS(BS2000), BCAM und POSIX sockets is used to access the DNS in the case of *gethostbyname()* and *gethostbyaddr()*. For more details, see the manual “BCAM Volume 1/2”. The POSIX resolver daemon *dnsd* is no longer used.

gethostname() - get the name of the current host

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

Kernighan-Ritchie-C:
int gethostname(name, namelen);

char *name;
int namelen;

ANSI-C:
int gethostname(char* name, int namelen);
```

Description

Use of the *gethostname()* function only makes sense in the AF_INET and AF_INET6 address families.

The *gethostname()* function returns the socket host name in the *name* parameter. Socket host names are unique network-wide and are assigned in TCP/IP networks to all hosts that can be reached via a TCP/IP route (see the “BCAM Volume 1/2” manual for details).

The length of the *name* string variable must be specified in the *namelen* parameter in the *gethostname()* call.

If the length of the *name* string variable specified by *namelen* is sufficient to accept the host name, the host name is terminated with a null byte. Otherwise, the excess host name characters are truncated, and it is then undefined whether the host name returned in this way is terminated by a null byte.

Return value

0:

If successful.

-1:

If errors occur. *errno* is not set.

getipnodebyaddr(), getipnodebyname() - get information about host names and addresses

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

Kernighan-Ritchie-C:
struct hostent *getipnodebyaddr(addr, len, type, error_num);

char *addr;
int len;
int type;
int *error_num;

struct hostent *getipnodebyname(name, af, flags, error_num);

char *name;
int af;
int flags;
int *error_num;

ANSI-C:
struct hostent* getipnodebyaddr(char* addr, int len, int type, int* error_num);

struct hostent* getipnodebyname(char* name, int af, int flags, int* error_num);
```

Description

Use of the *getipnodebyaddr()* and *getipnodebyname()* functions only makes sense in the AF_INET and AF_INET6 address families. *getipnodebyaddr()* and *getipnodebyname()* are extensions of the functions *gethostbyaddr()* and *gethostbyname()* for IPv6 support.

The *getipnodebyaddr()* and *getipnodebyname()* functions return current information on all known hosts on the network by obtaining the required information (host name and host address) from a DNS server. Otherwise, i.e. only in cases where this is not successful, the information taken from the BCAM processor table (see the "BCAM Volume 1/2" manual for details).

For *getipnodebyaddr()*, *addr* is a pointer to the host address. This host address must be available in binary format with the length *len*. The only valid entry for *type* is AF_INET or AF_INET6.

For *getipnodebyname()*, the host name (socket host name) must be specified for *name*. You can specify the name

- as a fully-qualified DNS name, i.e. including host name and domain part (e.g. hostname.company.com) or
- as a partially-qualified DNS name (e.g. hostname) or
- only as a host name (e.g. hostname).

You can also specify an IPv4 address in decimal dotted notation or an IPv6 address in hexadecimal colon notation. If you do so, the corresponding address families must be specified for *af*. In this case, the converted binary address is returned in the *hostent* return structure. If an IPv4 address in decimal dotted notation and *af* = AF_INET6 and *flags* = AI_V4MAPPED is specified, a binary IPv4-mapped IPv6 address is returned in the output structure.

The *af* parameter in the call is used to specify the address family (AF_INET or AF_INET6).

The *flags* parameter can be used to control the output of the desired address family. If *flags* has the value 0, an address appropriate to the address family specified in *ai* is returned.

In the address family *af*, *flags* can be used to specify different options (they are defined in <netdb.h>):

AI_V4MAPPED

The caller accepts IPv4-mapped addresses if no IPv6 address is available.

AI_ALL

IPv6 addresses and IPv4-mapped addresses are returned if available. *af* must have the value AF_INET6.

AI_ADDRCONFIG

Depending on the value of *af*, only an IPv6 or IPv4 address is returned if the host on which the function is called has an interface address of the same type.

AI_DEFAULT

is the same as AI_ADDRCONFIG || AI_V4MAPPED.

- If *af* = AF_INET6 is set and the host on which the function is called has an IPv6 address, an IPv6 address is returned for the specified host name.
- If the host on which the function is called has only an IPv4 interface address, an IPv4-mapped IPv6 address is returned.

The *getipnodebyaddr()* and *getipnodebyname()* functions return a pointer to an object of the *hostent* structure described below. Memory for this object is requested dynamically and must be released again by the caller with the *freehostent()* function.

The *hostent* structure is declared as follows:

```
struct hostent {
   char *h_name;           /* socket host name */
   char **h_aliases;       /* alias list */
   int h_addrtype;         /* address type */
   int h_length;           /* length of the address (in bytes) */
   char **h_addr_list;     /* list of addresses for the host */
                           /* terminated with the null pointer */
};
#define h_addr h_addr_list[0]; /* first address, network byte order */
```

Description of *hostent* components:

h_name

Name of the host

h_aliases

A list of alternative names (aliases) for the host, terminated with null.
Aliases are currently not supported.

h_addrtype

Type of the returned address (always AF_INET)

h_length

Length of the address in bytes

**h_addr_list

A pointer to a null-terminated list of network addresses for the host. These addresses of length *h_length* are returned in network byte order.

Return value

Pointer to an object of the type *struct hostent*. If an error occurs, the null pointer is returned and the variable *errnum* is supplied with one of the following values. These values are defined in <netdb.h>.

HOST_NOT_FOUND

Host unknown.

NO_ADDRESS

No host address is available for the specified name.

NO_RECOVERY

An unrecoverable server error has occurred.

TRY_AGAIN

Access must be repeated.

Note

When DNS is not used, as a rule it makes sense not to specify a Fully Qualified Domain Name (FQDN), but only the host name in order to obtain the corresponding addresses from BCAM (e.g. *host* instead of *host.mydomain.net*). The use of FQDNs makes sense on systems on which DNS is not used only when an FQDN file with entries exists.

The GETDNS client which is common to the SOCKETS(BS2000), BCAM and POSIX sockets is used to access the DNS. For more details, see the manual “BCAM Volume 1/2”.

getnameinfo() - get the name of the communications partner

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

Kernighan-Ritchie-C:
int getnameinfo (sa, salen, host, hostlen, serv, servlen, flags);

const struct sockaddr *sa;
socklen_t salen;
char *host;
size_t hostlen;
char *serv;
size_t servlen;
int flags;

ANSI-C:
int getnameinfo (const struct sockaddr* sa, socklen_t salen, char* host,
size_t hostlen, char* serv, size_t servlen, int flags);
```

Description

The *getnameinfo()* function returns the name assigned to the IP address and port number specified in the call as a text string. The values are determined using either the DNS service or system-specific tables.

The *sa* parameter is a pointer to a *sockaddr_in* structure (for IPv4) or a *sockaddr_in6* structure (for IPv6), which contains the IP address and port number.
salen indicates the length of these structures.

If *getnameinfo()* is executed successfully, *host* is a pointer to the socket host name which corresponds to the specified IP address. The socket host name is terminated with the null byte, and its length corresponds to the value of *hostlen*. The same applies to the service name which corresponds to the specified port number. This is the service name to which the pointer *serv* points, and its length (including the null byte) corresponds to the value of *servlen*.

If the value 0 is specified for *hostlen* or *servlen* when *getnameinfo()* is called, this indicates that no name is to be returned in the corresponding *host* parameter or no service name or port number is to be returned in the *serv* parameter respectively.

However, a sufficiently large buffer, which can accommodate the host and service names including the null byte, must be made available for the desired information.

Specification of the maximum lengths for DNS and service names in <netdb.h>:

```
#define NI_MAXHOST  1025

#define             32
NI_MAXSERV
```

The *flags* parameter changes how *getnameinfo()* is executed. Normally, the fully-qualified domain name of the host is determined from the DNS and returned. Depending on the value of *flags*, a distinction is made between the following cases:

- If the *flags* bit NI_NOFQDN is set, only the host name part of an FQDN is returned.

  > The GETDNS client which is common to the SOCKETS(BS2000), BCAM and POSIX sockets is used to access the DNS. For more details, see the manual “BCAM Volume 1/2”.

- If the *flags* bit NI_NUMERICHOST is set, or it is impossible to determine the host name in the DNS or using local information, the numeric host name is returned in printable format after address conversion.
- If the *flags* bit NI_NAMEREQD is set, an error is reported if the host name cannot be determined in the DNS.
  If the *flags* bit NI_NAMEREQD is set in combination with NI_NOFQDN, the bit has no effect.
- If the *flags* bit NI_NUMERICSERV is set, the port number is returned in printable format instead of the service name.
- If the *flags* bit NI_DGRAM is set, the service name for the udp protocol is returned. If NI_DGRAM is not specified, the service name for the tcp protocol is always returned.

Return value

0:

If successful.

>0:

If an error occurs
As thread savety is required for the DNS Resolver, *errnos* cannot be set.
If the return value > 0, it corresponds to the value of an EAI_xxx error code as defined in <netdb.h>.

<0:

If an error occurs
An error has occurred, which prevents execution of the function. Therefore *errno* is set.

getpeername() - get the remote address of the socket connection

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

#include <netinet.in.h> /* nur bei AF_INET und AF_INET6 */
#include <iso.h> /* nur bei AF_ISO */

Kernighan-Ritchie-C:
int getpeername(s, name, namelen);

int s;
int *namelen;

struct sockaddr_in *name; /* only for AF_INET */
struct sockaddr_in6 *name; /* only for AF_INET6 */
struct sockaddr_iso *name; /* only for AF_ISO */

ANSI-C:
int getpeername(int s, struct sockaddr* name, int* namelen);
```

Description

The *getpeername()* function returns the name of the communications partner connected to socket *s* in the *name* parameter.

name points to a memory area. After *getpeername()* has been executed successfully, **name* contains the name (address) of the communications partner.
The integer variable to which the *namelen* parameter points must be assigned the maximum possible address length (in bytes) before *getpeername()* is called. After the function returns, **namelen* contains the current size of the returned name in bytes.

Return value

0:

If successful.

-1:

If errors occur. *errno* is set to indicate the error.

Errors indicated by *errno*

EBADF

The *s* parameter is not a valid descriptor.

EFAULT

The length of the area for accepting the address is too small.

ENOBUFS

There is not enough storage space in the buffer.

ENOTCONN

The socket has no connection.

EOPNOTSUPP

Socket *s* is not of type SOCK_STREAM, and the operation is not supported for the socket type of *s*.

See also

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

getprotobyname() - get the number of the protocol

```
#include <netdb.h>

Kernighan-Ritchie-C:
struct protoent *getprotobyname(name);

char *name;

ANSI-C:
struct protoent* getprotobyname(char* name);
```

Description

Use of the *getprotobyname()* function only makes sense in the AF_INET and AF_INET address families.

The *getprotobyname()* function returns a pointer to an object with the *protoent* structure described below. This structure contains the protocol number associated with the protocol name *name*.

The *protoent* structure is declared in <netdb.h> as follows:

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

Description of *protoent* components:

p_name

Name of the protocol

p_aliases

A list of alternative names (aliases) for the protocol, terminated with null.
Aliases are currently not supported.

p_proto

Number of the protocol; result field of *getprotobyname()*.

Return value

Pointer to an object of type *struct protoent*. The null pointer is returned if an error occurs.

Note

The data returned in the *protoent* object is supplied in a static area that is overwritten with each new *getprotobyname()* call. It must therefore be copied if it needs to be saved.

getservbyname(), getservbyport() - get information about services

```
#include <netdb.h>

Kernighan-Ritchie-C:
struct servent *getservbyname(name, proto);

char *name;
char *proto;

struct servent *getservbyport(port, proto);

int port;
char *proto;

ANSI-C:
struct servent* getservbyname(char* name, char* proto);

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

Description

Use of the *getservbyname()* and *getservbyport()* functions only makes sense in the AF_INET and AF_INET6 address families.

The *getservbyname()* and *getservbyport()* functions return information on the available services from the services file with the default name SYSDAT.BCAM.ETC.SERVICES which is managed by BCAM (see the “BCAM Volume 1 /2” manual). Both function return a pointer to an object with the *servent* structure described below.

getservbyname() returns the port number associated with the service name *name* and theprotocol *proto* in the *servent* object. If NULL is specified for *proto*, the service name and the port number of the first protocol found in the list are output.

getservbyport() returns the service name associated with the port number *port* and theprotocol *proto* in the *servent* object, as well as the (up to) four aliases which can be entered. If NULL is specified for *proto*, the service name and the aliases of the first protocol found in the list are output for the specified port number.

The *servent* structure is declared in <netdb.h> as follows:

```
struct servent {
   char *s_name;            /* name of the service */
   char **s_aliases;        /* alias list */
   int s_port;              /* number of the port on which the service lies*/
   char *s_proto;           /* protocol used */
};
```

Description of *servent* components:

s_name

Name of the service

s_aliases

A list of alternative names (aliases) for the service, terminated with null

s_port

Port number assigned to the service. Port numbers are returned in network byte order.

s_proto

Name of the protocol that must be used to access the service.

As long as a protocol name (not NULL) is specified, *getservbyname()* and *getservbyport()* search for the service that uses the matching protocol.

Return value

The null pointer is returned if the search reaches the end of the file.

Note

The data returned in the *servent* object is supplied in a static area and must therefore be copied if it needs to be saved.

getsockname() - get local address of the socket connection

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

#include <netinet.in.h> /* only for AF_INET and AF_INET6 */
#include <iso.h> /* only for AF_ISO */

Kernighan-Ritchie-C:
int getsockname(s, name, namelen);
int s;
int *namelen;

struct sockaddr_in *name; /* only for AF_INET */
struct sockaddr_in6 *name; /* only for AF_INET6 */
struct sockaddr_iso *name; /* only for AF_ISO */

ANSI-C:
int getsockname(int s, struct sockaddr* name, int* namelen);
```

Description

The *getsockname()* function returns the current name for socket *s* in the *name* parameter.

name points to a memory area. On successful execution of *getsockname()*, **name* contains the name (address) of socket *s*. Before calling *getsockname()*, the integer variable to which the *namelen* parameter points must be supplied with the address length (in bytes). When the function returns, **namelen* contains the current size of the returned name in bytes.

Return value

0:

If successful.

-1:

If errors occur. *errno* is set to indicate the error.

Errors indicated by *errno*

EBADF

The *s* parameter is not a valid descriptor.

EFAULT

The length of the area for accepting the address is too small.

EOPNOTSUPP

Socket *s* is not of type SOCK_STREAM, and the operation is not supported for the socket type of *s*.

See also

bind(), getpeername(), socket()

getsockopt(), setsockopt() - get and set socket options

```
#include <sys.types.h>
#include <sys.socket.h>
#include <netinet.in.h> /* only for AF_INET or AF_INET6 */

Kernighan-Ritchie-C:
int getsockopt(s, level, optname, optval, optlen);

int s;
int level;
int optname;
char *optval;
int *optlen;

int setsockopt(s, level, optname, optval, optlen);

int s;
int level;
int optname;
char *optval;
int optlen;

ANSI-C:
int getsockopt(int s, int level, int optname, char* optval, int* optlen);

int setsockopt(int s, int level, int optname, char* optval, int optlen);
```

Description

The *getsockopt()* function can be used to get the properties (options) of the socket interface or of a single socket *s* via the *optname*, *optval* and *optlen* parameters.

The *setsockopt()* function can be used to set the properties (options) of the socket interface or of a single socket *s* via the *optname*, *optval* and *optlen* parameters.

You can use the *level* parameter to specify whether you want to get or set the options of the socket interface or of a single socket.

The following values are allowed as the current value for *level*:

SOL_GLOBAL

Debugging outputs are enabled.

SOL_SOCKET

Get or set options of socket *s* of the AF_INET or AF_INET6 address family.

SOL_TRANSPORT

Get or set options of socket *s* of the AF_ISO address family.

IPPROTO_TCP

Get or set options of socket *s* in the AF_INET or AF_INET6 address family for modifying the protocol behavior of the TCP protocol.

IPPROTO_IPV4

Get or set options of socket *s* in the AF_INET address family for modifying the protocol behavior of the IPv4 protocol. For reasons of compatibility, both IPPROTO_IP and IPPROTO_IPV4 are evaluated.

IPPROTO_IPV6

Query or modify options for the socket *s* in the address family AF_INET6 for modifying the behavior of the IP protocol.

IPPROTO_ICMP

Query or modify options for the socket *s* in the address family AF_INET of the type SOCK_RAW for modifying the protocol behavior of the IP protocol in conjunction with the ICMP protocol.

IPPROTO_ICMPV6

Query or modify options for the socket *s* in the address family AF_INET6 of the type SOCK_RAW for modifying the protocol behavior of the IP protocol in conjunction with the ICMPv6 protocol.

> ¡ A bind() is neccessary for some of the options with *level* = IPPROTO_IPV4 and IPPROTO_IPV6.

Options for the SOL_GLOBAL level

At SOL_GLOBAL level operand s is meaningless and should be assigned the value 0.

optname = SO_DEBUG enables debugging outputs with verious depths of detail to be enabled.

With *getsockopt()* and *optlen* with a value of 4 (*sizeof int*), the value of the debugging output level currently set is output in *optval*.

The value 0 disables the debugging output.

With *setsockopt()* and *optlen* with a value of 4, the following debugging output level shown in the table below can be specified in *optval*

**optval* >= 1: Function calls		
**optval* >= 6: Function calls	+ additional information	
**optval* = 8: Function calls	+ additional information	BCAM parameter area after call
**optval* = 9: Function calls	+ additional information	+ BCAM parameter area after call + BCAM parameter area before call

With *setsockopt()* and *optname* = SO_ASYNC and *optlen* = 4 and *optval* with a pointer to the short ID of the event ID of the external bourse to be used, an external wait point is configured for SOCKETS(BS2000) when this is the first socket call of the subsystem (see chapter "setsockopt() - modify socket options").

optname	**optlen*	Value range of *optval*
SO_ASYNC	4	Pointer to short ID of the event ID

With *setsockopt()* and *optname* = SO_RESOLVE_BCAM and *optlen* = 4 the order of DNS resolving can be configured. The value 0 in *optval* corresponds to the standard behaviour - a DNS resolution first via the DNS resolver and only if the resolver does not return a result a resolution via the BCAM internal structures is triggered. If the value 1 was given, SOCKETS will first try to resolve via the BCAM internal structures before contacting the DNS resolver.

optname	*optlen*	*optval*
SO_RESOLVE_BCAM	4	0, 1

With the corresponding *getsockopt()* call the current setting in SOCKETS can be retrieved.

optname	**optlen*	Output format in *optval*
SO_RESOLVE_BCAM	4	int

Options for the SOL_SOCKET (AF_INET, AF_INET6) level

In this case, *s* specifies the socket for which the options are to be retrieved or set, and *optname* specifies the name of the option for which a value is to be retrieved or set.

With *getsockopt()*, *optval* and *optlen* identify the respective buffers in which the value of the desired option is returned. **optlen* initially contains the size of the buffer to which *optval* points. When the *getsockopt()* function returns, **optlen* contains the current size of the returned buffer. If the option in question has no value that can be returned, **optval* contains the value 0.

The following values can be returned by *getsockopt()* for *optname* and *optlen* in the AF_INET and AF_INET6 address families:

optname	***** *optlen*	Output format in *optval*
SO_DEBUG	4	int
SO_DISHALIAS	4	int
SO_ERROR	4	int
SO_KEEPALIVE	4	int
SO_LINGER	>= sizeof(struct linger)	*(struct linger)
SO_OUTPUTBUFFER	4	int
SO_RCVBUF	4	int
SO_SNDBUF	4	int
SO_TIMESTAMP	4	int

SO_TSTIPAD	4	int
SO_TYPE	4	int
SO_VHOSTANY	8	*(char[8])

SO_KEEPALIVE

An output value >0 specifies that a timer value has been set with *setsockopt()*. The caller cannot recognize the connection status for the selected socket. If the connection is active, the timer value is determined by BCAM from the connection information. If the connection is not yet active, the timer value is read from the socket.

If a timer value = 0 is output, this does not necessarily mean that KEEPALIVE is disabled! A global setting of the KEEPALIVE timer by the BCAM administration cannot be seen by means of this function.

The *setsockopt()* function can be used to set or change option values via the *optval* and *optlen* parameters. You can specify the following values for *optname*, *optlen* and *optval* in the AF_INET and AF_INET6 address families:

optname	*optlen*	*optval*
SO_BROADCAST (only AF_INET)	4	
SO_DEBUG	4	0 <= *optval* <= 9
SO_DISHALIAS	4	0, 1
SO_KEEPALIVE	4	0; 120...32767
SO_LINGER	>= sizeof(struct linger)	*(struct linger)
SO_REUSEADDR	4	0, 1
SO_TIMESTAMP	4	0, 1
SO_VHOSTANY	4	*(char[9])

The valid value range of *optval* for the SO_KEEPALIVE option is 0 and 120 ... 32767:

- If the value is 0, the timer is switched off.
- If the value is in the range of 120 ... 32767, the timer is switched on and the timer interval is set with the specified value (unit of measurement: seconds).

If a value outside the valid range is specified, the timer interval is set with the default value of the transport system.

SO_BROADCAST (only AF_INET)

This option has no functional meaning for sockets. Only a syntax check is performed.

If the syntax is valid, the value 0 is returned; otherwise, -1.

SO_DEBUG

If *level* = SOL_GLOBAL, this option defines the debugging level for the sockets of the active task.
If *level* = SOL_SOCKET, this option has no functional significance; only a syntax check is performed.

If the syntax is valid, the value 0 is returned; otherwise, -1.

SO_DISHALIAS

A value >0 makes an entry in the socket that host aliasing is to be deactivated for this application with the *bind()* call.

SO_ERROR

Shows the number of the last error issued.

SO_KEEPALIVE

Specifies whether TCP-KEEPALIVE monitoring is to be performed on this connection.

In particular it specifies:

- Whether KEEPALIVE monitoring should be activated in the TCP protocol machine for the current connection.
- Which time interval (in seconds) should be selected for this monitoring.

The effect of SO_KEEPALIVE depends on the status of the corresponding socket:

- If an active connection is not established for the socket, the desire to activate KEEPALIVE monitoring with the corresponding value of the timer interval is noted in the socket structure and transferred to the transport system when a connection is established.

  With a server application, i.e. in the case of a passive connection establishment, the active *listen()* socket must be executed with SO_KEEPALIVE, so that monitoring is automatically switched on every time a connection is established.
- If an active connection already exists for the socket, the information is transmitted to the TCP protocol machine using an internal transport system call.

If the time interval value is 0, monitoring is deactivated:

- If a connection already exists, monitoring is deactivated immediately.
- If no connection exists, monitoring is deactivated when a connection is established. With the server, the *listen()* socket must be marked accordingly.

> **i** Due to the various ways in which TCP protocol machines are implemented, it cannot be guaranteed that the connections are maintained (see also RFC 1122).

SO_LINGER

The SO_LINGER option uses a parameter of data type *struct linger*. This parameter specifies the desired option status and the delay interval.

The *linger* structure is defined in <sys.socket.h>. The current option status and delay interval can be obtained through a *linger* structure via *getsockopt()*.

```
struct linger {
        int l_onoff; /* option on/off */
        int l_linger; /* linger time */
};
```

The *l_linger* parameter specifies the maximum time for executing *soc_close()*, *l_onoff* activates and deactivates the linger function (0 = OFF, >0 = ON).

SO_OUTPUTBUFFER

Shows the user data accepted by the socket interface but not yet acknowledged by the partner transport system.

SO_RCVBUF

Shows the size of the receive (input) buffer.

SO_REUSEADDR

This option has no functional significance if the application was produced with SOCKETS(BS2000) version <= V2.1 or encounters BCAM < V18 ; only a syntax check is performed.

If the application was produced as of SOCKETS(BS2000) V.2.2 then the SO_REUSEADDR functionality is required as part of multihoming support. SO_REUSEADDR only affects the specified socket and must be set before *bind()*.

SO_REUSEADDR is set with *optval* = 1 and reset with *optval* = 0.

If the syntax is valid, the value 0 is returned; otherwise, -1.

SO_SNDBUF

Shows the size of the send (output) buffer.

SO_TIMESTAMP

all packets received via *recvmsg()* obtain a timestamp, which is saved in a *cmsghdr* structure. The *cmsg_data* field of this structure is a *struct timeval* indicating the reception time of the received packet.
The *timeval* structure is defined in <sys.time.h>.

```
struct timeval {
    long    tv_sec;         /* seconds */
    long    tv_usec;        /* and microseconds */
};
```

SO_TIMESTAMP is activated by setting *optval* = 1 and deactivated by setting *optval* = 0. SO_TIMESTAMP is deactivated by default.

Return value of SO_TIMESTAMP:

0: The flag is not set
1: The flag is set

SO_TSTIPAD

Compares the transferred IP address with the interface addresses of the socket host, on which the socket application runs. SO_TSTIPAD reports the comparison result via *optval*.
The IP address is transferred as an IPv4 or IPv6 address via the *optval* pointer that points to a *struct in_addr* or *struct in6_addr* structure.
The specified socket may not exist. However, the file descriptor must be in the permitted value range.
The value of the *optlen* parameter specifies whether it is an IPv4 or IPv6 address. Therefore the value of *optlen* must correspond to the length of *struct in_addr* or *struct in6_addr* depending on the address type used.

When used as a return value the first 4 bytes of the address structure passed at the call are overwritten.

Return value of *optval*:

0: The specified IP address is a separate interface address.

1: The specified IP address is not a separate interface address.

SO_TYPE

Shows the socket type.

SO_VHOSTANY

The BCAM host name of the real or virtual host is specified in the string pointed to by *optval*. This is then entered in the socket.
This makes it possible to address a virtual host with an ANYADDR or LOOPBACKADDR or to read the data from a virtual host with *soc_ioctl(...,SIOCGLIFCONF,....)*. It is also possible to address a real host if you are working under an ID assigned to a virtual host by an entry in the BCAM application table.
The name of the BCAM host as entered in the socket is output on reading.

BCAM host name:

The name is eight characters in length. Alphanumeric characters and the special characters #, @, $ or blanks can be used at the end of the name. As a rule, uppercase characters should be used, but the name is case-sensitive. Names comprising numeric characters only are not permitted.

Options for the IPPROTO_IPV4 (AF_INET) level

The *getsockopt()* function can be used to specify the following values for *optname* and *optlen* in the AF_INET address family:

optname	***** *optlen*	Output format in *optval*
IP_MTU_DISCOVER	4	int
IP_MULTICAST_TTL	4	int
IP_MULTICAST_IF	>= sizeof(struct in_addr)	*(struct in_addr)
IP_MULTICAST_LOOP	4	int
IP_OPTIONS	8..40	char*
IP_RECVERR	4	int
IP_RECVTTL	4	int
IP_TTL	4	int

Return value of IP_MTU_DISCOVER:

0: IP_PMTUDISC_DONT
2: IP_PMTUDISC_DO

Return value of IP_MULTICAST_TTL and IP_TTL:

Value of the selected hop limit.
The bind()-function has to be called before setting IP_MULTICAST_TTL.

Return value of IP_MULTICAST_IF:

IPv4 address of the interface to be used for sending.
The bind()-function has to be called before setting this option.

Return value of IP_MULTICAST_LOOP:

0: Loopback OFF
1: Loopback ON

The bind()-function has to be called before setting this option.

Return value of IP_OPTIONS:

The internet options set by the sender are returned in the same format as they were put in. The length has to be at least as big as the size of the internet options.

Return value of IP_RECVERR and IP_RECVTTL:

0: The flag is not set
1: The flag is set

The *setsockopt()* can be used to modify option values via the *optval()* and *optlen()* parameters. You can specify the following values in the AF_INET address family:

optname	*optlen*	*optval*
IP_ADD_MEMBERSHIP	>= sizeof(struct ip_mreq)	*(struct ip_mreq)
IP_DROP_MEMBERSHIP	>= sizeof(struct ip_mreq)	*(struct ip_mreq)
IP_MTU_DISCOVER	4	IP_PMTUDISC_DO IP_PMTUDISC_DONT
IP_MULTICAST_TTL	4	`0 <` *optval* `<= 255`
IP_MULTICAST_IF	>= sizeof(struct in_addr)	*(struct in_addr)
IP_MULTICAST_LOOP	4	>= 0
IP_OPTIONS	<=40	char*
IP_RECVERR	4	>= 0
IP_RECVTTL	4	0, 1
IP_TTL	4	1..255

The IP_ADD_MEMBERSHIP and IP_DROP_MEMBERSHIP options use a parameter of the type *struct ip_mreq*. This parameter specifies the IPv4 address of the desired multicast group and the local IPv4 address.

The *ip_mreq* structure is defined in <netinet.in.h>.

```
struct ip_mreq {
        struct in_addr imr_multiaddr; /* IP multicast address of group */
        struct in_addr imr_interface; /* local IP address of interface */
 };
```

IP_MULTICAST_TTL

Shows or sets the multicast hop limit.
The bind()-function has to be called before setting this option.
Hop limit values:

0: Send only within the local host (loopback)
1: Send within the local subnetwork
>1: Send beyond router boundaries

IP_ADD_MEMBERSHIP

Activates the delivery of messages of a selected multicast group to this socket.Specifies the multicast or local interface address (IPv4 address or INADDR_ANY, not INADDR_LOOPBACK).
INADDR_ANY is the default interface of BCAM for receiving multicast data.
The bind()-function has to be called before setting this option.

IP_DROP_MEMBERSHIP

Deactivates delivery of messages of a selected multicast group to this socket. Specifies the multicast and local interface addresses (IPv4 address or INADDR_ANY, not INADDR_LOOPBACK).
INADDR_ANY is the default interface of BCAM for receiving multicast data.
The bind()-function has to be called before setting this option.

IP_MTU_DISCOVER

Sets the DF flag in the IPv4 protocol header which permits or prevents fragmentation of an ICMP or UDP packet.

IP_PMTUDISC_DONT	Fragmentation permitted
IP_PMTUDISC_DO	Fragmentation is to be prevented

IP_MULTICAST_IF

IPv4 address of the interface over which transfer is to take place.
The bind()-function has to be called before setting this option.

IP_MULTICAST_LOOP

Is set by the sender of the messages and enables reception on the local sending host.0: OFF, 1: ON (default: ON)
The bind()-function has to be called before setting this option.

IP_OPTIONS

Internet options that can be set by the sender of the messages. The received char-pointer has to be aligned to a 4-byte-boundary. The exact format of the Internet options can be seen in the RFC791 at chapter 3.1. Currently, only Loose Source Route, Strict Source Route, Record Route and the Timestamp Option are supported by Sockets.

IP_RECVERR

Activates the delivery of ICMP error messages to this socket if the option was set before *bind()*. The error reports for datagram and raw sockets are fetched with using recvmsg() and the flag MSG_ERRQUEUE.

IP_RECVTTL

the TTL-field from every IPV4-packet received via *recvmsg()* is read out and saved in a *cmsghdr* structure.

IP_RECVTTL is activated by setting *optval* = 1 and deactivated by setting *optval* = 0. IP_RECVTTL is deactivated by default.

IP_TTL

Modifies the hop limit in the corresponding field of the packet's IP protocol header.

Options for the IPPROTO_IPV6 (AF_INET6) level

In the case of *getsockopt()*, you can enter the following values for *optname* and *optlen* in the address family AF_INET6:

optname	***** *optlen*	**Output format of** *optval*
IPV6_HOPLIMIT	4	int
IPV6_MTU_DISCOVER	4	int
IPV6_MULTICAST_HOPS	4	int
IPV6_MULTICAST_IF	4	int
IPV6_MULTICAST_LOOP	4	int
IPV6_RECVERR	4	int
IPV6_RECVHOPLIMIT	4	int
IPV6_UNICAST_HOPS	4	int
IPV6_V6ONLY	4	int

Return value of IPV6_MTU_DISCOVER:

0: IP_PMTUDISC_DONT
2: IP_PMTUDISC_DO

Return value of IPV6_MULTICAST_HOPS, IPV6_UNICAST_HOPS and IPV6_HOPLIMIT:

Value of the selected hop limit.
The bind()-function has to be called before setting IPV6_MULTICAST_HOPS or IPV6_UNICAST_HOPS.

Return value of IPV6_MULTICAST_IF:

Index of the sender interface.
The bind()-function has to be called before setting this option.

Return value of IPV6_MULTICAST_LOOP:

0: Loopback OFF
1: Loopback ON

The bind()-function has to be called before setting this option.

Return value of IPV6_RECVERR, IPV6_RECVHOPLIMIT and IPV6_V6ONLY:

0: The flag is not set
1: The flag is set

In the case of *setsockopt(),* you can enter the following values for *optname* and *optlen* in the address family AF_INET6:

optname	*optlen*	*optval*
IPV6_JOIN_GROUP	>= sizeof(struct ipv6_mreq)	*(struct ipv6_mreq)
IPV6_LEAVE_GROUP	>= sizeof(struct ipv6_mreq)	*(struct ipv6_mreq)
IPV6_HOPLIMIT	4	1..255
IPV6_MTU_DISCOVER	4	IP_PMTUDISC_DO IP_PMTUDISC_DONT
IPV6_MULTICAST_HOPS	4	0..255
IPV6_MULTICAST_IF	4	1..255
IPV6_MULTICAST_LOOP	4	>= 0
IPV6_RECVERR	4	>= 0
IPV6_RECVHOPLIMIT	4	0, 1
IPV6_UNICAST_HOPS	4	0..255
IPV6_V6ONLY	4	>= 0

The IPV6_JOIN_GROUP and IPV6_LEAVE_GROUP options use a parameter of the data type *struct ipv6_mreq*. This parameter specifies the IPv6 address of the required multicast group and the index of the local interface.

The *ipv6_mreq* structure is defined in <netinet.in.h>:

```
struct ipv6_mreq {
        struct in6_addr ipv6mr_multiaddr;   /* IPv6 multicast addr */
        int    ipv6mr_interface;            /* interface index */
};
```

IPV6_HOPLIMIT

Modifies the hop limit in the corresponding field of the packet's IPv6 protocol Header.

IPV6_MULTICAST_HOPS

Displays or sets the multicast hop limit.
The bind()-function has to be called before setting this option.
Hop limit values:

0: Send only within the local host (loopback)
1: Send within the local subnetwork
>1: Send beyond router boundaries

IPV6_JOIN_GROUP

Activates the delivery of messages of a selected multicast group to this socket. Specifies the IPv6 multicast address and the index of the local interface address (index for IPv6 address or index 0, no index for loopback). Index 0 stands for the default interface of BCAM for receiving multicast data.
The bind()-function has to be called before setting this option.

IPV6_LEAVE_GROUP

Deactivates the delivery of messages of a selected multicast group to this socket. Specifies the IPv6 multicast address and the index of the local interface address (index for IPv6 address or index 0, no index for loopback). Index 0 stands for the default interface of BCAM for receiving multicast data.
The bind()-function has to be called before setting this option.

IPV6_MTU_DISCOVER

IP_PMTUDISC_DONT	Fragmentation permitted
IP_PMTUDISC_DO	Fragmentation is to be prevented

This function has no effect because on the IPv6 protocol level, in contrast to IPv4, no flag is envisaged which could prevent fragmentation. The end systems are obliged to perform fragmentation by the IPv6 protocol definition.

IPV6_MULTICAST_IF

Index of the IPv6 interface over which transfer is to take place.
The bind()-function has to be called before setting this option.

IPV6_MULTICAST_LOOP

Is set by the sender of the messages and enables reception on the local sending host.0: OFF, 1: ON (default: ON)
The bind()-function has to be called before setting this option.

IPV6_RECVERR

Activates the delivery of ICMP error messages to this socket if the option was set before *bind()*. The error reports for datagram and raw sockets are fetched with using recvmsg() and the flag MSG_ERRQUEUE.

IPV6_RECVHOPLIMIT

The hop limit field from every IPV6-packet received via *recvmsg()* is read out and saved in a *cmsghdr* structure.

IPV6_RECVHOPLIMIT is activated by setting *optval* = 1 and deactivated by setting *optval* = 0.
IPV6_RECVHOPLIMIT is deactivated by default.

IPV6_UNICAST_HOPS

Sets the unicast hop limit.
The bind()-function has to be called before setting this option.
Hop limit values:

0: Send only within the local host (loopback)
1: Send within the local subnetwork
>1: Send beyond router boundaries

IPV6_V6ONLY

Using the IPV6_V6ONLY option, it is possible to restrict a socket to the use of genuine IPv6 addresses (*optval* >=1) if it is set ahead of *bind()* in the socket. This makes it possible to provide server applications which open a listen socket in the domains AF_INET and AF_INET6 using the same port number.

Options for the IPPROTO_TCP (AF_INET, AF_INET6) level

Calling the *getsockopt()* function you can specify the following values for *optname* and *optlen* in the AF_INET and AF_INET6 address families:

optname	***** *optlen*	Output format in *optval*
SO_TCP_NODELAY	4	int

The *setsockopt()* function can be used to modify option values via the *optval()* and *optlen()* parameters. You can specify the following values in the AF_INET and AF_INET6 address families:

optname	*optlen*	*optval*
SO_TCP_NODELAY	4	1 or 0 (reset/set)
TCP_DELAY	4	1 or 0 (reset/set)

SO_TCP_NODELAY (TCP_NODELAY)

Allows the Nagle algorithm of the TCP protocol to be deactivated. If this option is set in the socket *(connect()* or *listen()*), the action is activated on *connect()* or when the connection is acknowledged.
If the connection has already been established, the option takes effect immediately.

TCP_DELAY

Enable or disable the Delayed-Ack timer for a connection. If *optval >0*, delayed acknowledgements are disabled and if *optval = 0*, they are enabled again.
The connection must have been established.

Options for the IPPROTO_ICMP (AF_INET) level

In the case of *setsockopt()*, you can modify option values using the *optval()* and *optlen()* parameters. You can specify the following values in the AF_INET address family:

optname	*optlen*	*optval*
IP_TTL	4	1...255
IP_MTU_DISCOVER	4	IP_PMTUDISC_DO IP_PMTUDISC_DONT

IP_TTL

Modifies the hop limit in the corresponding field of the ICMP echo request packet's IP protocol header.

IP_MTU_DISCOVER

Sets the DF flag in the IPv4 protocol header which permits or prevents fragmentation of an ICMP echo request packet.

IP_PMTUDISC_DONT	Fragmentation permitted
IP_PMTUDISC_DO	Fragmentation is to be prevented

Options for the IPPROTO_ICMPV6 (AF_INET6) level

In the case of *setsockopt()*, you can modify option values using the *optval()* and *optlen()* parameters. You can specify the following values in the AF_INET6 address family:

optname	*optlen*	**Output format in** *optval*
IPV6_HOPLIMIT	4	1...255
IPV6_MTU_DISCOVER	4	IP_PMTUDISC_DO IP_PMTUDISC_DONT

IPV6_HOPLIMIT

Modifies the hop limit in the corresponding field of the ICMPv6 echo request packet's IPv6 protocol header.

IPV6_MTU_DISCOVER

IP_PMTUDISC_DONT	Fragmentation permitted
IP_PMTUDISC_DO	Fragmentation is to be prevented

This function has no effect because on the IPv6 protocol level, in contrast to IPv4, no flag is envisaged which could prevent fragmentation. The end systems are obliged to perform fragmentation by the IPv6 protocol definition.

Options for the SOL_TRANSPORT level (only for AF_ISO)

In this case, *s* specifies the socket for which the options are to be retrieved or set, and *optname* specifies the name of the option for which a value is to be retrieved or set.

With *getsockopt()*, *optval* and *optlen* identify the respective buffers in which the value of the desired option is returned. **optlen* initially contains the size of the buffer to which *optval* points. When the *getsockopt()* function returns, **optlen* contains the current size of the returned buffer. If the option in question has no value that can be returned, **optval* contains the value 0.

The following values can be returned by *getsockopt()* for *optname* and **optlen* in the AF_ISO address family:

optname	***** *optlen*	**Output format in** *optval*
TPOPT_CONN_DATA	0..33	string incl. null byte (length as specified in *optlen*)
TPOPT_CFRM_DATA	0..33	string incl. null byte (length as specified in *optlen*)
TPOPT_DISC_DATA	0..33	string incl. null byte (length as specified in *optlen*)
TPOPT_REDI_CALL	sizeof(struct cmsg_redhdr)	struct cmsg_redhdr

The actual length of the connection data is specified in **optlen*. The value range for *optlen* is 0..33 since the maximum length of the connection data is 32 bytes.

setsockopt() can be used in the AF_ISO address family to write connection data to the socket depending on the socket status. The options described below are known on the socket level.

Description of the socket options for AF_ISO:

TPOPT_CONN_DATA

The socket *s* has been created and has received an address with *bind()*, but *connect()* has not yet been executed:
You can enter the connection data to be sent to the partner when *connect()* is called in the socket *s* by using TPOPT_CONN_DATA as the current parameter value for *optname*.

TPOPT_CFRM_DATA

A connection request is received for the socket *s*, and the socket *s* has not yet accepted it:
You can enter the connection data to be sent for accepting the connection to the partner in the socket *s* by using TPOPT_CFRM_DATA as the current parameter value for *optname* (see also figure 4 in section "Interaction between functions for connection-oriented communications").

TPOPT_DISC_DATA

The connection to the partner socket has been set up completely:
You can enter the connection data to be sent to the partner for *soc_close()* in the socket *s* by using TPOPT_DISC_DATA as the current parameter value for *optname*.

The actual length of the connection data is specified in **optlen*. The value range for *optlen* is 1..32 since the maximum length of the connection data is 32 bytes and the minimum length is 1 byte.

TPOPT_REDI_CALL

This is required for the handoff procedure (see chapter "Extended SOCKETS(BS2000) functions").

Return value

0:

If successful.

-1:

If errors occur. *errno* is set to indicate the error.

Errors indicated by *errno*

ENOPROTOOPT

The option is not supported by the protocol, or
an invalid value was specified for *level*, *optname*, *optvalue* or *optlen*.

ENOTSOCK

Descriptor *s* does not point to a socket.

See also

socket()

if_freenameindex() - release the dynamic storage occupied with if_nameindex()

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

void if_freenameindex(struct if_nameindex *ptr);
```

Description

The function *if_freenameindex()* is required in order to release the storage which is dynamically requested for the return informationfrom *if_nameindex()*.

if_indextoname() - convert interface index to interface name

```
#include <sys.socket.h>
#include <net.if.h>

char * if_indextoname(unsigned int ifindex, char * ifname);
```

Description

The function *if_indextoname()* returns the interface name corresponding to the specified interface index. To achieve this, a pointer to a buffer of minimum length IF_NAMESIZE (present in <net.if.h>) is supplied in *ifname*.

Return value

If execution is successful, *if_indextoname()* returns the interface name stored in *ifname*. Otherwise a NULL pointer is returned

Errors indicated by ***errno***

ENXIO

No interface name corresponding to the specified index was found.

if_nameindex() - list of interface names with the associated interface indexes

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

struct * if_nameindex if_nameindex(void);
```

Description

The function *if_nameindex()* generates an array consisting of the interface names and the associated interface index.

For each interface present, an *if_nameindex* structure is created.
The *if_nameindex* structure is declared in b<net.if.h> as follows:

```
struct if_nameindex {
   unsigned int       if_index;       /*1, 2, .....*/
   char *             if_name;        /* name terminated with null byte*/
};
```

Return value

An array with structures of type *if_nameindex* is returned as the result. The end of the array is indicated by the fact that the last *if_nameindex* structure contains the values 0 for *if_index* and NULL for *if_name*.

If an error occurs then a NULL pointer is returned and *errno* is set accordingly

Errors indicated by *errno*

EINVAL

No interface information is available

Note

The storage required for the array is requested dynamically and must be released again using the function *if_freenameindex()*.

if_nametoindex() - convert interface name to interface index

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

unsigned int if_nametoindex(const char * ifname);
```

Description

The function *if_nametoindex()* returns the interface index corresponding to the specified interface name. *ifname* is a null-terminated string containing the interface name.

Return value

If execution is successful, *if_nametoindex()* returns the interface index. Otherwise 0 is returned

Errors indicated by ***errno***

No errors are defined.

inet_addr(), inet_lnaof(), inet_makeaddr(), inet_netof(), inet_network(), inet_ntoa() - manipulate IPv4 Internet address

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


Kernighan-Ritchie-C:
unsigned long inet_addr(cp);
char *cp;

int inet_lnaof(in);
struct in_addr in;

struct in_addr inet_makeaddr(net, lna);
int net;
int lna;

int inet_netof(in);
struct in_addr in;

unsigned long inet_network(cp);
char *cp;

char *inet_ntoa(in);
struct in_addr in;

ANSI-C:
unsigned long inet_addr(char* cp);
int inet_lnaof(struct in_addr in);
struct in_addr inet_makeaddr(int net, int lna);
int inet_netof(struct in_addr in);
unsigned long inet_network(char* cp);
char* inet_ntoa(struct in_addr in);
```

Description

Use of the *inet_addr()*, *inet_lnaof()*, *inet_makeaddr(), inet_netof() inet_network()* and *inet_ntoa()* functions only makes sense in the AF_INET address family.

- The *inet_addr()* function converts the character string to which the *cp* parameter points from the normal Internet dotted notation to an integer value which can then be used as the Internet address.
- The *inet_lnaof()* function extracts the local network address in the byte order of the host from the Internet host address passed in the *in* parameter.
- The *inet_makeaddr()* function creates an Internet address from the following
  - the subnetwork section of the Internet address specified in the *net* parameter and
  - the subnetwork local address section specified in the *lna* parameter.

  The subnetwork section of the Internet address and subnetwork local address section are both passed in the byte order of the host.

- The *inet_netof()* function extracts the network number in the byte order of the host from the Internet host address passed in the *in* parameter.
- The *inet_network()* function converts the character string to which pointer *cp* points from the normal Internet dotted notation to an integer value which can then be used as the subnetwork section of the Internet address.
- The *inet_ntoa()* function converts the Internet host address passed in the *in* parameter into a character string in the normal Internet dotted notation.

All Internet addresses are returned in network byte order in which the bytes are arranged from left to right.

Values can be specified in the following dotted notation formats:

- a.b.c.d
  If a four-part address is specified, each part is interpreted as one data byte and assigned from left to right to the four bytes of an Internet address.
- a.b.c
  If a three-part address is specified, the last part is interpreted as a 16-bit sequence and transferred to the two right bytes of the Internet address. This allows three-part address formats to be used without problems for specifying class B addresses
  (e.g. 128.*net*.*host*).
- a.b
  If a two-part address is specified, the last part is interpreted as a 24-bit sequence and transferred to the right three bytes of an Internet address. This allows two-part address formats to be used without problems for specifying class A addresses (e.g. *net.host*).
- a
  If a single-part address is specified, the value is transferred without changing the byte order directly to the network address.

The numbers specified as address parts in dotted notation may be either decimal, octal or hexadecimal numbers:

- Numbers not prefixed with either 0, 0x or 0X are interpreted as decimal numbers.
- Numbers prefixed with 0 are interpreted as octal numbers.
- Numbers prefixed with 0x or 0X are interpreted as hexadecimal numbers.

Return value

- After successful execution, *inet_addr()* returns the Internet address.
  Otherwise, -1 is returned.
- After successful execution, *inet_network()* returns the subnetwork portion of the Internet address. Otherwise, -1 is returned.
- The *inet_makeaddr()* function returns the created Internet address.
- The *inet_lnaof()* returns the local network address.
- The *inet_netof()* function returns the network number.
- The *inet_ntoa()* returns a pointer to the network address in the normal Internet dotted notation.

Errors indicated by *errno*

No errors are defined.

Note

The return value of *inet_ntoa()* points to static data that may be overwritten by subsequent *inet_ntoa()* calls. This information must therefore be copied if it needs to be saved.

inet_ntop(), inet_pton() - manipulate Internet addresses

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


Kernighan-Ritchie-C:
char *inet_ntop(af, addr, dst, size);

int af;
char *addr;
char *dst;
int size;


int inet_pton(af, addr, dst);

int af;
char *addr;
char *dst;

ANSI-C:
char* inet_ntop(int af, char* addr, char* dst, int size);
int inet_pton(int af, char* addr, char* dst);
```

Description

The use of the *inet_ntop()* and *inet_pton()* functions only makes sense in the AF_INET and AF_INET6 address families.

The *inet_ntop()* function converts the binary IP address to which the *addr* parameter is pointing to printable notation. The value passed in the *af* parameter indicates whether the address involved is an IPv4 address or an IPv6 address:

- If the value AF_INET is passed in *af*, a binary IPv4 address is converted to printable decimal dotted notation.
- If the value AF_INET6 is passed in *af*, a binary IPv6 address is converted to printable hexadecimal colon notation.

inet_ntop() returns the printable address in the buffer of the length *size* to which the pointer *dst* is pointing. You can ensure that the buffer is big enough by using the integer constant INET_ADDRSTRLEN (for IPv4 addresses) or INET6_ADDRSTRLEN (for IPv6 addresses) as the current value for *size* when you call *inet_ntop()*. Both constants are defined in <netinet.in.h>.

The *inet_pton()* function converts an IPv4 address in decimal dotted notation or an IPv6 address in hexadecimal colon notation to a binary address. The value passed in the *af* parameter indicates whether the address involved is an IPv4 address or an IPv6 address:

- If the value AF_INET is passed in *af*, an IPv4 address is converted.
- If the value AF_INET6 is passed in *af*, an IPv6 address is converted.

inet_pton() returns the binary address to the buffer to which the pointer *dst* is pointing. The buffer must be sufficiently large: 4 bytes for AF_INET and 16 bytes for AF_INET6. The conversion of IPv6 addresses in abbreviated notation with two colons (“::”) is not supported.

Note

If the output of *inet_pton()* is used as the input for a new function, make sure that the starting address of the destination area *dst* has doubleword alignment.

Return value

If the *inet_pton()* function is executed successfully, it returns a pointer to the buffer in which the text string is stored. The null pointer is returned if an error occurs.

inet_ntop() returns the following values:

1:

If conversion is successful.

0:

If the input is an invalid address string.

-1:

If a parameter is invalid.

Errors indicated by *errno*

EAFNOSUPPORT

Illegal operand.

ENOSPC

The result buffer is too small.

listen() - test a socket for pending connections

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


Kernighan-Ritchie-C:
int listen(s, backlog);

int s;
int backlog;

ANSI-C:
int listen(int s, int backlog);
```

Description

The *listen()* function is supported in the AF_INET and AF_INET6 address families (only for sockets of the type SOCK_STREAM), and in the AF_ISO address family.

The *listen()* function authorizes socket *s* to accept connection requests and then tests the socket for pending connection requests. To do this, *listen()* sets up a queue for incoming connection requests for socket *s*.

The user can define the maximum number of connection requests that the queue can hold by using the *backlog* parameter.
Note, however, that SOCKETS(BS2000) does not evaluate the *backlog* parameter at present and continues to accept connection requests until the maximum number of available sockets have been used.

The following steps are required to enable a task to communicate on a socket with the partner that sends connection requests:

1. Create a socket (*socket()*) and bind it (*bind()*)
2. Specify an incoming connection request queue for the socket with *listen()*.
3. Accept the connection requests with *accept()*.
4. In AF_ISO, you also have to send user data or confirm data (CFRM data, *sendmsg()*) (see also figure 4 in section "Interaction between functions for connection-oriented communications").

You can only use *connect()* to initiate connections between two sockets of the AF_ISO address family, or between two sockets of the AF_INET address family or AF_INET6 address family, or between a socket of the AF_INET address family and a socket of the AF_INET6 address family.

Therefore, if you are using all the address families, you should set up a *listen()* socket of the AF_ISO address family, as well as a *listen()* socket of the AF_INET or AF_INET6 address family. This ensures that connections to every supported address family can be set up.

Return value

0:

If successful.

-1:

If errors occur. *errno* is set to indicate the error.

Errors indicated by *errno*

EBADF

The *s* parameter is not a valid descriptor.

EISCONN

The socket already has a connection.

EOPNOTSUPP

The socket is not of type SOCK_STREAM and is not supported by *listen()*.

See also

accept(), connect(), socket()

recv(), recvfrom() - receive a message from a socket

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

#include <netinet.in.h> /* AF_INET, AF_INET6 and connectionless operation*/


Kernighan-Ritchie-C:
int recv(s, buf, len, flags);

int s;
char *buf;
int len;
int flags;


int recvfrom(s, buf, len, flags, from, fromlen);

int s;
char *buf;
int len;
int flags;

struct sockaddr_in *from; /* AF_INET */
struct sockaddr_in6 *from; /* AF_INET6 */
int *fromlen;

ANSI-C:
int recv(int s, char* buf, int len, int flags);
int recvfrom(int s, char* buf, int len, int flags, struct sockaddr* from, int*
fromlen);
```

Description

The *recv()* and *recvfrom()* functions receive messages from a socket. The function *recvmsg()* is necessary, if messages are to be received from a raw socket.

recv() can only receive messages from a socket on which a connection has already been set up (see the *connect()* function in section "connect() - initiate a connection on a socket").

recvfrom() can receive messages from a socket with or without a connection.
The function call *recvfrom()* with *from* != null pointer and *fromlen* != null pointer is only supported for datagrams.

The *s* parameter designates the socket from which the message is received.
buf specifies the storage area in which the data is to be received.
len specifies the length of this buffer.

If the *from* parameter is not the null pointer (connectionless operation), the address of the message sender is stored in the address area referenced by *from*.

fromlen is a result parameter. Before the function is called, the integer variable to which *fromlen* points must contain the size of the buffer referenced by *from*. After the function returns, **fromlen* contains the current length of the address stored in **from*.
The function returns the length of the message.

The *flags* parameter is currently only supported with a datagram socket with the MSG_PEEK flag in the AF_INET and AF_INET6 address families. In the other cases *flags* should be supplied with the value 0.
MSG_PEEK allows data to be read without it being deleted at the source. This means that a repeat read operation is necessary.

The complete message must be read in a single operation with a datagram socket (only AF_INET, AF_INET6). If the specified message buffer is too small, the data extending beyond the buffer size is deleted.

Message limits are ignored with a stream socket (AF_INET, AF_INET6). As soon as data is available, it is returned to the caller, and no data is deleted.

Message limits are observed for a socket belonging to the AF_ISO address family. As soon as data is available, it is returned to the caller, and no data is deleted.

If no messages are available on the socket, the receive call waits for an incoming message unless the socket is non-blocking (see *soc_ioctl()*). In this case, -1 is returned, and the *errno* variable is set to the value EWOULDBLOCK.

The *select()* function can be used to determine when further data arrives.

Return value

>0:

If successful. The value indicates the number of received bytes.

=0:

If successful.
No more data can be received by sockets of type SOCK_STREAM or sockets belonging to the AF_ISO address family. The partner has closed his connection.

Sockets of type SOCK_DGRAM receive a data packet with the length 0 or the data is deleted by the transport system for a timeout.

-1:

If errors occur. *errno* is set to indicate the error.

Errors indicated by *errno*

EBADF

The *s* parameter is not a valid descriptor.

ECONNRESET

The connection to the partner was interrupted, although there were still some data packets expected (only with sockets of type SOCK_STREAM).

EFAULT

The length of the area for accepting the address is too small.

EIO

User data has been lost.

ENETDOWN

The connection to the network is down.

ENOTCONN

No connection exists for the socket.

EOPNOTSUPP

- The *flags* parameter contains a non-zero value.

or

- The socket is not of type SOCK_STREAM, and *recv()* supports only stream sockets.

EPIPE

There is no connection to the partner (only with sockets of type SOCK_STREAM).

EWOULDBLOCK

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

See also

connect(), getsockopt(), select(), send(), soc_ioctl(), soc_read(), soc_readv(), socket()

recvmsg() - receive a message from a socket

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


Kernighan-Ritchie-C:
int recvmsg(s, msg, flags);

int s;
flags struct msghdr *msg;

ANSI-C:
int recvmsg(int s, struct msghdr* msg, int flags);
```

Description

The *recvmsg()* function is supported in the AF_INET, AF_INET6 and AF_ISO address families and provides the following functionality depending on the parameterization (*msg* parameter):

- *recvmsg()* can be used to receive user data from the partner socket on socket *s*.
- Only in AF_ISO: *recvmsg()* can be used to read connection data from socket *s*.

A pointer to an object of the data type *struct msghdr* must be specified as the current parameter for *msg*. The desired functionality for *recvmsg()* is selected via the component *msg->msg_control* (data type *caddr_t* or *char* *)

- If *msg->msg_control* is the null pointer, user data is received.
- Only in AF_ISO: If *msg->msg_control* is not the null pointer, *msg->msg_control* is interpreted as a pointer to a storage area with the structure *cmsghdr*, and connection data is read from the socket.

  Due to the internal socket status for *s*, *recvmsg()* selects the connection data type (CONN_DATA, CFRM_DATA or DISC_DATA) and writes the relevant data in the data area of the *msg->msg_control* pointer (see *msghdr* and *cmsghdr* structures on the next page).

  TPOPT_REDI_DATA and TPOPT_REDI_BDOK are provided for the handoff procedure. In this case, the structure *cmsg_redhdr* is required. Refer to chapter "Extended SOCKETS(BS2000) functions" for a description.

msghdr structure

The *msghdr* structure is declared in <sys.socket.h> as follows:

```
struct msghdr {
   caddr_t       msg_name;           /* destination address */
   int           msg_namelen;        /* length of the destination address */
   struct iovec *msg_iov;            /* scatter/gather fields */
   int           msg_iovlen;         /* number of elements in msg_iov */
   caddr_t       msg_control;        /* auxiliary data*/
   int           msg_controllen;     /* length of the buffer for */
                                     /* auxiliary data */
   int           msg_flags;          /* flag for received message */
};
struct msghdr *msg;
```

msg->msg_name and *msg->msg_namelen* can return different information in connectionless mode:
1: The flag MSG_ERRQUEUE is set and an error packet is received: Destination address and its length
2: The flag MSG_ERRQUEUE is set and no error packet is received: No information is written to both fields and *recvmsg()* returns with -1
3: The flag MSG_ERRQUEUE is not set and an error packet is received: No information is written to both fields
4: The flag MSG_ERRQUEUE is not set and no error packet is received: Address of the sender of the just received packet and its length

If msg_name is not null then the content is interpreted as the pointer to a buffer in which the partner's address information is entered. msg_namelen specifies the length of this buffer. If the socket is "non-connected" then the sender's address information is stored using a *sockaddr* structure and *msg_namelen* contains the length of this structure.

If these parameters are not to be used, *msg_name* should have the value of the null pointer and *msg_namelen* the value 0.
msg->msg_iov is a pointer to a storage area with objects of the type *struct iovec*. *msg->msg_iovlen* indicates the number of elements (max. 16) in this storage area.*msg->msg_control* is a pointer to an object of the type *struct cmsghdr* in which *recvmsg()* enters the expected connection data.
msg->msg_controllen indicates the length of **msg->msg_control*.
msg->msg_flags = MSG_EOR indicates the end of a record (only AF_ISO).

iovec structure

The *iovec* structure is declared in <sys.uio.h> as follows:

```
struct iovec{
    caddr_t  iov_base;  /* buffer for auxiliary data */
    int      iov_len;    /* buffer length */
};
```

cmsghdr structure

The *cmsghdr* structure is declared in <sys.socket.h> as follows:

```
struct cmsghdr {
   u_int     cmsg_len;           /* number of data bytes incl. header */
   int       cmsg_level;         /* generating protocol */
   int       cmsg_type;          /* protocol-specific type */
   /* followed by u_char cmsg_data[] */
};
struct cmsghdr *cmsg;
```

cmsg->cmsg_len contains the length of the storage area of **msg->msg_control*.
cmsg->cmsg_level describes the type of information included in the data area by stating a specific protocol level.
cmsg->cmsg_type gives additional information about the contents of the data area with an options value.
Connection data and the final null byte are entered in the data portion of the *cmsghdr.* It can be accessed with the CMSG_DATA macro.

The *flags* Parameter currently supports the following flags:

- MSG_PEEK (only for datagram sockets):

  MSG_PEEK allows the data to be read without it being deleted at the source. This means that a repeat read operation is necessary.

- MSG_ERRQUEUE (only for datagram and raw sockets):

  MSQ_ERRQUEUE provides more detailed information in case there is an error. The data is written into msg_iov and the error protocol into msg_control.

Return value

>0:

If successful:
Number of bytes of the received user data

=0:

- only AF_ISO
  for connection data (CONN_DATA, CFRM_DATA, DISC_DATA)
- AF_INET, AF_INET6
  No more data can be received. The partner has closed his connection correctly (only for sockets of the type SOCK_STREAM).

  Sockets of type SOCK_DGRAM receive a data packet with the length 0 or the data is deleted by the transport system for a timeout.

-1:

If errors occur. *errno* is set to indicate the error.

Errors indicated by *errno*

EBADF

The *s* parameter is not a valid descriptor.

ECONNRESET

The connection to the partner was interrupted.

EFAULT

The flag MSG_ERRQUEUE was set and no error package is available

EINVAL

A parameter specifies an invalid value.

ENETDOWN

The connection to the network is down.

ENOTCONN

No connection exists for the socket.

EOPNOTSUPP

The function call includes illegal attributes

EPIPE

The partner has interrupted the connection.

EWOULDBLOCK

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

select() - multiplex input/output

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


Kernighan-Ritchie-C:
int select(nfds, readfds, writefds, exceptfds, timeout);

int nfds;
fd_set *readfds, *writefds, *exceptfds;
struct timeval *timeout;


FD_SET(fd, &fdset);

FD_CLR(fd, &fdset);

FD_ISSET(fd, &fdset);

FD_ZERO(&fdset);

int fd;
fd_set fdset;

ANSI-C:
int select(int nfds, fd_set* readfds, fd_set* writefds, fd_set* exceptfds, struct
timeval* timeout);

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

int fd;
fd_set fdset;
```

Description

The *select()* function tests three different sets of socket descriptors passed with the *readfds*, *writefds* and *exceptfds* parameters.
select() determines

- which descriptors in the set passed with *readfds* are ready for reading,
- which descriptors in the set passed with *writefds* are ready for writing,
- for which descriptors in the set passed with *exceptfds* a pending exception exists.

The *exceptfds* parameter is not evaluated by SOCKETS(BS2000) at present.

The bit masks for the individual descriptor sets are stored as bit fields in integer strings. The maximum size of the bit fields should be determined by using the *getdtablesize()* function (see section "getdtablesize() - get size of descriptor table"). The required memory should be requested from the system dynamically.

The *nfds* parameter specifies how many bits are to be tested in each bit mask. *select()* tests bits 0 to *nfds*-1 in the individual bit masks.
select() replaces the descriptor sets passed in the call with the appropriate subsets. These subsets include all the respective descriptors that are ready for the operation involved.

You can use the following macros to manipulate bit masks or descriptor sets:

FD_SET(fd, &fdset)

Extends the descriptor set *fdset* by descriptor *fd*.

FD_CLR(fd, &fdset)

Removes descriptor *fd* from descriptor set *fdset*.

FD_ISSET(fd, &fdset)

Tests whether descriptor *fd* is a member of descriptor set *fdset*:

- Return value != 0: *fd* is a member of *fdset*.
- Return value == 0: *fd* is not a member of *fdset*.

FD_ZERO(&fdset)

Removes all file descriptors from descriptor set *fdset*.

The behavior of these macros is undefined if the descriptor value is <0 or greater than the maximum size for bit fields that was determined with the *getdtablesize()* function.

The *timeout* parameter defines the maximum time that the *select()* function has to complete the selection of the ready descriptors. If *timeout* is the null pointer, *select()* blocks for an undefined time. The maximum time value that will be accepted is ~ 24.85 days. Values bigger than that will lead to waiting until an event occurs.
You can enable polling by passing a pointer for *timeout* to a *timeval* object whose components all have the value 0.

If the descriptors are of no interest, the null pointer can be passed as the current parameter for *readfds*, *writefds* and *exceptfds*.

If *select()* determines the “read” readiness of a socket descriptor after calling *listen()*, this indicates that a subsequent *accept()* call for this descriptor will not block.

Return value

>0:

The positive number indicates the number of ready descriptors in the descriptor set.

0:

Indicates that the timeout limit has been exceeded. The descriptor sets are then undefined.

-1:

If errors occur. *errno* is set to indicate the error. The descriptor sets are then undefined.

7F000000:

Trace event for user sockets trace.

Errors indicated by *errno*

EBADF

One of the descriptor sets specified an invalid descriptor.

EINTR

The *select()* call was interrupted by *soc_wake()*.

ENETDOWN

The connection to the network is down.

> When virtual hosts are used, ENETDOWN does not necessarily mean that the entire network is down. It can also mean that only the network of a virtual host has failed.

Note

In rare circumstances, *select()* may indicate that a descriptor is ready for writing, although a write attempt would actually block. This can occur if the system resources required for writing are exhausted or not present. If it is critical for your application that writes to a file descriptor do not block, you should set the descriptor to non-blocking input/output with a *soc_ioctl()* call.

See also

accept(), connect(), listen(), recv(), send(), soc_ioctl(), soc_write(), soc_writev()

send(), sendto() - send a message from socket to socket

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

#include <netinet.in.h> /* AF_INET, AF_INET6 and connectionless operation */


Kernighan-Ritchie-C:
int send(s, msg, len, flags);

int s;
char *msg;
int len, flags;


int sendto(s, msg, len, flags, to, tolen);

int s;
char *msg;
int len, flags;

struct sockaddr_in *to; /* AF_INET */
struct sockaddr_in6 *to; /* AF_INET6 */
int tolen;

ANSI-C:
int send(int s, char* msg, int len, int flags);
int sendto(int s, char* msg, int len, int flags, struct sockaddr* to,int tolen);
```

Description

The *send()* and *sendto()* functions send messages from one socket to another. *send()* can only be used with a socket on which a connection with *connect()* has been set up (see the *connect()* function in section "connect() - initiate a connection on a socket").

sendto() can also be used during connectionless operation. The function call *sendto()* with *to* î null pointer and *tolen* î 0 is only supported for datagrams.

The *s* parameter designates the socket from which a message is sent. The destination address is passed with *to*, where *tolen* specifies the length of the destination address.

The length of the message is specified with *len*. If the message is too long to be transported completely by the underlying protocol level, error SCEMSGSIZE is returned and the message is not transferred for datagram sockets (i.e. only AF_INET and AF_INET6).

The *flags* parameter is currently not supported and should be supplied with the value 0. A value not equal to 0 leads to an error, and the *errno* variable is set to the value EOPNOTSUPP.

If the message cannot be sent immediately, *send()* blocks if the socket was not set to the non-blocking input/output mode. You can use *select()* to determine when further data can be sent.

Return value

>=0:

If successful. The value indicates the number of sent bytes.

-1:

If errors occur. *errno* is set to indicate the error.

Errors indicated by *errno*

EBADF

The *s* parameter is not a valid descriptor.

EFAULT

The length of the area for accepting the address is too small, or the length of the area for the message is too small.

EIO

I/O error. The message could not be passed to the transport system.

EMSGSIZE

The message is too long to be sent in one piece.

ENETDOWN

The connection to the network is down.

ENOTCONN

No connection exists for the socket. A read/write attempt was rejected.

EOPNOTSUPP

- The *flags* parameter was specified with a non-zero value, and this is not supported.

or

- The socket is not of type SOCK_STREAM, and the operation is supported only for stream sockets.

EPIPE

There is no connection to the partner (only with sockets of type SOCK_STREAM).

EWOULDBLOCK

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

Note

If the connection is established with a non-blocking socket then *errno* EINPROGRESS may occur on the next function call. This message indicates that the connection is not yet in a state which permits a data transfer phase.

See also

connect(), getsockopt(), recv(), select(), soc_ioctl(), soc_write(), socket()

sendmsg() - send a message from socket to socket

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


Kernighan-Ritchie-C:
int sendmsg(s, msg, flags);

int s, flags;
struct msghdr *msg;

ANSI-C:
int sendmsg(int s, struct msghdr* msg, int flags);
```

Description

The *sendmsg()* function is supported in the AF_INET, AF_INET6 and AF_ISO address families and provides the following functionality depending on the parameterization(*msg* parameter):

- *sendmsg()* can be used to send user data from a socket to a partner socket.
- Only in AF_ISO: *sendmsg()* can be used to write connection data to socket *s*.

A pointer to an object of the data type *struct msghdr* must be specified as the current parameter for *msg*. The desired functionality for *sendmsg()* is selected via the component *msg->msg_control* (data type *caddr_t* or *char* *)

- If *msg->msg_control* is the null pointer, user data is sent.
- Only in AF_ISO: If *msg->msg_control* is not the null pointer, *msg->msg_control* is interpreted as a pointer to a storage area with the structure *cmsghdr*, and connection data is written to the socket.

  This allows *sendmsg()* to send an acknowledgment of the connection request to the communications partner without transferring user data or connection data.

msghdr structure

The *msghdr* structure is declared in <sys.socket.h> as follows:

```
struct msghdr {
   caddr_t       msg_name;           /* destination address */
   int           msg_namelen;        /* length of the destination address */
   struct iovec *msg_iov;            /* scatter/gather fields */
   int           msg_iovlen;         /* number of elements in msg_iov */
   caddr_t       msg_control;        /* auxiliary data */
   int           msg_controllen;     /* length of the buffer for */
                                     /* auxiliary data */
   int           msg_flags;          /* flag for received message */
};
struct msghdr *msg;
```

msg->msg_name and *msg->msg_namelen* are only interpreted in the AF_INET and AF_INET6 address families with the socket type SOCK_DGRAM. *msg_name* indicates the address of a socket address structure, and *msg_namelen* indicates the length of this address structure. If these parameter are not to be used, *msg_name* should have the value of the null pointer and *msg_namelen* the value 0.
msg->msg_iov is a pointer to a storage area with objects of the type *struct iovec*. *msg->msg_iovlen* indicates the number of elements (max. 16) in this storage area.*msg->msg_control* is a pointer to an object of the type *struct cmsghdr* which must be supplied with the connection data to be written before *sendmsg()* is called (only AF_ISO). *msg->msg_controllen* indicates the length of **msg->msg_control*.
In *msg->msg_flags*, *sendmsg()* indicates the end of a record with MSG_EOR (only AF_ISO).

iovec structure

The *iovec* structure is declared in <sys.uio.h> as follows:

```
struct iovec{
    caddr_t  iov_base;  /* buffer for auxiliary data */
    int      iovlen;    /* buffer length */
};
```

cmsghdr structure

The *cmsghdr* structure is declared in <sys.socket.h> as follows:

```
struct cmsghdr {
   u_int     cmsg_len;           /* number of data bytes incl. header */
   int       cmsg_level;         /* generating protocol */
   int       cmsg_type;          /* protocol-specific type * */
   /* followed by u_char cmsg_data[] */
};
struct cmsghdr *cmsg;
```

cmsg->cmsg_len contains the length of the storage area of **msg->msg_control*. SOL_TRANSPORT is entered for the ISO transport service in *cmsg->cmsg_level*. *cmsg->cmsg_type* indicates the connection data type (TPOPT_CONN_DATA, TPOPT_CFRM_DATA, TPOPT_DISC_DATA).
Connection data and the final null byte are entered in the data area of the *cmsghdr.*

TPOPT_REDI_DATA and TPOPT_REDI_BDOK are provided for the handoff procedure. In this case, the structure *cmsg_redhdr* is required. Refer to chapter "ExtendedSOCKETS(BS2000) functions" for a description.

Return value

>=0:

Number of bytes of user data sent.

AF_ISO: 0 for connection data (CONN_DATA, CFRM_DATA, DISC_DATA)

-1:

If errors occur. *errno* is set to indicate the error.

Errors indicated by *errno*

EBADF

The *s* parameter is not a valid descriptor.

ECONNRESET

The connection to the partner was interrupted.

EINVAL

A parameter specifies an invalid value.

EIO

I/O error. The message could not be passed to the transport system.

ENETDOWN

The connection to the network is down.

ENOTCONN

No connection exists for the socket.

EOPNOTSUPP

The function call includes illegal attributes

EPIPE

The partner has interrupted the connection.

EWOULDBLOCK

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

shutdown() - terminate full-duplex connection

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


Kernighan-Ritchie-C:
int shutdown(s, how);

int s, how;

ANSI-C:
int shutdown(int s, int how);
```

Description

The *shutdown()* function limits the functionality of the socket and terminates the connection completely or in part. However, the socket still remains. The socket can be closed with the *soc_close()* function.

The *shutdown()* function is supported in the AF_INET and AF_INET6 address families.

The *how* parameter controls how the connection belonging to socket *s* should be terminated. The following values of *how* can be used:

SHUT_RD:

Read access is not possible for the socket, i.e. a read function can no longer be executed. This functionality can cause problems in the application because the partner socket is not informed of this limitation.

> **i** If the partner socket continues to send data, this can lead to a jam situation: The sent data uses memory in the transport system and these resources cannot be released because the data has not been fetched by the receiver. If the memory resources are used up, data can also no longer be sent.

SHUT_WR:

Write access is not possible for the socket. The partner socket is informed that data can no longer be sent from this socket. This corresponds to a “graceful disconnect”.

SHUT_RDWR:

Read and write access are not possible for the socket. The partner socket is informed that data cannot be sent or read. This is corresponds to an “abortive disconnect”.

Return value

0:

If successful

-1:

If errors occur. *errno* is set to indicate the error.

Errors indicated by *errno*

EBADF

The *s* parameter is not a valid descriptor.

ENOTCONN

No connection exists for the socket.

Note

Up to SOCKETS(BS2000) < V.2.1 the *shutdown()* function was supported without functionality, i.e. the call was not rejected, however no action was executed.

The functionality described above is provided if the user program has been compiled with the user library of SOCKETS(BS2000) as of version 2.1.

See also

soc_close()

soc_close() (close) - close socket

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


Kernighan-Ritchie-C:
int soc_close(s);

int s;

ANSI-C:
int soc_close(int s);
```

Description

The exact functionality of *soc_close()* is determined by the address family used.

soc_close() for AF_INET and AF_INET6

soc_close() closes socket *s* depending on the SO_LINGER option (see the *setsockopt()* function in section "getsockopt(), setsockopt() - get and set socket options").

If *soc_close()* is used with the SO_LINGER option, *soc_close()* will try to shut down the connection within the time specified by SO_LINGER after sending all pending data.

soc-close() for AF_ISO

soc_close() closes the socket *s*. Any data in the network and in BCAM is lost.

Return value

0:

If successful.

-1:

If errors occur. *errno* is set to indicate the error.

Errors indicated by *errno*

EBADF

The *s* parameter is not a valid descriptor.

See also

setsockopt()

soc_eof(), soc_error(), soc_clearerr() (eof, error, clearerr) - get status information

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


Kernighan-Ritchie-C:
int soc_eof(s);
int s;

int soc_error(s);
int s;

int soc_clearerr(s);
int s;

ANSI-C:
int soc_eof(int s);
int soc_error(int s);
int soc_clearerr(int s);
```

Description and return value

The *soc_eof()* function returns a value !=0 if the EOF condition applies to socket *s*; otherwise, *soc_eof()* returns the value 0.

The *soc_error()* function returns a value !=0 if a read or write error has occurred on socket *s*; otherwise, *soc_error()* returns the value 0. The error indicator remains set until it is deleted with the *soc_clearerr()* function.

The *soc_clearerr()* function deletes the error indicator for socket *s*.

soc_flush () (flush) - flush data from output buffer

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


Kernighan-Ritchie-C:
soc_flush(s);
int s;

ANSI-C:
int soc_flush(int s);
```

Description

The *soc_flush()* function is only supported in the AF_INET and AF_INET6 address families.

soc_flush() flushes all the data associated with socket *s* from the output buffer in the transport system.

Return value

0:

If the buffer was flushed or was empty.

EOF:

If the socket descriptor is invalid or the data transfer to the transport system failed.

soc_getc() (getc) - get character from input buffer

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


Kernighan-Ritchie-C:
int soc_getc(s);
int s;

ANSI-C:
int soc_getc(int s);
```

Description

The *soc_getc()* function is only supported in the AF_INET and AF_INET6 address families and can only be used on stream sockets.

The *soc_getc()* function reads the next character from the input buffer of socket *s* and returns the character as the result.

Return value

Integer value of the read character:

If successful.

EOF:

If no character could be read due to the end-of-file (EOF) condition.

Errors indicated by *errno*

EWOULDBLOCK

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

soc_gets() (gets) - get string from input buffer

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


Kernighan-Ritchie-C:
char *soc_gets(s, n, d);

char *s;
int n, d;

ANSI-C:
char* soc_gets(char* s, int n, int d);
```

Description

The *soc_gets()* function is only supported in the AF_INET and AF_INET6 address families and can only be used on stream sockets.

The *soc_gets()* function reads a character string of up to *n*-1 characters from the start of the input buffer of socket *d* into the buffer *s*. The maximum possible characters are read up to the first newline (represented by the sequence 0x15 in EBCDIC) or to the end of the input buffer of socket *d* or until *n*-1 characters are reached. The string returned in the buffer *s* is terminated with a null byte.

Return value

Pointer to the result string:

If successful.

Null pointer:

If read errors occur.

Errors indicated by *errno*

EWOULDBLOCK

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

soc_ioctl() (ioctl) - control sockets

```
#include <sys.socket.h>
#include <ioctl.h>
#include <net.if.h>

Kernighan-Ritchie-C:
int soc_ioctl(s, request, argp);

int s;
unsigned long request;
char *argp;

ANSI-C:
int soc_ioctl(int s, unsigned long request, char* argp);
```

Description

The *soc_ioctl()* function executes control functions for sockets.
s designates the socket descriptor.

The following control functions are supported for sockets in the AF_INET address family:

request	***arg**	**Function**
FIONBIO	int	Enable/disable blocking mode
FIONREAD	int	Get message length in buffer
SIOCGIFCONF	struct ifconf	Get interface configuration
SIOCGIFADDR	struct ifreq	Get Internet address of interface
SIOCGIFBRDADDR	struct ifreq	Get interface broadcast address
SIOCGIFFLAGS	struct ifreq	Get interface flags
SIOCGIFNETMASK	struct ifreq	Determine net mask for the interface
SIOCGLIFADDR	struct lifreq	Determine interface address
SIOCGLIFBRDADDR	struct lifreq	Determine broadcast address of the interface
SIOCGLIFCONF	struct lifconf	Output interface configuration list
SIOCGLIFFLAGS	struct lifreq	Determine interface flags
SIOCGLIFHWADDR	struct lifreq	Determine MAC address for the interface
SIOCGLIFINDEX	struct lifreq	Determine interface index

SIOCGLIFNETMASK	struct lifreq	Determine net mask for the interface
SIOCGLIFNUM	struct lifnum	Determine number of interfaces
SIOCGLAHCONF	struct lvhost	Output list of all active hosts
SIOCGLVHCONF	struct lvhost	Output list of active virtual hosts
SIOCGLVHNUM	int	Determine number of active virtual hosts

The following control functions are supported for sockets in the AF_INET6 address family:

request	***argp**	**Function**
FIONBIO	int	Enable/disable blocking mode
FIONREAD	int	Get message length in buffer
SIOCGLIFADDR	struct lifreq	Determine interface address
SIOCGLIFBRDADDR	struct lifreq	Determine broadcast address of the interface
SIOCGLIFCONF	struct lifconf	Output interface configuration list
SIOCGLIFFLAGS	struct lifreq	Determine interface flags
SIOCGLIFHWADDR	struct lifreq	Output MAC address for the interface
SIOCGLIFINDEX	struct lifreq	Determine interface index
SIOCGLIFNETMASK	struct lifreq	Determine MAC address for the interface
SIOCGLIFNUM	struct lifnum	Determine number of interfaces
SIOCGLAHCONF	struct lvhost	Output list of all active hosts
SIOCGLVHCONF	struct lvhost	Output list of active virtual hosts
SIOCGLVHNUM	int	Determine number of active virtual hosts

The following control functions are supported for sockets in the AF_ISO address family:

request	***argp**	**Function**
FIONBIO	int	Enable/disable blocking mode

The following control functions can be used without socket (s = 0):

request	***argp**	**Funktion**
SIOCGBCPROC	struct ifproc	Returns the BCAM processor name to an FQDN
SIOCGBCFQDN	struct ifproc	Returns the FQDN to a BCAM processor name

FIONBIO

This option affects the execution behavior of socket functions on socket *s* with data flow control enabled and for actions of the communications partner that have not yet been completed.

- **argp* = 0:
  Socket functions block until the function can be executed.

- **argp* != 0:
  Socket functions return with the *errno* code EWOULDBLOCK if the function cannot be executed immediately. The *select()* or *soc_poll()* function can be used to determine which sockets are ready for reading or writing.
  Default case: FIONBIO is not set.

FIONREAD

Returns the length of the message currently in the input buffer (in bytes).

SIOCGIFCONF

An output list is created in the form of non-concatenated elements of the type *struct ifreq* (see SIOCGIFADDR option). The caller provides the corresponding memory area for this list by entering the start address and the length in the relevant fields of the *ifconf* structure.
Only as many elements of the type *struct ifreq* are output as fit into the buffer made available.

These interfaces belong to a host, normally to the standard host. If virtual hosts are also configured, you can receive the corresponding interfaces in the following way:

- If the application is started under an ID relocated to a virtual host by an entry in the BCAM application table, the information about this virtual host is output.
- The *setsockopt()* subfunction SO_VHOSTANY can be used to select, before calling *soc_ioctl()*, the host for which information is to be output.
  The *ifconf* structure is declared in <net.if.h> as follows:

```
struct  ifconf {
        int             ifc_len;
        union {
                caddr_t ifcu_buf;
                struct  ifreq *ifcu_req;
        } ifc_ifcu;
#define ifc_buf ifc_ifcu.ifcu_buf
#define ifc_req ifc_ifcu.ifcu_req
};
```

SIOCGIFADDR

Returns the Internet address for the interface specified in the *ifreq* structure with the interface name *ifr_name*.

The *ifreq* structure is declared in <net.if.h> as follows:

```
struct ifreq {
#define IFNAMSIZ     16
   char ifr_name[IFNAMSIZ];      /* Interfacename z.B. IF000003" */
   union {
       struct sockaddr ifru_addr;
       struct sockaddr ifru_dstaddr;
       struct sockaddr ifru_broadaddr;
       short ifru_flags;
       int ifru_metric;
       caddr_t ifru_data;
    } ifr_ifru;
#define ifr_addr      ifr_ifru.ifru_addr      /* address */
#define ifr_dstaddr   ifr_ifru.ifru_dstaddr   /* dest. addr. of conn, */
#define ifr_broadaddr ifr_ifru.ifru_broadaddr /* broadcast address */
#define ifr_flags     ifr_ifru.ifru_flags     /* flags */
#define ifr_metric    ifr_ifru.ifru_metric    /* metric */

#define ifr_data      ifr_ifru.ifru_data      /* for use by interface */
};
```

Note that SOCKETS(BS2000) only returns information on *one* interface at present.

SIOCGIFBRDADDR

Returns the broadcast address specified for the interface in the *ifreq* structure (see SIOCGIFADDR option) when an IPv4 interface is concerned and when the IFF_BROADCAST flag is set. This is normally not the case because no broadcast can be generated with socket language means and transport system support.

SIOCGIFFLAGS

Returns the interface flags in the *ifr_flags* element for the interface name specified in the *struct ifreq*:

- IFF_UP - if the interface is active
- IFF_BROADCAST - if broadcast messages can be sent via this interface
- IFF_MULTICAST - if multicast messages can be sent via this interface
- IFF_LOOPBACK - if messages can be sent to loopback via this interface
- IFF_CONTROLLAN - if communication with the CONTROLLAN is possible via this interface

SIOCGIFNETMASK

Outputs the subnet mask in the form of bits of the network share of the subnet mask set to “1” (e.g. “FFFFFF00”) in the *ifr_addr* element for the IPv4 interface specified in the *ifreq* structure.

SIOCGLIFADDR

Returns the interface address for the name specified in *struct lifreq*. The *lifreq* structure is an *ifreq* structure enhanced particularly with respect to IPv6.

SIOCGLIFBRDADDR

Returns the broadcast address for the interface specified in the *lfreq* structure when the IFF_BROADCAST flag is set for this interface. This is normally not the case because no broadcast can be generated with socket language means and transport system support.
Separate subfunctions under *getsockopt()* / *setsockopt()* are provided for the use of MULTICAST (see section "getsockopt(), setsockopt() - get and set socket options").

SIOCGLIFCONF

An output list is created in the form of non-concatenated elements of the type *struct li-freq*. The caller provides the corresponding memory area for this list by entering the start address and the length in the relevant fields of the *lifconf* structure.
Only as many elements of the type *struct lifreq* are output as fit into the buffer made available. The output can be filtered by means of the assignments of the *lifc_family* and *lifc_flags* elements.
Values for *lifc_family*: AF_INET, AF_INET6, AF_UNSPEC
Values for *lifc_flags*: see SIOCGLIFFLAGS

These interfaces belong to a host, normally to the standard host. If virtual hosts are configured, you can receive the corresponding interfaces in the following way:

- If the application is started under an ID relocated to a virtual host by an entry in the BCAM application table, the information is output about this virtual host.
- The *setsockopt()* subfunction SO_VHOSTANY can be used to select, before calling *soc_ioctl()*, the host for which information is to be output.
  The *lifconf* structure is declared in <net.if.h> as follows:

```
struct  lifconf {
        sa_family_t     lifc_family;
        int             lifc_flags;
        int             lifc_len;
        union {
                caddr_t lifcu_buf;
                struct  lifreq *lifcu_req;
        } lifc_lifcu;
#define lifc_buf lifc_lifcu.lifcu_buf
#define lifc_req lifc_lifcu.lifcu_req
};
```

SIOCGLIFFFLAGS

Returns the interface flags in the *lifr_flags* element for the interface name specified in the *lifreq* structure:

- IFF_UP - if the interface is active
- IFF_BROADCAST - if broadcast messages can be sent via this interface
- IFF_MULTICAST - if multicast messages can be sent via this interface
- IFF_LOOPBACK - if messages can be sent to loopback via this interface
- IFF_CONTROLLAN - if communication with the CONTROLLAN is possible via this interface
- IFF_AUTOCONFIG - if this interface was supplied with an address generated by IPv6 autoconfig. This also includes the IPv6 link local address with the prefix FE80::/10 created locally in the host.

SIOCGLIFHWADDR

Outputs the MAC address for the interface name specified in the *lifreq* structure.

SIOCGLIFINDEX

Outputs the index for the interface name specified in *lifreq* structure.

SIOCGLIFNETMASK

Outputs the the subnet mask in the *lifr_addr* element and the prefix length in bits in the *ifr_addrlen* element for the interface name *lifr_name* specified in the *lifreq* structure. When an IPv4 interface is involved, the output takes place in the form of all bits concerned of the network share being set to “1” (e.g. “FFFFFF00”). When an IPv6 interface is involved, the network share is output as the original address and the following bits are then set to “0” (e.g. “FD11F052433485AA000000000000”).

SIOCGLIFNUM

Outputs the number of interfaces for the address family specified in the *lifnum* structure.

SIOCGLAHCONF

A non-concatenated list containing elements of the type *struct lvhost* is returned. These contain the socket host name, the BCAM host name, the host number and an active flag of the real host and, if present, also of virtual hosts. The user must transfer the memory space for this list in **argp* with the type *struct lvhost*. The length must be entered in the *lvhostlen* field.
If information on all hosts is to be output, a memory space of n x *sizeof(struct lvhost)* is required, where *n* is the maximum possible number of active hosts.
The return information can be accessed by direct addressing or using indexes. The number of returned list elements of the type *struct lvhost* is entered in the *vhostsum* field of the first element. In the last list element, the *vhostlast* field is flagged with “1”.
If the memory space provided is not large enough, the *vhostlast* field in the last list element is flagged with “1”.

SIOCGLVHCONF

A non-concatenated list containing elements of the type *struct lvhost* is returned, which in contrast to SIOCGLAHCONF contains only the information about the virtual hosts. The user must pass the memory space for this list with **argp* of the type *struct lvhost*. The length must be entered in the field *lvhostlen*. If information on all virtual hosts is to be output, a memory space of n x *sizeof(struct lvhost)* is required, where *n* is the return value from SIOCGLVHNUM.

The return information can be accessed by direct addressing or using indexes. The number of returned list elements of the type *struct lvhost* is entered in the field *vhostsum*. In the last list element, the *vhostlast* field is flagged with "1".

The structure *lvhost* is declared in <net.if.h> as follows:

```
struct  lvhost {
       int             lvhostlen;    /* length of memory for lvhostlist */
       unsigned short  vhostsum;     /* number of vhosts delivered      */
       unsigned short  vhostlast;    /* last element if not zero        */
       int             vhost_num;    /* vhost number, must be greater 1 */
       short           vhost_flag;   /* vhost active ?                  */
       char            vsockethost[33];  /* sockethostname of vhost     */
       char            vbcamhost[9]; /* bcamhostname of vhost           */
};
```

SIOCGLVHNUM

Returns the number of active virtual hosts.

The structure *lifreq* is declared as follows in <net.if.h>:

```
struct  lifreq {
#define LIFNAMSIZ 32
        char                            lifr_name[LIFNAMSIZ];
        union {
                int                     lifru_addrlen;
                unsigned int            lifru_ppa;
        } lifr_lifru1;
#define lifr_addrlen    lifr_lifru1.lifru_addrlen
#define lifr_ppa        lifr_lifru1.lifru_ppa
        unsigned int                    lifr_movetoindex;
        union {
                struct sockaddr_storage  lifru_addr;
                struct sockaddr_storage  lifru_dstaddr;
                struct sockaddr_storage  lifru_broadaddr;
                struct sockaddr_storage  lifru_token;
                struct sockaddr_storage  lifru_subnet;
                struct sockaddr          lifru_hwaddr;
                int                      lifru_index;
        union {
                unsigned int            lifru_flags_0,lifru_flags_1;
                u_int64_t               lifru_flags;
        } lifr_lifruflags;
                int                     lifru_metric;
                unsigned int            lifru_mtu;
                char                    lifru_data[1];
                char                    lifru_enaddr[6];
                int                     lif_muxid[2];
                struct lif_nd_req       lifru_nd_req;
                struct lif_ifinfo_req   lifru_ifinfo_req;
                char                    lifru_groupname[LIFNAMSIZ];
                unsigned int            lifru_delay;
        } lifr_lifru;

#define  lifr_addr       lifr_lifru.lifru_addr
#define  lifr_dstaddr    lifr_lifru.lifru_dstaddr
#define  lifr_broadaddr  lifr_lifru.lifru_broadaddr
#define  lifr_token      lifr_lifru.lifru_token
#define  lifr_subnet     lifr_lifru.lifru_subnet
#define  lifr_index      lifr_lifru.lifru_index
#define  lifr_flags      lifr_lifru.lifr_lifruflags.lifru_flags
#define  lifr_flags_l    lifr_lifru.lifr_lifruflags.lifru_flags_1
#define  lifr_flags_h    lifr_lifru.lifr_lifruflags.lifru_flags_0
#define  lifr_metric     lifr_lifru.lifru_metric
#define  lifr_mtu        lifr_lifru.lifru_mtu
#define  lifr_data       lifr_lifru.lifru_data
#define  lifr_enaddr     lifr_lifru.lifru_enaddr
#define  lifr_index      lifr_lifru.lifru_index
#define  lifr_ip_muxid   lifr_lifru.lif_muxid[0]
#define  lifr_nd         lifr_lifru.lifru_nd_req
#define  lifr_ifinfo     lifr_lifru.lifru_ifinfo_req
#define  lifr_groupname  lifr_lifru.lifru_groupname
#define  lifr_delay      lifr_lifru.lifru_delay
#define  lifr_hwaddr      lifr_lifru.lifru.lifru_hwaddr
};
```

The structure *sockaddr_storage* is declared as follows in <sys.socket.h>:

```
#define _SS_MAXSIZE     128         /* Implementation specific max size */
#define _PADSIZE        (_SS_MAXSIZE - (sizeof(u_int64_t) + 8 ))
struct sockaddr_storage {
        sa_family_t     ss_family;         /* address family */
#define __ss_family     ss_family
        char            res[6];            /* reserved for alignment */
        u_int64_t       addr;              /* address */
        char            pad[_PADSIZE];     /* pad up to max size */
};
```

The structure *lifconf* is declared as follows in <net.if.h>:

```
struct  lifconf {
        sa_family_t     lifc_family;
        int             lifc_flags;
        int             lifc_len;
        union {
                caddr_t lifcu_buf;
                struct  lifreq *lifcu_req;
        } lifc_lifcu;
#define lifc_buf lifc_lifcu.lifcu_buf
#define lifc_req lifc_lifcu.lifcu_req
};
```

The structure *lifnum* is declared as follows in <net.if.h>:

```
struct lifnum {
        sa_family_t     lifn_family;
        int             lifn_flags;
        int             lifn_count;
};
```

SIOCGBCPROC

The caller has to pass the *ifproc* structure's fields *fqdn and fqdnlen to the program. Pointer *fqdn has to be supplied as a char array which contains the FQDN (Fully Qualified Domain Name) to be checked. The length of this FQDN has to be inserted in the fqdnlen field. After the function call the BCAM processor name belonging to the FQDN is written to the field bcproc[] with the string null termination (‘\0‘).

If there is no BCAM processor name to this FQDN, a negative return value is returned. This also applies, if there is a processor name assigned to this FQDN in the FQDN file, but the corresponding processor/route has not been created.
An IP address must be converted into an FQDN beforehand (see getnameinfo() function).
The socket descriptor is neither needed nor evaluated for this function.

The structure *ifproc* is declared as follows in <net.if.h>:

```
struct ifproc {
         char            *fqdn;     /* FQDN (Fully Qualified Domain Name) */
         int             fqdnlen;   /* length of the FQDN                 */
         char            bcproc[9]; /* space for BCAM processor name      */
};
```

SIOCGBCFQDN

The caller must provide a char array for *fqdn and enter the length of this array in the fqdnlen field. In addition, the BCAM processor name must be recorded into the bcproc[] field. After calling this function the FQDN (Fully Qualified Domain Name) associated with the BCAM processor name is written to the field to which *fqdn refers with the string null termination ('\0'). If fqdnlen is smaller than the length of the FQDN, an error will be set and no FQDN will be returned. An error will also be set, if the specified BCAM processor name does not exist or if there is no FQDN for this processor name. In this case length 0 will be written to fqdnlen.

The socket descriptor is neither needed nor evaluated for this function.

Return value

-1:

If errors occur. *errno* is set to indicate the error.

Errors indicated by *errno*

EBADF

The *s* parameter is not a valid descriptor.

EINVAL

request or *arg* are not valid for this device (interface, socket).

soc_poll() - multiplex input/output

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


Kernighan-Ritchie-C:
int soc_poll(fds, nfds, timeout);

struct pollfd fds[];
unsigned long nfds;
int timeout;

ANSI-C:
int soc_poll(struct pollfd fds[], unsigned long nfds, int timeout);
```

Description

The *soc_poll()* function tests a set of socket descriptors, which are transmitted with an array of structure elements of type *pollfd*. Depending on the desired test, each descriptor-specific structure element states whether messages can be received or sent on this socket descriptor or whether specific events have occured.

The *soc_poll()* function is supported in the AF_INET, AF_INET6 and AF_ISO address families.

The *fds* parameter is a pointer to the array to be sent by the caller with an element of type *struct pollfd* for each socket descriptor to be tested.

The *nfds* parameter specifies the set of descriptors to be tested.

The *timeout* parameter specifies the maximum waiting time in seconds that the *soc_poll()* function is available for testing the descriptors, if no event occurs:

- If *timeout* = 0: No waiting time, only all marked file descriptors are tested.
- If *timeout* = -1: *soc_poll()* is blocked, until an event occurs in at least one of the selected file descriptors.

The maximum time value that will be accepted is ~ 24.85 days. Values bigger than that will lead to waiting until an event occurs.

pollfd structure

The *pollfd* structure is declared in <sys.poll.h> as follows:

```
struct pollfd {
   int      fd;              /* socket file descriptor to poll*/
   short    events;          /* events on interest on fd*/
   short    revents;         /* events that occured on fd */
};
```

The *fd* socket descriptor designates the socket to be tested.

events designates the events to be tested on this socket.

revents returns the test result. The POLLNVAL, POLLERR, POLLHUP bits are always set in *revents*, if the conditions for this are met, regardless of the bits set in *events*.

Events can be requested in the *events* element field using the following bit masks:

- POLLIN
- POLLOUT

The following bit masks are not supported in the *events* element field and are not set in the *revents* element field:

- POLLPRI
- POLLRDNORM
- POLLWRNORM
- POLLRDBAND
- POLLWRBAND

The following events can be displayed in the bit mask of the *revents* element field:

POLLIN

With an existing connection data can be read non-blocking.

POLLOUT

With an existing connection data can be written non-blocking.

POLLNVAL

The socket selected with the socket descriptor is not available or it does not have the status that displays an active connection. This flag is only written as a result in the *revents* field.

POLLERR

An error has been reported to the socket selected and the connection is inactive. This flag is only written as a result in the *revents* field.

POLLHUP

The application or the transport system have closed the connection.
This flag is only written as a result in the *revents* field.

If a negative value is specified for an *fd* socket descriptor, this value will be ignored and the *revents* field will be set to 0.

Return value

0:

The time specified in the *timeout* parameter has elapsed without an event display being set.

>0

The positive value indicates the number of socket descriptors when at least one event display has been set in the *revents* field.

-1

If errors occur. *errno* is set to indicate the error.

Errors

EACCES

The socket function is not supported by the called subsystem.

EINTR

The *soc_poll()* call was interrupted by *soc_wake()*.

EINVAL

The value of *nfds* is greater than the maximum number permitted of socket descriptors. The maximum value is determined by calling *getdtablesize()*.

See also

select()

soc_putc() (putc) - put character in output buffer

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


Kernighan-Ritchie-C:
int soc_putc(c, s);

char c;
int s;

ANSI-C:
int soc_putc(char c, int s);
```

Description

The *soc_putc()* function is only supported in the AF_INET and AF_INET6 address families and can only be used on stream sockets.

The *soc_putc()* function writes the character *c* to the output buffer of socket *s*.

The characters are written to the output buffer of the socket up to the maximum capacity of 32760 bytes before being automatically transmitted to the BCAM transport system. If desired, the output buffer intended for BCAM can be cleared unconditionally by using the socket function *soc_flush()*.

Return value

!=EOF:

If successful.

EOF:

If nothing could be written to the buffer due to the end-of-file (EOF) condition.

See also

soc_flush()

soc_puts() (puts) - put string in output buffer

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


Kernighan-Ritchie-C:
int soc_puts(s, d);

char *s;
int d;

ANSI-C:
int soc_puts(char* s, int d);
```

Description

The *soc_puts()* function is only supported in the AF_INET and AF_INET6 address families and can only be used on stream sockets.

The *soc_puts()* function writes the character string *s* to the output buffer of socket *d*.

Return value

0:

If successful.

EOF:

If nothing could be written to the buffer due to the end-of-file (EOF) condition, or if errors occur.

See also

soc_flush()

soc_read(), soc_readv() (read, readv) - receive a message from a socket

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


Kernighan-Ritchie-C:
int soc_read(s, buf, nbytes);

int s;
char *buf;
int nbytes;


int soc_readv(s, iov, iovcnt);

int s;
struct iovec *iov;
int iovcnt;

ANSI-C:
int soc_read(int s, char* buf, int nbytes);
int soc_readv(int s, struct iovec* iov, int iovcnt)
```

Description

The *soc_read()* and *soc_readv()* functions read messages

- from a stream socket *s* in the AF_INET or AF_INET6 address family or
- from a socket *s* in the AF_ISO address family.

soc_read() and *soc_readv()* can only be used with a socket for which a connection has already been set up.

For *soc_read()*, the *buf* parameter points to the first byte of the receive buffer *buf*.
nbytes specifies the length (in bytes) of the receive buffer, and thus the maximum message length.

For *soc_readv()*, the received data is placed in a vector with the elements *iov*[0], *iov*[1], ..., *iov*[*iovcnt*-1]. The vector elements are objects of type *struct iovec*.
iovcnt indicates the number of vector elements.

The *iovec* structure is declared in <sys.uio.h> as follows:

```
struct iovec
{
   caddr_t iov_base;  /* buffer for auxiliary data */
   int iovlen;        /* buffer length */
};
```

The address of the vector is passed in the parameter iov. Each vector element specifies the address and length of a storage area in which soc_readv() places the data received from socket s. The soc_readv() function fills these areas with data sequentially and only moves to the next area when the current area has been totally filled.

Return value

>=0:

If successful (number of received bytes).

-1:

If errors occur. *errno* is set to indicate the error.

Errors indicated by *errno*

EBADF

The *s* parameter is not a valid descriptor.

EIO

There is no available user data to be read.

ENETDOWN

The connection to the network is down.

ENOTCONN

No connection exists for the socket.

EOPNOTSUPP

The socket type is not supported. The socket is not of type SOCK_STREAM.

EWOULDBLOCK

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

EPIPE

The connection has been shut down.

See also

connect(), getsockopt(), recv(), select(), send(), soc_ioctl(), soc_read(), soc_write(), soc_writev(), socket()

soc_wake() - awaken a task waiting with select() or soc-poll()

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

Kernighan-Ritchie-C:
int soc_wake(pid);
int *pid;

ANSI-C:
int soc_wake(int* pid);
```

Description

The *soc_wake()* function can be used to awaken the task identified by **pid* if the task is currently waiting with *select()* or *soc_poll()*. The *soc_wake()* call causes *select()* or *soc_poll()* to exit with a return value of “-1” and the error EINTR.

soc_wake() can awaken another task with the same user ID and, if called from a signal routine, can also awaken its own task.

**pid* is a variable which must be supplied with the task sequence number (TSN) of the waiting task.
The task **pid* must exist at the time of the *soc_wake()* call and have an opened socket. If the task **pid* is not waiting with *select()*, the signal set by *soc_wake()* is assigned to the next *select()* call.

Return value

0:

If successful

-1:

If errors occur. *errno* is set to indicate the error.

Errors indicated by *errno*

ESRCH

There is no task with the number **pid*. This may be either because **pid* is an invalid task number or because the SOCKETS(BS2000) program has not yet been loaded.

EACCES

No access authorization. The task **pid* has another user ID.

soc_write(), soc_writev() (write, writev) - send a message from socket to socket

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

Kernighan-Ritchie-C:
int soc_write(s, buf, nbytes);

int s;
char *buf;
int nbytes;


#include <sys.types.h>
#include <sys.uio.h>

int soc_writev(s, iov, iovcnt)

int s;
struct iovec *iov;
int iovcnt;

ANSI-C:
int soc_write(int s, char* buf, int nbytes);
int soc_writev(int s, struct iovec* iov, int iovcnt);
```

Description

The *soc_write()* and *soc_writev()* functions support the following means of message transfer:

- Messages from a stream socket *s* to another stream socket (AF_INET, AF_INET6)
- Message from a connected datagram socket to another socket (AF_INET, AF_INET6)
- From a socket *s* belonging to the AF_ISO address family to another socket belonging to the same address family.

soc_write() and *soc_writev()* can only be used if a connection between the two sockets has already been established.

For *soc_write()*, the *buf* parameter points to the first byte of the send buffer, and *nbytes* specifies the length (in bytes) of the send buffer.

For *soc_writev()*, the data to be sent is supplied in the vector with the elements *iov*[0], *iov*[1], ... ,*iov*[*iovcnt*-1]. The vector elements are objects of type *struct iovec*.
iovcnt indicates the number of vector elements.

The *iovec* structure is declared in <sys.uio.h> as follows:

```
struct iovec{
    caddr_t  iov_base;  /* buffer for auxiliary data */
    int      iovlen;    /* buffer length */
};
```

The address of the vector is passed in the *iov* parameter. Each vector element specifies the address and length of a storage area from which *soc_writev()* reads the data to be sent to the receiving socket *s*.

Return value

>=0:

If successful (number of bytes actually sent).

-1:

If errors occur. *errno* is set to indicate the error.

Errors indicated by *errno*

EBADF

The *s* parameter is not a valid descriptor.

ECONNRESET

The connection to the partner was interrupted (only with sockets of type SOCK_STREAM).

EINVAL

A parameter has specified an illegal value.

EIO

I/O error. The message could not be passed to the transport system.

ENETDOWN

The connection to the network is down.

ENOTCONN

No connection exists for the socket.

EOPNOTSUPP

The socket type is not supported. The socket is not of type SOCK_STREAM.

EPIPE

The socket is not activated for writing, or the socket is connection-oriented and the partner has shut the connection down.

EWOULDBLOCK

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

Note

If the connection is established with a non-blocking socket then *errno* EINPROGRESS may occur on the next function call. This message indicates that the connection is not yet in a state which permits a data transfer phase.

See also

connect(), getsockopt(), recv(), select(), soc_read(), soc_readv(), socket()

socket() - create socket

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


Kernighan-Ritchie-C:
int socket(domain, type, protocol);

int domain, type, protocol;

ANSI-C:
int socket(int domain, int type, int protocol);
```

Description

The *socket()* function creates a communications endpoint and returns a descriptor.

The *domain* parameter defines the communications domain in which communications are to take place. This also defines the protocol family to be used and thus the family of the addresses used for later operations on the socket. These families are defined in the <sys/socket.h> header file. The AF_INET, AF_INET6 and AF_ISO address families are supported.

The *type* parameter defines the type of the socket and the semantics of the communications. The following socket types are currently defined:

- SOCK_STREAM
- SOCK_DGRAM
- SOCK_RAW

Each of these types is supported in the AF_INET and AF_INET6 address families. Only the type SOCK_STREAM is defined in the AF_ISO address family.

The *protocol* parameter is not evaluated.

Socket operations are controlled by socket level options and defined in the <sys.socket.h> header file. The user can get and set these options with the *getsockopt()* and *setsockopt()* functions, respectively.

AF_INET and AF_INET 6 address families

A socket of type SOCK_STREAM provides a secured and bidirectional connection on which data is transmitted sequentially. A stream socket must be connected to another stream socket before any data can be sent or received on it.

A connection to another socket is set up when one socket requests a connection to a partner socket with *connect()*, and the partner system confirms the connection with *accept()*. After the connection has been successfully established, both partners can transmit data with *soc_read()* or *soc_readv()* and *soc_write()* or *soc_writev()* or similar calls such as *send()* and *recv()*.

A socket of type SOCK_DGRAM supports the transmission of datagrams. A datagram is a connectionless, unsecured message with a fixed maximum length. The *sendto()* function sends datagrams from one datagram socket to another datagram socket specified in the *sendto()* call. Conversely, datagrams are received with the *recvfrom()* function. *recvfrom()* returns the next datagram together with the address of the sender.

If the communications partner for a datagram socket has been preset with the function *connect()*, the functions *send()* and *recv()* can also be used for this datagram socket.

A socket of the type SOCK_RAW supports the transmission of ICMP/ICMPv6 messages.

AF_ISO address family

A socket of type SOCK_STREAM provides a secured and bidirectional connection on which a record-oriented and sequential transfer of data takes place. A stream socket must be connected to another stream socket before any data can be sent or received on it.

A connection to another socket is set up when one socket requests a connection to a partner socket with *connect()*, and the partner system confirms the connection with *accept()* and one of the functions *send(), soc_write()* or *sendmsg()*.

After the connection has been successfully established, both partners can transmit data with *soc_read()* or *soc_readv()* and *soc_write()* or *soc_writev()* or similar calls such as *send()* and *recv()* or *sendmsg()* and *recvmsg()*.

Return value

>=0:

Designates a non-negative descriptor if successful.

-1:

If errors occur. *errno* is set to indicate the error.

Errors indicated by *errno*

EMFILE

The table of descriptors per task is full; the maximum number of socket descriptors that can be processed concurrently has been reached. This maximum value can be determined with the *getdtablesize()* function.

ENOBUFS

There is not enough storage space in the buffer. The socket cannot be created until sufficient storage resources have been freed.

EAFNOSUPPORT

The address family is not supported by this protocol family. The specified address is incompatible with the protocol used.

ENOMEM

Memory bottleneck. Not enough virtual storage space could be assigned when executing the function.

EPROTONOSUPPORT

The socket type is not supported in this domain.

See also

accept(), bind(), connect(), getsockname(), getsockopt(), listen(), recv(), recvfrom(), select(), send(), sendto(), soc_close(), soc_ioctl(), soc_read(), soc_readv(), soc_write(), soc_writev()

SOCKETS(BS2000) interface for an external bourse

This chapter describes the additional functions of the socket interface for BS2000 in the special mode for using an external bourse. This makes it possible to coordinate various events by specifying an external bourse with a common wait point.

The socket mode (external wait point or not) is set with the first socket call and is then static. The setting is made with a *setsockopt()* subfunction.

The additional function *soc_getevent()* and the subfunction SO_ASYNC for *setsockopt()* are provided to achieve this functionality.

Description of the additional functions

setsockopt() - modify socket options

```
#include <sys.types.h>
#include <sys.socket.h>
#include <netinet.in.h> /* only with AF_INET or AF_INET6 */


Kernighan-Ritchie-C:


int setsockopt(s, level, optname, optval, optlen);

int s;
int level;
int optname;
char *optval;
int optlen;

ANSI-C:
int setsockopt(int s, int level, int optname, char* optval, int optlen);
```

Description

The *level, optname*, *optval* and *optlen* parameters of the *setsockopt()* function allow users to modify the properties (options) of the socket interface or of an individual socket *s*.

level SOL_GLOBAL

In this case, the current parameter value for *s* is of no significance. For this reason, the value 0 should be specified for *s*.

optname	**optlen*	Value range of *optval*
SO_ASYNC	4	Pointer to short ID of the event ID

The subfunction *SO_ASYNC* is only permitted for *setsockopt()* and is only effective when the user calls the subsystem for the first time.

If this is the case, then the subsystem for this user is switched to an operating mode which permits coordination with other events via a common wait point. *optval* must then be used to specify the short ID of the bourse's event ID to which the sockets should send the communication events. These events can then be retrieved with the *soc_getevent* function.

Comment

In this operating mode, the sockets generated with *socket()* are automatically generated in non-blocking mode.

soc_getevent() - get socket event

#include <sys.socket.h>

ANSI-C:
int soc_getevent(struct aevent * exb_event);

Description

The function *soc_getevent()* provides the caller with information about a delivered event signaled to the bourse used by the caller.
Standard data and event-specific data are stored in the output structure *aevent*.

The signal to the bourse transfers a 2-word post code with the following structure:

Word 1:

Byte 1: Event code X‘3E‘

Byte 2: User Call Indicator

X‘14‘ = IPv4 event

X‘15‘ = IPv6 event

Byte 3: Event Indicator

C‘E’ = Normal event (see list of possible events in section "Possible events" in chapter "soc_getevent() - get socket event")

C‘W’ = Event was triggered by a Wake incident
User Call Indicator = X‘00‘.

C‘S‘ = Event was triggered by BCEND.
User Call Indicator = X‘00‘ and word 2 = X‘00000000‘.

C'N' = No event has occurred. This event mainly gives feedback to SOCKETS.
User Call Indicator = X'00'

Byte 4: X‘00‘

Word 2:

Undefined

The *aevent* is declared as follows in <sys.socket.h>

```
struct aevent {
   int   fd;              /* socket filedescriptor */
   int   event;            /* transport system event*/
   int   subevent;         /* additional information about the event*/
   int   datalen;          /* data length transferred to caller*/
   int   fd_errno;         /* errno */
   int   fd_array_cnt;     /* number of FD's, if ECLS event*/
   int   fd_array[FD_MAX]; /* FD's for ECLS event*/
   }
```

Return value

0:

Success

1:

Error

Possible events

EXB_ECLS

TSAP Termination Indication, forced termination of the TSAP by the transport system.Data: *fd*, *event, fd_array_cnt, fd_array*

EXB_ERQQ

Connection Request Indication, connection request.
Data: *fd*, *event*

EXB_ERSP

Connection Response Indication, partner’s acknowledgment of connection request.Data: *fd*, *event*

EXB_EDIS

Disconnect Indication, disconnection request.
Data: *fd*, *event*

EXB_EDTA

Data Indication, TCP data reception
Data: *fd*, *event, datalen*

EXB_EDTU

Unitdata Indication, UDP data reception.
Data: *fd*, *event, datalen*

EXB_EERR

Error Report Indication
ICMP error message

EXB_EGOD

Data Go Indication, data transfer can be continued.
Data: *fd*, *event*

EXB_NOEV

No event present.
The return value is set to 1.

EXB_TRYL

No event can currently be retrieved.
However, another attempt can be made later.
The return value is set to 1.

EXB_SHUT

The BCAM transport system has been terminated or is currently in the termination phase.
The application must also be terminated.
The return value is set to 1.

The *errno* is set if an error occurs.

Software package SOCKETS(BS2000) V21.1

SOCKETS(BS2000) subsystems

SOC-TP	Subsystem for system program
SOC6	Subsystem for user programs on all hardware platforms
SOC6-X8	Subsystem for special programs on SE series with SU x86

SOCKETS(BS2000) programs

- ping4
- ping6
- traceroute

ping4

Autonomous diagnostic program for determining the availability of a host in an IPv4 network. An ICMP echo request is sent and a test is made to see whether the host answers with an ICMP echo reply.

Online help: `ping4 -h`
Description: See the “BCAM Volume 1/2” manual

ping6

Autonomous diagnostic program for determining the availability of a host in an IPv6 network. An ICMPv6 echo request is sent and and a test is made to see whether the host answers with an ICMPv6 echo reply.

Online help: `ping6 -h`
Description: See the “BCAM Volume 1/2” manual

traceroute

Autonomous diagnostic program for determining the (possible) routes and internet nodes a data package passes to get to a specified ip address.

Online help: `tracert --help`

Description: See the "BCAM Volume 1/2" manual

SOCKETS(BS2000) DNS access

Access to DNS takes place using the software package openNet Server V21.0, which contains SOCKETS(BS2000) V21.1, using the ported program GETDNS (see the “BCAM Volume 1/2” manual).

SOCKETS(BS2000) V21.1 contains its own copy of the GETDNS and is therefore independent from the GETDNS program needed for the name resolution for BCAM.

If DNS does not provide any corresponding information for the DNS information function which is called, an attempt is made to obtain this information from the BCAM transport system.

It is possible to reverse the order of the DNS resolution so that the information from the BCAM transport system is determined first. This can be configured with the option SO_RESOLVE_BCAM (see "getsockopt(), setsockopt() - get and set socket options").

This applies both for implementing host names and for the FQDN (Fully Qualified Domain Name).

Certain settings are to be configured via the configuration file SYSDAT.SOCKETS.*nnn*.RESOLV.CONF. *nnn* is the SOCKETS version, the current one being 211. A list of all available options can also be found in the manual “BCAM Volume 1/2”.

SOCKETS(BS2000) - query to FQDN file

OpenNet Server supports an FQDN file which is managed by BCAM. With an entry in this file an FQDN can be converted to a BCAM name. This information is taken into account when SOCKETS(BS2000) issues a request to BCAM because no DNS server was available or because DNS was unable to supply any information in response to a request (see the “BCAM Volume 1/2” manual).

Producing the SOCKETS(BS2000) user program

The SOCKETS(BS2000) V21.1 subsystem *SOC6* is compatible with the predecessor version.

The include files of the user program library are compatible for the Kernighan-Ritchie, ANSI-C and C++ mode, i.e. corresponding defines have been implemented in the include files.

Software requirements

The following software is required to use SOCKETS(BS2000) V21.1:

- openNet Server V21.0
- BS2000 C-Compiler >= V4.0

Programming

- For the compiler run, only the header file library SYSLIB.SOCKETS.211 is required in addition to the libraries with the private header files and the header files of the C runtime system of SOCKETS(BS2000).
- When linking, no resolve library of SOCKETS-BS2000 is required.

> The subsystem entries of the SOCKETS(BS2000) functions used are reported by the linkage editor as unresolved externs. However, at the time the program executes, they are resolved by the Sockets subsystem.

Related publications

You will find the manuals on the internet at http://bs2manuals.ts.fujitsu.com.

C/C++
C/C++ Compiler
User Guide

C/C++
C Library Functions
User Guide

openNet Server
BCAM
User Guide

interNet Services
Administrator Guide

interNet Services
User Guide

SNMP-AGENTS
User Guide

RFCs

You can find complete information about the Request for Comments (RFCs) on the home page of the Internet Engineering Task Force (IETF):

www.ietf.org