1 Preface

SOCKETS/XTI(POSIX) is the name for the socket and XTI functions within the POSIX interface for BS2000/OSD. These functions provide the development environment for BS2000/OSD users who want to write socket or XTI application programs under POSIX.

1.1 Brief description of the product

POSIX offers the socket and XTI functions according to the X/Open Group specification for UNIX95 branding. The socket and XTI programming provides a number of options for developing communication applications.

- The socket interface (SOCKETS) is an interface for network programming within the POSIX subsystem. It can be used to develop communication applications based on the TCP/IP protocols.
- The X/Open Transport Interface (XTI) is the standard defined by X/Open for a number of programming interfaces which make it possible for an application to access the network levels.
- RFC 2553 for socket applications.

1.2 Target group

This manual is aimed at programmers who wish to use the SOCKETS or XTI interface functions to develop communication applications based on the POSIX interface. Familiarity with C programming and the POSIX functions is required and assumed.

1.3 Summary of contents

In this manual, the various options available for socket and XTI programming are described and illustrated using simple examples. The example programs show how the SOCKETS or XTI functions are used for connection mode communication applications using the TCP protocol 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(POSIX) communication applications. Example programs are used to illustrate basic topics such as address structures, connection setup, data transfer and client/server communications.
- Chapter 6 contains an alphabetic reference section with the user functions of the SOCKETS(POSIX) interface.
- Chapters 7 to 9 provide an introduction to developing XTI(POSIX) communication applications. Example programs are used to illustrate basic topics such as connection setup, data transfer and client/server communications.
- Chapter 10 deals with the XTI trace.
- Chapter 11 contains an alphabetic reference section with the library functions of the XTI(POSIX) interface.
- Chapter 12 uses two example procedures to illustrate how you can compile and link the program you created.
- Chapter 13 describes the Internet *inetd* daemon and the configuration files for Internet communications. The dependencies of SOCKET(POSIX) and XTI(POSIX) applications on the BS2000/OSD BCAM transport system are also shown.
- Chapter 14 describes the compatibility restrictions of the SOCKETS(POSIX) and XTI(POSIX) interfaces over the following interfaces:
  - socket/XTI interface under UNIX systems
  - socket interface in BS2000/OSD

1.4 Changes compared with the previous edition of the manual

This section provides you with an overview of the changes made to the March 2005 edition of the “SOCKETS/XTI for POSIX” manual compared with the February 2001 edition.

- Enhancement of the *ioctl()* function by new control functions:

  SIOCGLIFNUM
  SIOCGLIFCONF
  SIOCGLIFADDR
  SIOCGLIFINDEX
  SIOCGLIFBRDADDR
  SIOCGLIFNETMASK
  SIOCGLIFFLAGS
  SIOCGIFNUM
  SIOCGIFINDEX
  SIOCGIFNETMASK

- Furthermore, the functions of SOCKETS(POSIX) now support the ability of the C compiler to generate programs containing ASCII literals.

1.5 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*)
<angled brackets>	Identify header files in descriptive text.
[ ]	Optional entries. The square brackets are metacharacters which may not be input 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 meaningful for understanding the example. The ellipses are metacharacters which may not be input within statements.

The notational conventions for describing the user functions are explained at the beginning of each chapter concerned.

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.

1.6 README file

Please refer to the product-specific README file for any functional changes and revisions to the current product version not contained in this manual. You will find the README file on your BS2000/OSD computer under the name SYSRME.*product.version.language.* Please consult your system service staff for the user ID under which the README file is stored. You can view the README file with the /SHOW-FILE command or an editor and print it on a standard printer with the following command:

/PRINT-DOCUMENT *filename*, LINE-SPACING=*BY-EBCDIC-CONTROL

2 SOCKETS(POSIX) basics

This chapter explains the basic terms and functions of socket programming. Program examples for the topics handled in this chapter are summarized in chapter “Client/server model with SOCKETS(POSIX)” on page 57. The separate functions of the SOCKETS interface are described in detail in chapter “SOCKETS(POSIX) user functions” on page 69.

2.1 POSIX network connection via the SOCKETS interface

The SOCKETS interface is one of the interfaces for network programming within the POSIX subsystem. It can be used to develop communication applications based on the TCP/IP protocols. NEA and OSI protocols are not supported.

The SOCKETS interface is defined in a separate library. If this library is linked into a POSIX application, the SOCKETS interfaces set up the connection to the network over the POSIX subsystem and BCAM transport system.

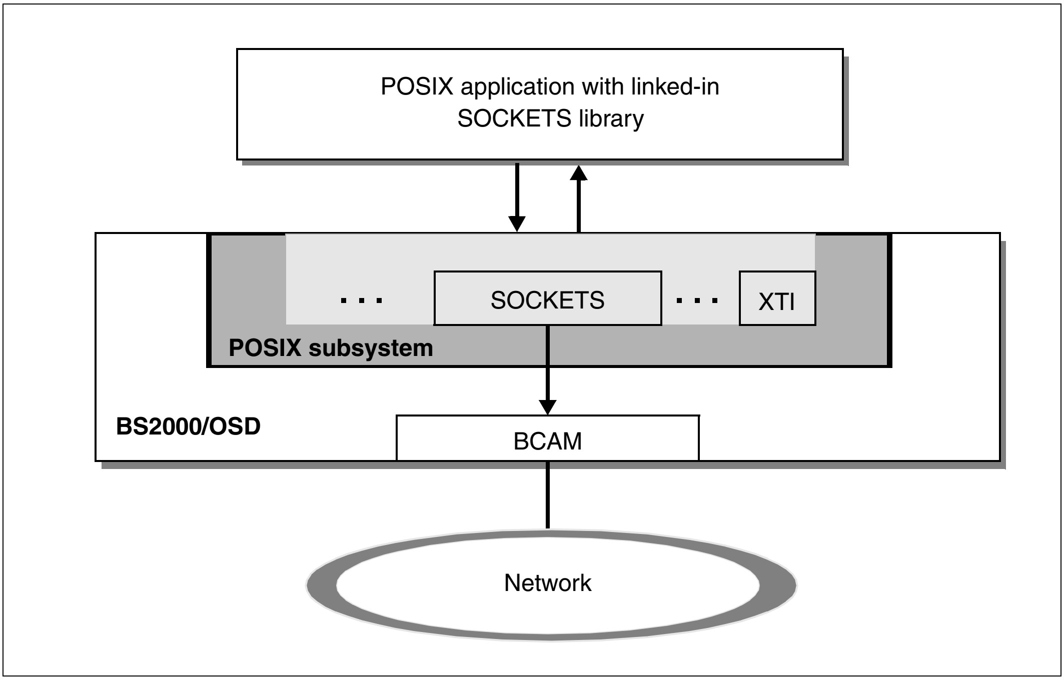

Figure 1: SOCKETS in BS2000/OSD and POSIX

The POSIX network connection libraries represent the link between the POSIX objects, such as e.g. file descriptors, and the BS2000 mechanisms. There are a few restrictions when using the functions because of the differences between the POSIX concepts and BS2000. These restrictions are described in detail in chapter “Compatibility restrictions” on page 305.

The functions for opening a network connection return a socket file descriptor. This can be used in all relevant POSIX functions which work with file descriptors.

2.2 Header files

When SOCKETS(POSIX) is installed, X/Open-compliant header files are copied into the */usr/include* directory. In chapter “SOCKETS(POSIX) user functions” on page 69 and chapter “XTI(POSIX) library functions” on page 223, the description of the socket or XTI function also specifies which header file(s) the application must link in to execute the function concerned.
SOCKETS/XTI(POSIX) provides the following header files:

arpa/inet.h

- defines utility functions and macros for manipulating Internet addresses
- defines the data types in *in_port_t* and *in_addr_t* as defined in <netinet/in.h>
- defines the *in_addr* structure as defined in <netinet/in.h>

sys/sockio.h

- defines the socket control functions called by *soc_ioctl()*

net/if.h

- structures for the packet 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/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/byteorder.h

- macros for converting the byte sequence

sys/un.h

- address structure for UNIX system domain (AF_UNIX)

sys/xti_inet.h

- Internet-specific structures and options of the transport provider xti.h

- declares the XTI functions
- structures and constants of the transport provider
- symbolic constants for XTI error codes
- states and options of the transport endpoint

2.3 Socket types

A socket is a basic component for developing communications applications by forming a communications endpoint. It can be assigned a name via which the socket can then be accessed and addressed.

Each socket belongs to a specific type and has at least one affiliated process. Several related processes can use the same socket and a process can also have connections to several sockets.

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 together the common characteristics of processes which communicate over sockets.
The socket interface in BS2000/OSD supports the Internet communications domains AF_INET and AF_INET6, and in the local host communications domain AF_UNIX.

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

– stream sockets
– datagram sockets

2.3.1 Stream sockets (connection-oriented)

Stream sockets support connection-oriented communications in the Internet communications domains AF_INET and AF_INET6, and in the local host communications domain AF_UNIX. A Stream socket provides bidirectional, secured and sequential data flow, thus ensuring that the data is only transferred once and in the correct order. The data record limits are lost when connection-oriented communications are used with stream sockets.

Stream sockets are used to develop connection-oriented communications applications based on the TCP protocol.

2.3.2 Datagram sockets (connectionless)

Datagram sockets support connectionless communications in the Internet communications domains AF_INET and AF_INET6, and in the local host communications domain AF_UNIX. 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 is transferred more than once. A process which receives messages on a datagram socket may therefore possibly receive the messages more than once and/or in a different order from that transmitted. The application therefore has the responsibility of 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.

2.4 Socket addressing

A socket is created initially without a name or address. You then have to use the *bind()* function to assign the socket a name (address) according to its address family (see section “Assigning a name to a socket” on page 18) so that processes can address it. You can then receive messages over the socket.

2.4.1 Using socket addresses

When the *bind()*, *connect()*, *getpeername()*, *getsockname()*, *recvfrom()*, *recvmsg()*, *sendto()* and *sendmsg()* 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 to the address family used, according to the address structure. This address structure is different for each address family used (see section “sockaddr_in address structure of the AF_INET address family” on page 14, section “sockaddr_in6 address structure of the AF_INET6 address family” on page 14 and section “sockaddr_un address structure of the AF_UNIX address family” on page 15).

Before passing the parameter, the pointer which passes the address must be converted with the cast operator from type “pointer to the structure of the used address family” to type “pointer to *struct sockaddr*“. The *sockaddr* structure is the general address structure used in the socket functions and is independent of domains.

The address structures for the AF_INET, AF_INET6 and AF_UNIX address families are described in the following sections. The structures for the host, protocol and service names are described in chapter “Address conversion with SOCKETS(POSIX)” on page 41.

2.4.2 Addressing with an Internet addresses

SOCKETS(POSIX) 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 manual “openNet Server V3.0 (BS2000/OSD)“. You will find a detailed explanation of IPv6 functionality in the "IPv6 Introduction and Conversion Guide, Stage 1".

2.4.2.1 sockaddr_in address structure of the AF_INET address family

With the AF_INET address family, a name comprises an Internet address and a port number. You use the *sockaddr_in* address structure for the AF_INET address family. The *sockaddr_in* structure is declared as follows in the <netinet/in.h> header file:

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

struct in_addr {
   in_addr_t s_addr;
   };
```

You can supply a variable *server* of type *struct sockaddr_in* with a name, 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 (..., (struct sockaddr *)&server, ...)  /* bind() call with
                                                type conversion */
```

2.4.2.2 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.

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

```
struct sockaddr_in6 {
sa_family_t sin6_family;    /* AF_INET6 address family */
in_port_t sin6_port;        /* 16-bit port number  */
uint32_t sin6_flowinfo
struct in6_addr sin6_addr;  /* IPv6 address */
uint32_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 */
```

2.4.2.3 sockaddr_un address structure of the AF_UNIX address family

With the AF_UNIX address family, a name (address) comprises a path name. You use the *sockaddr_un* address structure for the AF_UNIX address family.

The *sockaddr_un* structure is declared as follows in the <sys/un.h> header file:

```
struct sockaddr_un {
   sa_family_t  sun_family;       /* address family */
   char         sun_path[108];    /* path name */
   };
```

You can supply a variable *server* of type *struct sockaddr_un* with a name, e.g. using the following statements:

```
struct sockaddr_un server;
...
server.sun_family = AF_UNIX;
strcpy(server.sun_path, “/tmp/unix_socket“);
```

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

2.5 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

– local host communications domain AF_UNIX

You must therefore specify AF_INET, AF_INET6 or AF_UNIX 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.

If you set *protocol* to 0, you specify the standard protocol:

– TCP for socket type SOCK_STREAM

– UDP for socket type SOCK_DGRAM

2.5.1 Creating a socket in the AF_INET domain

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

```
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 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.

2.5.2 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.

2.6 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, according to its address family. Processes can only address the socket and receive messages over it after this is done. You bind a name to the socket. i.e. you assign the socket a local address, with the *bind()* function.

You call *bind()* as follows:

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

The structure of the name *name*, which is assigned to socket *s*, differs according to the address family (AF_INET, AF_INET6 or AF_UNIX).

- 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 page 14).
- 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 page 14).

namelen contains the length of the data structure that describes the name.

2.6.1 bind() call with AF_INET

With AF_INET, *name* comprises an IPv4 address and a port number. *name* is passed in a variable of type *struct sockaddr_in* (see page 14).

The following program extract illustrates how a name is assigned to a socket.

```
#include <sys/types.h>
#include <netinet/in.h>
 ...
struct sockaddr_in sin;
 ...
/* The statements which supply sin with an Internet
   address and a port number must be inserted here.*/
 ...
bind(s, (struct sockaddr *)&sin, sizeof sin);
```

2.6.2 bind() call with AF_INET6

With AF_INET6, *name* comprises an IPv6 address and a port number. *name* is passed in a variable of type *struct sockaddr_in6* (see page 14).

The following program extract illustrates how a name is assigned to a socket.

```
#include <sys/types.h>
#include <netinet/in.h>
 ...
struct sockaddr_in6 sin6;
 ...
/* The statements which supply sin6 with an Internet
   address and a port number must be inserted here.*/
 ...
bind(s, (struct sockaddr *)&sin6, sizeof sin6);
```

2.6.3 Dependencies on port numbers

You must note the following when selecting the port number:

– Port numbers lower than IPPORT_RESERVED (1024) are reserved for privileged users.

– Port numbers in the range from 1024 to PRIVPORT# must differ from port numbers with fixed assignments for privileged applications (see section “Dependencies of the BS2000/OSD BCAM transport system” on page 302).

– Certain port numbers are reserved for some standard applications. These worldwide applicable assignments are stored in the */etc/inet/services* file. This file can be extended for local networks to record assigned port numbers.

2.6.4 bind() call with AF_UNIX

With AF_UNIX, *name* only comprises a path name which is passed in a variable of type *struct sockaddr_un* (see page 15).

The following program extract illustrates how a name is assigned to a socket.

```
#include <sys/types.h>
#include <sys/un.h>
 ...
struct sockaddr_un sun;
 ...
/* The statements which supply sun with the path name
   must be inserted here.*/
 ...
bind(s, (struct sockaddr *)&sun, sizeof sun);
```

The path name, which must be specified in the *sun.sun_path* component, is created as a file in the file system using *bind()*. The process that calls *bind()* must therefore have write rights to the directory in which the file is to be written. The system does not delete the file. It should therefore be deleted by the process when it is no longer required.

2.6.5 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 (see page 18). 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 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:

- Receiving messages:
  - The socket *s* bound to INADDR_ANY can receive messages over 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 process 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 over 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:0:0:0:A00:6FF:FE08:9A6B, a task to which socket *s* is assigned can accept connection requests which are addressed to 128.32.0.4 and 3FFE:0:0:0:A00:6FF:FE08:9A6B.
- Sending messages:
  - The socket *s* bound to INADRR_ANY can send messages over any IPv4 network interface on its host.
  - The socket *s* bound to IN6ADDR_ANY can send messages over any network interfaces on its host.

  This allows the socket *s* bound to INADDR_ANY to address any other socket that can be reached via an IPv4 network interface of the host on which socket *s* lies.
  The socket *s* bound to IN6ADDR_ANY, on the other hand, can address any other socket that can be reached via any network interface of the host on which socket *s* lies.

The following examples show how a process can bind a local name to a socket without an Internet address being specified. The process 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);
memset(&sin6, 0 , sizeof sin6);
sin6.sin6_family = AF_INET6;
memcpy(sin6.sin6_addr.s6_addr,sin6addr_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 process 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);
memset(&sin6, 0 , sizeof sin6);
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);
```

2.6.6 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 free port number from the range of non-privileged port numbers.

2.7 Connection-oriented communications

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

```
<local address, local port, remote address, remote 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 processes is generally asymmetric, with one process assuming the role of the client and the other the role of the server.

2.7.1 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, (struct sockaddr *)&server, sizeof server);
```

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

```
struct sockaddr_in6 server;
 ...

connect(s, (struct sockaddr *)&server, sizeof server);
```

The *server* parameter passes the IPv4 or IPv6 address and the port number of the server with which the client wishes to communicate.
If the socket of the client process has no address 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 next section). However, all names assigned automatically by the system are retained even if the connection setup fails.

2.7.2 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 process to take up contact with the server.

   listen() also causes the POSIX subsystem 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;
 ...
listen(s, 5);
fromlen = sizeof (from);
newsock = accept(s, (struct sockaddr *)&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, (struct sockaddr_in6 *)&from, &fromlen);
```

The first parameter passed when *listen()* is called is the descriptor *s* of the socket over 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 process. The transport system currently supports a maximum of 50 pending connection requests.

The first parameter passed when *accept()* is called is the descriptor *s* of the socket over 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 over the new socket. The server can accept additional connections over 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 process also has no way of indicating that it only wants to accept connection requests from one or more specific partners. The server process must therefore note where the connection comes from and terminate it if it does not want to communicate with the client process concerned.

The following is described in detail in chapter “Extended SOCKETS(POSIX) functions” on page 49:

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

2.7.3 Data transfer with connection-oriented communications

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

There are several functions for sending and receiving data. You can elect to use either the functions *read()* and *write()* or *readv()* and *writev()* :

```
write(s, buf, sizeof buf);
read(s, buf, sizeof buf);
writev(s, iovec, iovcnt);
readv(s, iovec, iovcnt);
```

These functions are part of the basic scope of the POSIX interface. They are described in the manual „C Library Functions (BS2000/OSD) for POSIX Applications“. Socket-specific features of these functions are described in section “Using standard POSIX functions for sockets” on page 128.

You can alternatively use the following socket-specific functions:

```
send(s, buf, sizeof buf, flags);
sendmsg(s, msg, flags);
recv(s, buf, sizeof buf, flags);
recvmsg(s, msg, flags);
```

The socket-specific functions are described in detail in section “Functions” on page 76.

2.7.4 Examples of connection-oriented client/server communications

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

The example programs are only valid for the communications domain AF_INET. If they are modified according to the information in the sections “Socket addressing” on page 13 and “Creating a socket” on page 16, they are also valid for the AF_INET6 domain.

Example 1: Initialization of a streams connection by the client

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

/*
 * This program creates a socket and initializes a connection with the
 * socket passed in the command line.
 * A message is sent over the connection.
 * The socket is then closed and the connection shut down.
 * The program is called as follows:
 * program_name host_name  port_number
 */

main(argc, argv)
      int argc;
      char *argv[];
{
      int sock;
      struct sockaddr_in server;
      struct hostent *hp;
      /* Create 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");
        close(sock);
        exit(0);
}
```

Example 2: Acceptance of the streams connection by the server

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.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);
      }

      /* Find and output the appropriate port number. */
      length = sizeof server;
      if (getsockname(sock, (struct sockaddr *)&server,
        &length) < 0) {
              perror("getting socket name");
              exit(1);
      }
      printf("Socket port #%d\n", ntohs(server.sin_port));

      /* 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");
                      if (rval == 0)
                              printf("Ending connection\n");
                      else
                              printf("-->%s\n", buf); }
              while (rval > 0);
              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 process is
       * terminated or reaches its normal conclusion.
       */

       exit(0);
}
```

2.8 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 carried out via datagram sockets (SOCK_DGRAM). A datagram socket provides a symmetric interface for data exchange via datagrams. In contrast to connection-oriented communication, where the client and server communicate with each other over a fixed connection, no connection is set up for datagram transfers. Each message contains the destination address instead.

In section “Creating a socket” on page 16 there is 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 page 18). Otherwise, the system assigns the local Internet address and/or port number the first time data is sent (see page 23).

2.8.1 Data transfer with connectionless communications

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

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

You use the *s*, *buf*, *buflen* and *flags* parameters in exactly the same way as with connection-oriented sockets. You pass the destination address with *to* and the length of the address with *tolen*. The sender is not informed of any errors when a datagram socket is used. If the system has the information locally that a message cannot be transferred (e.g. if a network cannot be reached), the *sendto()* call returns “-1” and the global *errno* variable contains the appropriate error code.

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

```
recvfrom(s, buf, buflen, flags, (struct sockaddr *)&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 process 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 process 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.

2.8.2 Examples of connectionless communications

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

The example programs are only valid for the communications domain AF_INET. If they are modified according to the information in the sections “Socket addressing” on page 13 and “Creating a socket” on page 16, they are also valid for the AF_INET6 domain.

Example 1: receiving datagrams

```
#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, (struct sockaddr *)&name,
        sizeof name ) < 0) {
              perror("binding datagram socket");
              exit(1);
      }

      /* Find and output the corresponding port number. */
      length = sizeof(name);
      if (getsockname(sock, (struct sockaddr *)&name,
        &length) < 0) {
              perror("getting socket name");
              exit(1);
      }
      printf("Socket port #%d\n", ntohs(name.sin_port));
       /* Read from the socket. */
      peerlen=sizeof peer;
      if (recvfrom(sock, buf, 1024, (struct sockaddr *)&peer, &peerlen) < 0)
              perror("receiving datagram packet");
      printf("-->%s\n", buf);
      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. The command format is as follows:
 * program_name host_name port_number
 */

main(argc, argv)
      int argc;
      char *argv[];
{
      int sock;
      struct sockaddr_in name;
      struct hostent *hp;
      /* Create socket over 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 over 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,
        (struct sockaddr *)&name, sizeof name) < 0)
              perror("sending datagram message");
      close(sock);
      exit(0);
}
```

2.9 Closing a socket

If you no longer need a socket, you can close its descriptor with the *close()* function:

```
close(s);
```

This function is also part of the basic scope of the POSIX interface (see page 129 and the manual "C Library Functions (BS2000/OSD) for POSIX Applications").

2.10 Multiplexing input/output

It is often meaningful to distribute inputs and outputs over several sockets. You use the *select()* or the *poll()* function for this type of input/output multiplexing. A program can monitor several connections simultaneously using these functions.

The following program extract illustrates using *select()*.

```
#include <sys/time.h>
#include <sys/types.h>
#include <sys/select.h>
 ...
fd_set readmask, writemask, exceptmask;
struct timeval timeout;
 ...
select(nfds, &readmask, &writemask, &exceptmask, &timeout);
```

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

– *select()* uses the bit mask passed with *readmask* to test which sockets data can be read from.

– *select()* uses the bit mask passed with *writemask* to test which sockets data can be written to.

– *select()* uses the bit mask passed with *exceptmask* to test which sockets have an exception pending.

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.

The bit masks which represent the descriptor sets are stored as bit fields in integer strings. The size of the bit fields is defined via the FD_SETSIZE constant. FD_SETSIZE is defined in <sys/select.h> with a default value that is at least as large as the maximum number of descriptors supported by the system.

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 on page 143 in 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
- which descriptors have exceptions pending

If *select()* terminates with a timeout, it returns the value 0. However, the bit masks have already been updated.
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 executing *select()*, you can use the FD_ISSET(*fd*, &*mask*) macro call to test 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 testing 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*: i.e. a connection request is pending on socket *fd*.

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

Any process can use the following program code to read data from two sockets. The timeout value is set to five seconds.
The example program is only valid for the communications domain AF_INET. If it is modified according to the information in the sections "Socket addressing" on page 13 and "Creating a socket" on page 16, it is also valid for the AF_INET6 domain.

```
#include <sys/select.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define TRUE 1
#define TESTPORT 2222

/*
 * This program uses select to test whether someone is trying to set up
 * a connection and then calls accept.
 */

main()
{
        int sock, length;
        struct sockaddr_in server;
        int msgsock;
        char buf[1024];
        int rval;
        fd_set ready;
        struct timeval to;
        /* 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);
        }
```

```
/* Find and output corresponding port number */
      length = sizeof server;
      if (getsockname(sock, (struct sockaddr *)&server,
        &length) < 0) {
              perror("getting socket name");
              exit(1);
      }

      printf("Socket port #%d\n", ntohs(server.sin_port));

    /* Start acceptance of connections. */
      listen(sock, 5);
      do {
              FD_ZERO(&ready);
              FD_SET(sock, &ready);
              to.tv_sec = 5;
              to.tv_usec=0;
              if (select(sock + 1, &ready, (fd_set *)0,
                (fd_set *)0, &to) < 0) {
                      perror("select");
                      continue;
              }
              if (FD_ISSET(sock, &ready)) {
                      msgsock = accept(sock, (struct sockaddr *)0,
                        (int *)0);
                      if (msgsock == -1)
                              perror("accept");
                      else do {
                              memset(buf, 0, sizeof buf);
                              if ((rval = read(msgsock, buf, 1024)) < 0)
                                      perror("reading stream message");
                              else if (rval == 0)
                                      printf("Ending connection\n");
                              else
                                      printf("-->%s\n", buf);
                      } while (rval > 0);
                      close(msgsock);
              } else
                      printf("Do something else\n");
      } while (TRUE);
      exit(0);
}
```

2.11 Interaction of the SOCKETS interface functions

The two following figures illustrate the interaction between the functions of the SOCKETS(POSIX) interface. The separate functions are described in detail in section “Functions” on page 76.

Figure 2 illustrates the interaction of the SOCKETS(POSIX) interface functions with stream sockets (SOCK_STREAM) in the Internet domains AF_INET and AF_INET6.

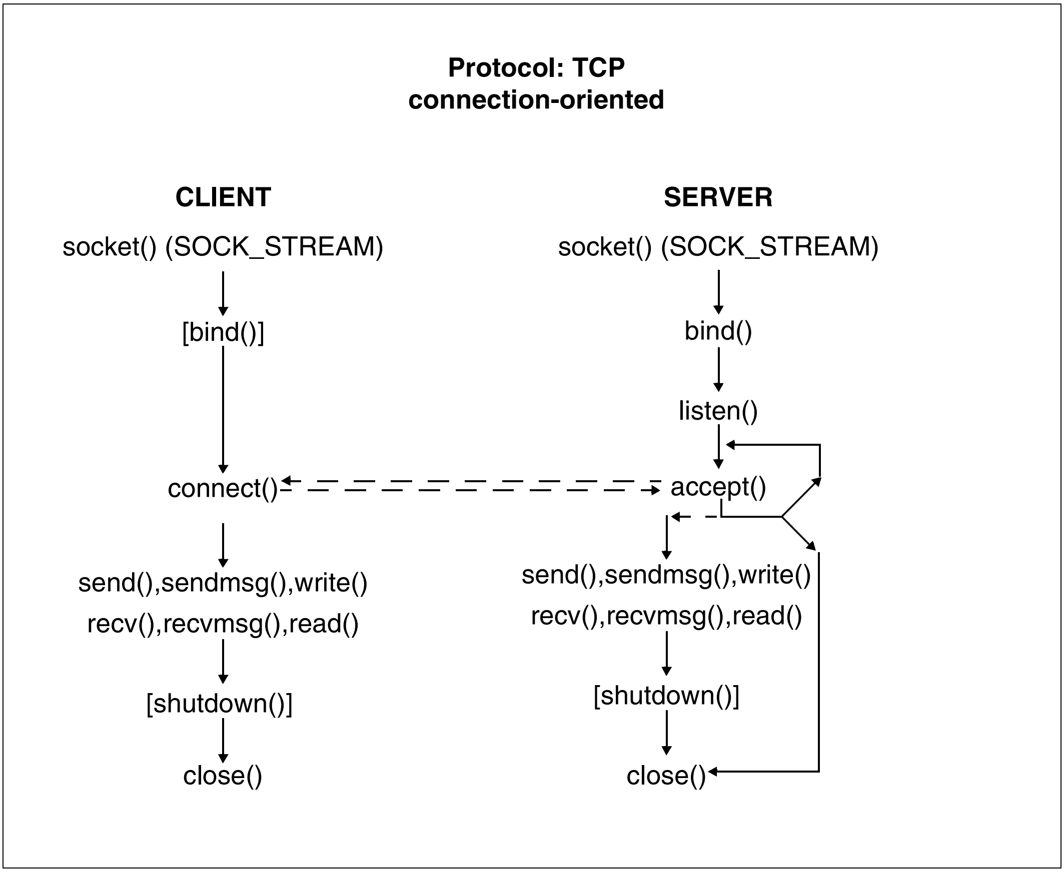

Figure 2: Interaction of the SOCKETS(POSIX) interface functions with stream sockets.

Figure 3 illustrates the interaction of the SOCKETS(POSIX) interface functions with datagram sockets (SOCK_DGRAM) in the Internet domains AF_INET and AF_INET6.

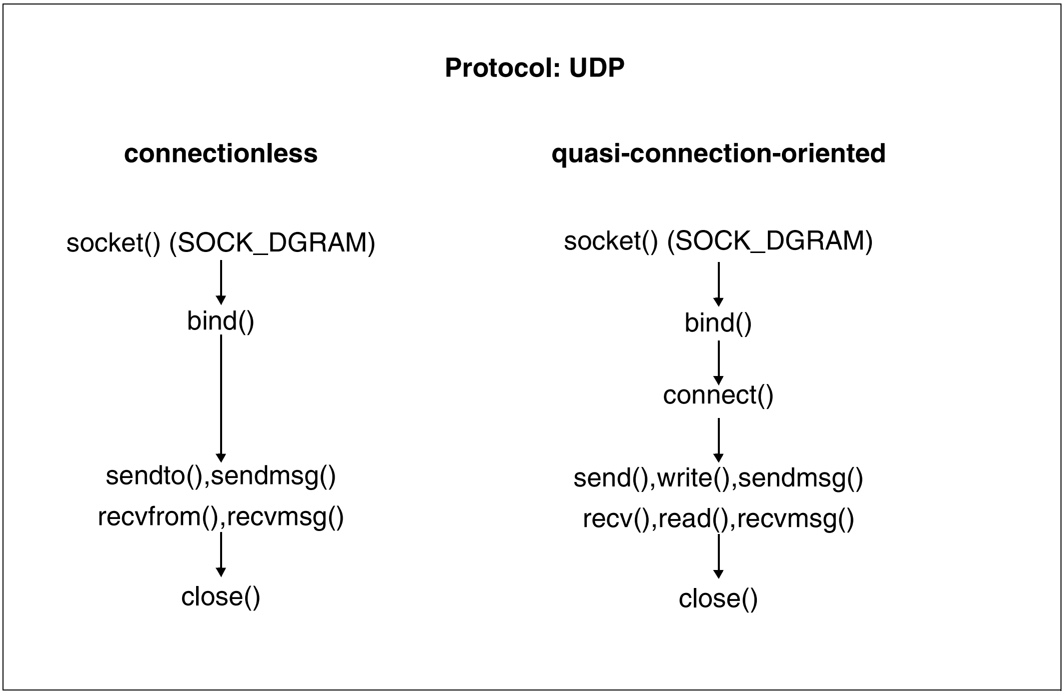

Figure 3: Interaction of the SOCKETS(POSIX) interface functions with datagram sockets.

3 Address conversion with SOCKETS(POSIX)

Network addresses have to be determined and created to enable processes to communicate with each other over sockets. The SOCKETS library provides various utility functions and macros for this purpose and these are described in this chapter.
All utility functions are described in detail in chapter “SOCKETS(POSIX) user functions” on page 69.

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 into a service number (port number) and a host name into a network address (IPv4 or IPv6 address).
3. Using the port number and IPv4 or IPv6 address, the system determines the route to the host on which the service is provided.

It is not meaningful to use the host name to get the location, i.e. physical address of a host. Lower level network services should locate a host at the time that another host wishes to communicate with it. This method makes it possible to change the physical location of a host without affecting addressing by the communicating partner.

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 wish to use one of these functions, you have 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(POSIX)” on page 57.

3.1 Converting host names into 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()* 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()* always contains the following information:

– 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.

3.2 Converting protocol names into protocol numbers

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

getprotobyname() returns a pointer to an object of data type *struct protoent* as its result. The *protoent* structure is declared as follows:

```
struct protoent {
     char *p_name;          /* Official protocol name */
     char **p_aliases;      /* Alias list */
     int  p_proto;          /* Protocol number */
};
```

3.3 Converting service names into 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 networks. 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 into 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 into 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 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 */
};
```

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");
```

3.4 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 degree. However, some network dependency still remains as the IPv4 or IPv6 address has to be specified in a user program if a name is assigned to a service or socket.

In addition to the library functions for converting names to addresses, there are also macros which simplify handling 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 table 1 convert bytes and integers from host byte order to network byte order and vice versa.

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

Table 1: Library macros for converting byte orders

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.

For IPv6 addresses, there is no definition of a difference between host byte order and network byte order, and there is therefore no corresponding conversion function.

The host and network byte orders are identical in BS2000/OSD. The macros listed in table 1 are therefore defined as null macros (macros without contents). However, it is strongly recommended that you use the macros for creating portable programs.

3.5 Example of address conversion

The client program code of the *remote login* shown below demonstrates address conversion.

```
#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, (struct sockaddr *)&server, sizeof server) < 0) {
              perror("rlogin: connect");
              exit(5);
      }

      exit(0);
}
```

The example program is only valid for the communications domain AF_INET.
It is also valid for the AF_INET6 domain if you make the following changes:

– *server* is of the type *struct sockaddr_in6*
– *struct sockaddr_in6* is supplied with a value from the *getipnodebyname* socket function (not from *gethostbyname* as for AF_INET)

You will find a more detailed description in the sections “Socket addressing” on page 13 and “Creating a socket” on page 16.

4 Extended SOCKETS(POSIX) 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(POSIX) features:

- non-blocking sockets
- broadcast messages
- socket options
- multicast messages
- interrupt-controlled socket input/output

4.1 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. This means that, in contrast to normal sockets, non-blocking sockets prevent a process from being interrupted because it has to wait for termination of *accept()*, *connect()* or input/output functions. You can mark a socket created with *s=socket()* as non-blocking with the *fcntl()* function as follows:

```
#include <fcntl.h>
...
int s;
 ...
s = socket(AF_INET, SOCK_STREAM, 0);
 ...
if (fcntl(s, F_SETFL, O_NONBLOCK) < 0) {
      perror("fcntl(s, F_SETFL, O_NONBLOCK) <0");
      exit(1);
}
 ...
```

The *fcntl()* function is part of the basic scope of the POSIX interface. *fcntl()* is described on page 130 and in the manual "C Library Functions (BS2000/OSD) for POSIX Applications".

You should particularly watch out for the EWOULDBLOCK error when executing the *accept()*, *connect()* or input/output 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.

4.2 Broadcast messages

When using a datagram socket, it is possible to send broadcast packets to many of the networks that are connected to the system. The network itself must support broadcasts as the system does not support software broadcast simulation. Broadcast messages can load the network heavily as they force all hosts in the network to service them.

Broadcasting is only provided in the AF_INET address family since there is no broadcast mechanism in IPv6.

Broadcasting generally used for one of the two following reasons:

– A resource whose address is initially unknown, is to be found in a local network.

– Important functions, such as the routing function, want to send information to all reachable hosts.

To send a broadcast message, a datagram socket must first be created:

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

Then the socket is marked as being allowed to send broadcasts:

```
int on = 1;
setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof on);
```

Finally, the socket is assigned a port number:

```
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, (struct sockaddr *)&sin, sizeof sin);
```

The destination address to which the broadcast message is to be sent depends on the network(s) to which the message is to be sent. The Internet domain supports a short name for broadcasts in the local network, the address INADDR_BROADCAST (defined in <netinet/in.h>).

Familiarity with the topology of the networks to which the host is connected is required to make up a list of all hosts which can be reached by a broadcast.

As this information should be stored host-independent and may possibly not be available, BS2000/OSD supports a method of obtaining the information from the system data structures.

For ipv4, The *ioctl()* SIOCGIFCONF call returns the network configuration of a host as a simple structure *ifconf*. This structure contains a data area comprising a list of *ifreq* structures. The list contains an *ifreq* structure for each address domain supported by each network interface via which the host is connected.

The *ifreq* structure is declared in <net/if.h> as follows:

```
struct ifreq {
#define IFNAMSIZ    16
    char ifr_name[IFNAMSIZ];             /* if name, e.g., "en0" */
    union {
        struct sockaddr ifru_addr;
        struct sockaddr ifru_dstaddr;
        char ifru_oname[IFNAMSIZ];       /* other if name */
        struct sockaddr ifru_broadaddr;
        short ifru_flags;
        int ifru_metric;
        char ifru_data[1];               /* interface-dependent data */
        char ifru_enaddr[6];
    } ifr_ifru;
#define ifr_addr      ifr_ifru.ifru_addr      /* address */
#define ifr_dstaddr   ifr_ifru.ifru_dstaddr   /* other end of p-to-p link */
#define ifr_oname     ifr_ifru.ifru_oname     /* other if name */
#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 */
#define ifr_enaddr    ifr_ifru.ifru_enaddr    /* ethernet address */
};
```

The following program code returns the ipv4 interface configuration:

```
struct ifconf ifc;
int    ifn;
char   *buf;

if (ioctl(s, SIOCGIFNUM, (char *)&ifn) < 0) {
  ...
}
ifc.ifc_len = ifn * sizeof (struct ifreq);
if ((buf = malloc(ifc.ifc_len)) == NULL) {
  ...
}
ifc.ifc_buf = buf;
if (ioctl(s, SIOCGIFCONF, (char *)&ifc) < 0) {
  ...
}
```

After calling *ioctl()*, *buf* contains a list of *ifreq* structures: one *ifreq* structure for each network to which the host is connected. These *ifreq* structures can initially be sorted according to interface names and then to supported address families. The *ifc.ifc_len* component specifies the size of the memory area (in bytes) required by the *ifreq* structures.

Each *ifreq* structure contains a record of interface flags. These interface flags indicate whether the network belonging to the interface concerned is active or not, or whether it is a broadcast network, etc.

The SIOCGIFFLAGS control function specified in the *ioctl()* call searches through these flags for an interface that is specified via an *ifreq* structure as follows:

```
struct ifreq *ifr;

ifr = ifc.ifc_req;

for (n=ifc.ifc_len/sizeof (struct ifreq);
       --n >= 0; ifr++) {
       /*
        * It must be ensured that no interface is used that belongs to
        * an address domain other than the desired one.
        */
       if (ifr->ifr_addr.sa_family != AF_INET)
               continue;
       if (ioctl(s, SIOCGIFFLAGS, (char *) ifr) < 0) {
           ...
       }
       /*
        * Trivial cases are skipped.
        */
       if ((ifr->ifr_flags & IFF_UP) == 0 ||
           (ifr->ifr_flags & IFF_LOOPBACK) ||
           (ifr->ifr_flags &
           (IFF_BROADCAST) == 0)
               continue;
}
```

As soon as the flags have been found, the Broadcast address has to be determined. In broadcast networks, you use SIOCGIFBRDADDR in an *ioctl()* call for this.

```
struct sockaddr dst;
if (ifr->ifr_flags & IFF_BROADCAST) {
       if (ioctl(s, SIOCGIFBRDADDR, (char *) ifr) < 0) {
               ...
       }
       memcpy((char *) &dst, (char *) &ifr->ifr_broadaddr,
               sizeof ifr->ifr_broadaddr);
}
```

After obtaining the broadcast or destination address (now in *dst*) with the *ioctl()* call, you can call the *sendto()* function as follows:

```
sendto(s, buf, buflen, 0, (struct sockaddr *)&dst, sizeof dst);
```

In the above loop, *sendto()* is run for each interface connected to the host that supports broadcast or point-to-point addressing. Similar program code can be used if a process only wishes to send broadcast messages to a specific network. However, the correct destination address must be found in the loop.

Because the sending datagram socket must have a name assigned to it, received broadcast messages always contain the address and port number of the sender.

i The BCAM BCOPTION command can be used to control whether a host may receive

broadcast messages (see the "openNet Server V3.0 (BS2000/OSD)" manual). However, this setting can neither be influenced nor determined with the *setsockopt()* and *getsockopt()* socket functions. It must therefore be ensured when using broadcasting that this option is set on the hosts concerned.

4.3 Socket options

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

For example, you can set options for the following purposes:

– to identify a socket for sending broadcast messages

– to direct a socket to stop connection shutdown until all data is transferred

The general form of the calls are as follows:

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

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

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, indicated by the symbolic constant SOL_SOCKET. SOL_SOCKET is defined in <sys/socket.h>.

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*.

optlen defines the length of the option value, as with *setsockopt()*.
optlenp is a pointer which defines the size of the memory area to which *optval* points when *getsockopt()* is called. After returning from *getsockopt()*, *optlenp* specifies the current length of the option value returned in **optval*.

4.4 Multicast messages (AF_INET)

Multicast messages can be sent and received if you use datagram sockets.

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
- IP_DROP_MEMBERSHIP: log off from a multicast group
- IP_MULTICAST_TTL: display or define the multicast hop limit

4.5 Interrupt-controlled socket input/output

The SIGIO signal informs a process as soon as a socket (or generally a file descriptor) has data that can be read.

You have to include the following in the basic program code to enable a process to react to the SIGIO signal:

1. Define a signal handling function using the *sigaction()* function (see the manual "C Library Functions (BS2000/OSD) for POSIX Applications").
2. Set either the process number or the process group number to allow your process or group process number to be informed of pending inputs. Set the process or process group number with the *fcntl()* function. The default process group of a socket is group 0.
3. You must enable asynchronous reporting of pending input/output requests with an additional *fcntl()* call.

The following program code illustrates how a process is prepared for receiving SIGIO signals. The process is informed asynchronously when data can be read or written by calling a user-defined *signal()* function for handling SIGIO.

```
#include <fcntl.h>
#include <sys/file.h>
 ...
int io_handler();
 ...
signal(SIGIO, io_handler);

/* Sets the process for receiving the SIGIO signals. */

if (fcntl(s, F_SETOWN, getpid()) < 0) {
perror("fcntl F_SETOWN");
exit(1);
}

/* Allows reception of asynchronous I/O signals */

if (fcntl(s, F_SETFL, FASYNC) < 0) {
perror("fcntl F_SETFL, FASYNC");
exit(1);
}
```

5 Client/server model with SOCKETS(POSIX)

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 process. This implies the asymmetry when setting up connections between a client and server as described in chapter “SOCKETS(POSIX) basics” on page 7. 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 client or server. 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, when a service is accessed there is a client and a server.

The following are described in the sections below:

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

5.1 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 “awakes” and serves the client by executing the relevant actions for the client request. The server is accessed via the known Internet address.

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

- *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
- *close()*: close socket

Example: connection-oriented server

```
#include <stdio.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;
    int msgsock;
    char buf[1024];
    int rval;

    /* 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 *)0, (int *)0);
    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);

  close(msgsock);
  close(sock);
}
```

The server uses the *socket()* function to create a communications endpoint (socket) and the corresponding descriptor. 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.

The server uses the *listen()* function to determine whether connection requests are pending and it can then accept them with *accept()*. The value returned by *accept()* is tested to ensure that the connection was successfully set up. As soon as the connection is set up, data is read from the socket with the *recv()* function. The server closes the socket with the *close()* function.

The example program is only valid for the communications domain AF_INET.
If it is modified according to the information in the sections “Socket addressing” on page 13 and “Creating a socket” on page 16, it is also valid for the AF_INET6 domain.

5.2 Connection-oriented client

The client side was shown in the example on page 58. 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 while 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. As in the server process, the service definition for a *remote login* must first be determined.

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

- *socket()*: create socket
- *setsockopt()*: set options for the socket
- *gethostbyname()*: get the host name entry
- *connect()*: request a connection on the socket
- *send()*: write data to the socket
- *close()*: close socket

Example: connection-oriented client

```
#include <stdio.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 server;
    struct hostent *hp, *gethostbyname();
    char buf[1024];
    struct linger ling;

    ling.l_onoff = 1;
    ling.l_linger = 60;
```

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

    /* Fill in the address structure */
    server.sin_family = AF_INET;
    server.sin_port = htons(TESTPORT);
    hp = gethostbyname(argv[1]);
    if (hp == 0)
       { 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, (struct sockaddr *)&server,
                                     sizeof(server) ) < 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);
       }

    close(sock);
}
```

The client creates a communications endpoint (socket) and the corresponding descriptor with the *socket()* function.
The client gets the address of the host (the host name is passed as a parameter) with *gethostbyname()*. A connection must then be set up to the server for the desired host. The client initializes the address structure for this and the connection is set up with *connect()*.
After connection setup, data is written to the socket with the *send()* function.
The created socket is closed with the *close()* function.

The example program is only valid for the communications domain AF_INET.
If it is modified according to the information in the sections “Socket addressing” on page 13 and “Creating a socket” on page 16, it is also valid for the AF_INET6 domain.
Please also make sure that you use the *getipnodebyname()* function (instead of *gethostbyname()* as for AF_INET).

5.3 Connectionless server

Most services work connection-oriented, but some are based on using datagram sockets and work connectionless.

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

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

The program is shown in two variants:

- In the first variant (example1), the program is terminated when a message arrives (*read()*).
- In the second variant (example 2), the program waits in an endless loop for further messages after a message has been read.

Example 1: connectionless server without a program loop

```
#include <stdio.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, (struct sockaddr *)&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,
                    (struct sockaddr *)&server, &length) < 0 )
           { perror("Recvfrom");
             exit(1);
           }
    else
          printf("->%s\n",buf);

    close(sock);
}
```

Example 2: connectionless server with a program loop

```
#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, (struct sockaddr *)&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,
              (struct sockaddr *)&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 terminated or
       * reaches its normal end.
       */
}
```

The following steps are executed in the program example:

– The server creates a communications endpoint (socket) and corresponding descriptor with the *socket()* function.

– 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.

– The *recvfrom()* function can be used to read from a socket of type SOCK_DGRAM. The length of the read message is returned as the result.

– If no message is available, the process is blocked until a message arrives.

The example programs are only valid for the communications domain AF_INET.
If they are modified according to the information in the sections “Socket addressing” on page 13 and “Creating a socket” on page 16, they are also valid for the AF_INET6 domain.

5.4 Connectionless client

The client uses the following socket or POSIX interface functions in this program example:

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

Example: connectionless client

```
#include <stdio.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. The command format is:
 *                   progname hostname
 */

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,
                        (struct sockaddr *)&to, sizeof to) < 0) {
         perror("Sending datagram message");
         exit(1);
    }
    close(sock);
}
```

The following steps are executed in the program example:

– The client creates a communications endpoint (socket) and corresponding descriptor with *socket()*.

– The client gets the address of the host with *gethostbyname()* and the host name is passed as a parameter.

– Then the address structure is initialized.

– The client sends a datagram with *sendto()* which returns the number of transferred characters.

– The client closes the socket with *close()*.

The example program is only valid for the communications domain AF_INET.
If it is modified according to the information in the sections “Socket addressing” on page 13 and “Creating a socket” on page 16, it is also valid for the AF_INET6 domain.

Please also make sure that you use the *getipnodebyname()* function (instead of *gethostbyname()* as for AF_INET).

6 SOCKETS(POSIX) user functions

This chapter describes the SOCKETS(POSIX) interface functions for BS2000/OSD.

The first thing described is the format in which the separate functions are described. The subsequent overview collects functions together into task-oriented groups. Finally, all SOCKETS interface functions are described in alphabetic order.

The functions for handling file descriptors are provided by the POSIX interface. The functions concerned are *read()*, *readv()*, *write()*, *writev()*, *ioctl()*, *fcntl()* and *close()* as well as *poll()* and *select()*. These functions are described in the manual "C Library Functions (BS2000/OSD) for POSIX Applications". Special points on using these functions with sockets are described at the end of this chapter on page 128ff.

6.1 Description format

The SOCKETS(POSIX) user functions are described in a uniform format. The function descriptions have the following format:

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

Listing and description of the error codes in the external variable *errno* that can occur with a faulty call or function. This section may be missing.

Note

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

See also

Cross references to function descriptions, files, other manual sections or other manuals. Reference is made to the manual “C Library Functions (BS2000/OSD) for POSIX Applications” [1]. This section may be missing.

6.2 Overview of functions

The following overview of the SOCKETS interface functions collects several functions together into task-oriented groups. “[1]” is used in the following overview to refer to the manual “C Library Functions (BS2000/OSD) for POSIX Applications”.

The columns INET and INET6 indicate the address family (AF_INET and/or AF_INET6) in which the function involved is supported.

Setting up and shutting down connections over sockets

Function	Description	See	INET	INET6
socket()	Create socket	page 124	x	x
bind()	Assign a name to a socket	page 79	x	x
connect()	Initiate communication over a socket (e.g. by a client)	page 82	x	x
listen()	Test socket for pending connections (e.g. by a server)	page 114	x	x
accept()	Accept connection over a socket (e.g. by a server)	page 77	x	x
close()	Close socket	[1] and page 129	x	x
shutdown()	Shut full duplex connection down	page 123	x	x
socketpair()	Create a pair of connected sockets	page 126	x	

Transferring data between two sockets

Function	Description	See	INET	INET6
read(), readv()	Receive a message from a socket with a connection already set up	[1] and page 141	x	x
recv()	Receive a message from a socket with a connection already set up	page 116	x	x
recvfrom()	Receive a message from a socket	page 116	x	x
recvmsg()	Receive a message from a socket	page 116	x	x
send()	Send a message from socket to socket over a connection	page 119	x	x
sendto()	Send a message from socket to socket	page 119	x	x
sendmsg()	Send a message from socket to socket	page 119	x	x
write(), writev()	Send a message from socket to socket over a connection	[1] and page 146	x	x
poll()	Multiplex input/output	page 138	x	x
select()	Multiplex input/output	page 143	x	x

Receiving information about sockets

Function	Description	See	INET	INET6
getsockopt()	Get socket options	page 106	x	x
setsockopt()	Set socket options	page 106	x	x
getpeername()	Get name of communications partner	page 100	x	x
getsockname()	Get name of socket	page 105	x	x

Test configuration values

Function	Description	See	INET	INET6
gai_strerror()	Get description of a *getaddrinfo()* error code	page 87	x	x
getaddrinfo()	Get information about host name, host address and services	page 88	x	x
gethostbyaddr()	Get names of reachable hosts	page 91	x	
gethostbyname()	Get addresses of reachable hosts	page 91	x	
gethostname()	Get name of current host	page 93	x	x
getipnodebyaddr()	Get host name belonging to an IPv4 or IPv6 address	page 94	x	x
getipnodebyname()	Get IPv4 or IPv6 address belonging to a host name	page 94	x	x
getnameinfo()	Get host and service name corre-sponding to IP address and port number	page 96	x	x
getnetbyaddr()	Get name of a net	page 98	x	x
getnetbyname()	Get net address	page 98	x	x
getprotobyname()	Get number of a protocol	page 101	x	x
getprotobynumber()	Get name of a protocol	page 101	x	x
getservbyname()	Get port number of a service	page 103	x	x
getservbyport()	Get name of a service	page 103	x	x
sethostent()	Open host database	page 91	x	x
gethostent()	Read entry from host database	page 91	x	x
endhostent()	Close host database	page 91	x	x
setnetent()	Open network database	page 98	x	x
getnetent()	Read entry from network database	page 98	x	x
endnetent()	Close network database	page 98	x	x
setprotoent()	Open protocol database	page 101	x	x
getprotoent()	Read entry from protocol database	page 101	x	x
endprotoent()	Close protocol database	page 101	x	x
setservent()	Open services database	page 103	x	x
getservent()	Read entry from services database	page 103	x	xx
endservent()	Close services database	page 103	x	x

Manipulate Internet address

Function	Description	See	INET	INET6
inet_addr()	Convert character string from dotted notation to integer value (Internet address)	page 109	x	
inet_network()	Convert character string from dotted notation to integer value (subnetwork section)	page 109	x	
inet_makeaddr()	Create Internet address from subnetwork section and subnetwork local address section	page 109	x	
inet_lnaof()	Extract local network address in byte order of host from Internet host address	page 109	x	
inet_netof()	Extract network number in byte order of host from Internet host address	page 109	x	
inet_ntoa()	Convert Internet host address into a string conforming to normal Internet dotted notation	page 109	x	
inet_pton()	Converts an IP address in dotted or colon notation to the corresponding binary address.	page 112	x	x
inet_ntop()	Converts a binary IP address to dotted or colon notation.	page 112	x	x

Utility functions

Function	Description	See	INET	INET6
freeaddrinfo()	Release memory area for an *addrinfo* structure	page 85	x	x
freehostent()	Release memory area for a *hostent* structure	page 86	x	x
htonl()	Convert 32-bit fields from host to network byte order	page 81	x	
htons()	Convert 16-bit fields from host to network byte order	page 81	x	x
ntohl()	Convert 32-bit fields from network to host byte order	page 81	x	
ntohs()	Convert 16-bit fields from network to host byte order	page 81	x	x

Control functions

Function	Description	See	INET	INET6
fcntl()	Control sockets	[1] and page 130	x	x
ioctl()	Control sockets	[1] and page 132	x	x

Test macros for AF_INET6

The following test macros for the AF_INET6 address family are defined in <netinet/in.h>. The parameter *p* is an *in6_addr* structure.

Function	Description
IN6_IS_ADDR_UNSPECIFIED (p)	IPv6 address = 0 ?
IN6_IS_ADDR_LOOPBACK (p)	IPv6 address = loopback ?
IN6_IS_ADDR_LINKLOCAL (p)	IPv6 address = linklocal ?
IN6_IS_ADDR_SITELOCAL (p)	IPv6 address = sitelocal ?
IN6_IS_ADDR_V4MAPPED (p)	IPv6 address = IPv4-mapped ?
IN6_IS_ADDR_V4COMPAT (p)	IPv6 address = IPv4-compatible ?
IN6_ARE_ADDR_EQUAL	address1 = address2 ?

6.3 Functions

This section describes all user functions of the SOCKETS interface in alphabetic order.

accept() - accept a connection over a socket

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

int accept(int s, struct sockaddr *addr, size_t *addrlen);
```

Description

The server process accepts a connection, which was requested by the client with the *connect()* function, over a socket with the *accept()* function. *accept()* can only be used with the connection-oriented socket type SOCK_STREAM.

The *s* parameter designates the socket which waits for a connection request after *listen()* is called.
After returning from *accept()*, *addr* points to the address of the partner application as it is 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 section “Socket addressing” on page 13 describes how you assign an address to the socket.
addrlen points to a *size_t* object that holds the size of the memory area referenced by *addr* at the time of the *accept()* call. When the *accept()* function returns, the *size_t* object (i.e. **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 with the same properties as socket *s*.
3. *accept()* returns the descriptor of the new socket as its result. If socket *s* is non-blocking, neither is the new socket (see the *fcntl()* function on page 130).

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

– If the socket is **not** marked as non-blocking, *accept()* blocks the calling process until a connection is possible.

– If the socket is marked as non-blocking, *accept()* returns the error message EWOULDBLOCK.

The user can initially call *select()* or *poll()* 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 accepted a connection for socket *s*, data can be exchanged between the new socket created by *accept()* and the socket that requested the connection. Additional connections cannot be set up over 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

EBADF
s is not a valid descriptor.

EFAULT
addr does not point to the writable part of the user address range.

EINTR
The *accept()* function was interrupted by a signal that was received before a connection request was received.

EINVAL
The socket does not accept any connection requests.

EMFILE
OPEN_MAX file descriptors are currently open in the calling process.

ENETDOWN
The connection to the network is down.

ENOBUFS
No buffer space is available.

ENOTSOCK
The descriptor does not refer to a socket.

EOPNOTSUPP
The referenced socket is not of type SOCK_STREAM.

EPROTO
A protocol error occurred.

EWOULDBLOCK
The socket is not marked as non-blocking and no free connections are available.

See also

bind(), connect(), listen(), socket(); select() [1]

bind() - assign a socket a name

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

int bind(int s, const struct sockaddr *name, size_t 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()*. *name* points to the name (address) assigned to the socket. *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

EACCES
: The specified name is protected and the calling user has no rights to access it.

EADDRINUSE
: The specified name is already in use.

EADDRNOTAVAIL
: The specified name cannot be bound to the socket by the local system (see also the section “Dependencies of the BS2000/OSD BCAM transport system” on page 302 for more information).

EAFNOSUPPORT
: The specified address family does not match that of the socket.

EBADF
: *s* is not a valid descriptor.

EFAULT
name does not point to the writable part of the user address range.

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.

ENOBUFS
Not enough resources to execute *bind()*.

ENOTSOCK
The descriptor references a file and not a socket.

If the address family of the socket is AF_UNIX, executing bind() can also lead to an error for the following reasons:

EACCES
The specified name is protected or the calling user has no write rights for the specified name.

EDESTADDRREQ
The *name* parameter is the null pointer.

ENAMETOOLONG
A path name component exceeds NAME_MAX characters or the complete path name is longer than PATH_MAX characters.

ENOENT
A path name component refers to a non-existent file or the path name is blank.

ENOTDIR
A path name component is not a directory.

See also

connect(), getsockname(), listen(), socket();
unlink() [1]

Byte order macros - convert byte order

```
#include <arpa/inet.h>

in_addr_t htonl(in_addr_t hostlong);
in_port_t htons(in_port_t hostshort);
in_addr_t ntohl(in_addr_t netlong);
in_port_t ntohs(in_port_t netshort);
```

Description

The *htonl()*, *htons()*, *ntohl()* and *ntohs()* macros convert shorts and integers from host byte order to network byte order and viceversa.
The data type definitions in *in_addr_t* and *in_port_t* in <arpa/inet.h> correspond to the definitions in <netinet/in.h>.

These macros are mainly used in conjunction with IPv4 addresses and port numbers, e.g. as returned by the *gethostbyname()* and *getservent()* functions (see pages 91 and 103). The macros are only needed on systems on which the host and network byte orders are different and are provided in the <arpa/inet.h> header file as null macros (macros without functions):

- *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.

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(), gethostent(), getservent()

connect() - initiate a connection over a socket

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

int connect(int s, const struct sockaddr *name, size_t namelen);
```

Description

A process uses *connect()* to initiate communications with another process over a socket.

The *s* parameter designates the socket over which the process initiates communications with another process.
name is a pointer to the address of the communications partner. **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 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 process uses *connect()* to initiate a connection to a server over a stream socket.
  Stream sockets can generally only set up a connection with *connect()* once.

– With a socket of type SOCK_DGRAM (datagram socket), a process uses *connect()* to define the name of the communications partner with which data is to be exchanged. The process then sends the datagrams to this communications partner. This communications partner is also the only socket from which the process can receive datagrams.
  With datagram sockets, an existing assignment to a partner can be terminated by calling *connect()* with the null pointer as the current parameter for *name*.
  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.

Return value

0:
  If successful.

-1:
  If errors occur. *errno* is set to indicate the error.

Errors

EADDRINUSE
The specified address is already in use.

EADDRNOTAVAIL
The specified address is invalid.

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

EALREADY
This is a non-blocking socket and a previously received connection request has not been concluded yet.

EBADF
s is not a valid descriptor.

ECONNREFUSED
The connection attempt has been successfully rejected. The calling program must close the socket descriptor with *close()* and request a new descriptor by recalling *socket()*. It can then use *connect()* to repeat the connection attempt.

EFAULT
The *name* parameter points to an area outside the process address range.

EINTR
The connection setup attempt was interrupted by a signal.

EINVAL
The *namelen* parameter does not have the size of a valid address for the specified address family.

EISCONN
The socket already has a connection.

ENETUNREACH
The network is not reachable from this host.

ENETDOWN
The connection to the network is down.

ENOBUFS
Not enough resources to execute *connect()*.

ENOTSOCK
Descriptor references a file and not a socket.

ETIMEDOUT
The connection could not be set up within a specific time.

If the socket address family is AF_UNIX, executing connect() can also lead to an error for the following reasons:

EACCES
: Access rights for a path name component were refused or write rights to the specified socket has been refused.

EDESTADDRREQ
: The *name* parameter is the null pointer.

ENAMETOOLONG
: A path name component exceeds NAME_MAX characters or the complete path name is longer than PATH_MAX characters.

ENOENT
: A path name component refers to a non-existent file or the path name is blank.

ENOTDIR
: A component in the path name is not a directory.

See also

accept(), getsockname(), socket();
close(), select() [1]

freeaddrinfo() - release memory for addrinfo structure

```
#include <netdb.h>

void 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 */
   int ai_family;              /* PF_INET, PF_INET6 */
   int ai_socktype;            /* SOCK_STREAM, SOCK_DGRAM */
   int ai_protocol;            /* 0 or IPPROTO_xxx for IP */
   size_t ai_addrlen;          /* length of ai_addr */
   char *ai_canonname;         /* canon name */
   struct sockaddr *ai_addr;   /* socket address structure */
   struct addrinfo *ai_next;   /* next structure in list */
};
```

freehostent() - release memory for hostent structure

```
#include <netdb.h>

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.*

You will find the declaration of the *hostent* structure in section "Converting host names into network addresses and vice versa" on page 42.

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

```
#include <netdb.h>

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 <netdb.h>

int getaddrinfo(char *nodename, char *servname, struct addrinfo *hints,
                struct addrinfo **res);
```

Description

The *getaddrinfo()* function return protocol-independent host information for the AF_INET and AF_INET6 address families. The values are determined using either the Domain Name Service (DNS) or system-specific tables.

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* and *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 */
    int ai_family;                /* AF_INET, AF_INET6 */
    int ai_socktype;              /* SOCK_STREAM, SOCK_DGRAM */
    int ai_protocol;              /* 0 or IPPROTO_xxx for IP */
    size_t ai_addrlen;            /* length of ai_addr */
    char*ai_canonname;            /* canon name */
    struct sockaddr *ai_addr;     /* socket address structure */
    struct addrinfo *ai_next;     /* next structure in 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* and *ai_socktype*:

- *ai_family* = PF_UNSPEC means that any protocol family is desired.
- *ai_socktype* = 0 means that any socket type is accepted.
- *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, the first returned *addrinfo* structure in the element *ai_canonname* contains the socket host name terminated with the null byte of the selected host.
- If the AI_NUMERICHOST 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.

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.

Errors

EAFNOSUPPORT
The function is not supported on this system. See also the section “Dependencies of the BS2000/OSD BCAM transport system” on page 302 for more information.

Error codes defined in <netdb.h>:

EAI_ADDRFAMILY
The address family is 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.

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

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

struct hostent *gethostent(void);

struct hostent *gethostbyname(const char *name);

struct hostent *gethostbyaddr(const void *addr, size_t len, int type);

void sethostent(int stayopen);

void endhostent(void);
```

Description

The *gethostbyname()* and *gethostbyaddr()* functions return current information about the host reachable in the network by calling a BCAM information interface. The DNS concept (Domain Name Service) is supported if the DNS Resolver from the product *inter*Net Services (formerly TCP-IP-SV) is installed in POSIX or the subsystem SOCKETS (BS2000) has been started.
In contrast to this, the *gethostent()* function accesses the UFS */etc/inet/hosts* file which normally only contains an entry for the local host.

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

The *hostent* structure corresponds to the fields in a line of the host database and is declared 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 by the null pointer */
};
```

hostent components:

h_name
Name of the host

h_aliases
A list of alternative (alias) names for the host, terminated with null.
Alias names are currently not supported.

h_addrtype
Type of the returned address (always AF_INET)

length
Length of the address in bytes

**h_addr_list
A pointer to a list of network addresses for the host. The addresses are returned in network byte order.

In the case of *gethostbyaddr()*, *addr* is a pointer to the address in binary format with the length *len* (not a character string).

gethostent() reads the next line of the file. If necessary, *gethostent()* opens the file first.

sethostent() opens the file and resets it to the start. If the *stayopen* flag is not equal to zero, the database is not closed after any *gethostent()* call (neither directly nor indirectly via one of the other *gethost...()* calls).

Return value

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

Note

All information is in a static area and therefore has to be copied if it is to be saved.
Only the Internet address format is supported.

gethostname() - get the name of the current host

```
#include <unistd.h>

int gethostname(char *name, size_t namelen);
```

Description

The *gethostname()* function returns the socket host name for the current host in the *name* parameter. The length of the *name* string variable must be specified in the *namelen* parameter when *gethostname()* is called.

If the length of the *name* string variable specified by *namelen* suffices for storing the host name, the host name is terminated with a null byte. Otherwise, the excess socket 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 set to indicate the error.

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

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

struct hostent *getipnodebyaddr(char *addr,size_t len,int af,int *err);
struct hostent *getipnodebyname(char *name,int af,int flags,int *err);
```

Description

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 via the DNS Resolver integrated in SOCKETS(BS2000) V2.0. If this is not successful, the information is taken from the BCAM processor table (see the "openNet Server V3.0 (BS2000/OSD)" manual).

For *getipnodebyaddr()*, *addr* is a pointer to the host address. This host address must be available in binary format with the length *len*.

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.fujitsu-siemens.com) 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.

The *af* parameter in the call is used to specify the address family (AF_INET or AF_INET6). AF_UNSPEC can also be specified for *getipnodebyname* if an IP address is specified as *name* in dotted or colon notation.

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.

AI_V4MAPPED
: The caller accepts IPv4-mapped addresses if no IPv6 address is available.

AI_ALL
: Only if AI_V4MAPPED is also set: 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 type *struct hostent*. Memory for this object is requested dynamically and must be released again by the caller with the *freehostent()* function.

The *hostent* structure is described in section “Converting host names into network addresses and vice versa” on page 42.

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:

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.

-1:
[as an alternative value for the variable *err*]
If errors occur. *errno* is set to indicate the error.

Errors

EAFNOSUPPORT
The function is not supported on this system. See also the section “Dependencies of the BS2000/OSD BCAM transport system” on page 302 for more information.

getnameinfo() - get name of the communications partner

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


int getnameinfo (struct sockaddr *sa,size_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 either from a DNS server via the DNS Resolver integrated in SOCKETS(BS2000) V2.0 or using system-specific tables.

The *sa* parameter is a pointer to a *sockaddr_in* structure which contains the IP address and port number. The actual format of the *sockaddr* structure depends on the address family involved and is described in section “Socket addressing” on page 13. The exact format of **sa* is determined by the domain in which communications takes place. *salen* indicates the length of the structure.

If *getnameinfo()* is executed successfully, *host* and *serv* are pointers to two areas containing the corresponding null-byte-terminated socket host name and service name respectively. The lengths of the areas are specified in *hostlen* and *servlen* respectively. These areas must be large enough to accommodate the socket host name or service name (including the null byte). If the value 0 is specified for *hostlen* or *servlen* when *getnameinfo()* is called, this indicates that no socket host name or service name is to be returned.

The maximum lengths of the socket host name and service name are defined in <netdb.h>:

```
#define NI_MAXHOST 1025

#define NI_MAXSERV 32
```

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:

NI_NOFQDN
: Only the host name part of the full DNS name (socket host name) is returned.

NI_NUMERICHOST
: The numeric host name is returned in printable format after address conversion. The same is true if it is impossible to determine the host name in the DNS or using local information and NI_NAMEREQD is not set.

NI_NAMEREQD
: An error is reported if the host name cannot be determined in the DNS.

NI_NUMERICSERV
: The port number is returned in printable format instead of the service name.

Return value

0:
: If successful

<> 0:
: If errors occur

Errors

EAFNOSUPPORT
: The function is not supported on this system. See also the section “Dependencies of the BS2000/OSD BCAM transport system” on page 302 for more information.

EINVAL
: Invalid address family specified in the *sa* parameter, or the lengths of the output areas *host* and/or *serv* are too small.

getnetent(), getnetbyname(), getnetbyaddr(), setnetent(), endnetent() - get information about network names

```
#include <netdb.h>

struct netent *getnetent(void);

struct netent *getnetbyname(const char *name);

struct netent *getnetbyaddr(in_addr_t net, int type);

void setnetent(int stayopen);

void endnetent(void);
```

Description

The *getnetbyname()*, *getnetbyaddr()* and *getnetent()* functions return information about the names and addresses of the reachable networks. The information about local network names is not available in BCAM. Assignment is by way of a UFS */etc/inet/networks* file as normal in UNIX systems. The file contents are coded in EBCDIC.

The *getnetbyname()*, *getnetbyaddr()* and *getnetent()* functions return a pointer to an object with the *netent* structure described below.

The *netent* structure corresponds to the fields of a line in the network database and is declared as follows:

```
struct netent {
   char *n_name;           /* Official name of the network */
   char **n_aliases;       /* Alias list */
   int n_addrtype;         /* Address type */
   in_addr_t n_net;        /* Network address */
};
```

netent components:

n_name
: Official name of the network

n_aliases
: A list of alternative (alias) names for the host, terminated with null

n_addrtype
Type of the returned address (always AF_INET)

n_net
Address of the network, which is returned in host byte order.

getnetent() reads the next line of the file. If necessary, *getnetent()* opens the file first.

setnetent() opens the file and resets it to the start. If the *stayopen* flag is not equal to zero, the database is not closed after any *getnetent()* call (neither directly nor indirectly via one of the other *getnet...()* calls).

endnetent() closes the file.

getnetbyname() and *getnetbyaddr()* search sequentially through the file from the start until

- a matching name is found or
- the matching network address is found or
- the end of the file reached.

Return value

The null pointer is returned if the search reaches the end of the file.

Note

All information is in a static area and therefore has to be copied if it is to be saved.
Only the Internet protocols are supported.

See also

/etc/inet/networks

getpeername() - get the name of the communications partner

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

int getpeername(int s, struct sockaddr *name, size_t *namelen);
```

Description

The *getpeername()* returns the name of the communications partner connected to socket *s*.

name points to a memory area. After *getpeername()* has been executed successfully, **name* contains the address of the communications partner.

The *size_t* variable, to which the *namelen* parameter points, initially indicates the size of the memory area referenced by *name*. 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

EBADF
: The *s* parameter is not a valid descriptor.

EFAULT
: The *name* parameter points to an area outside the process address range.

ENOTCONN
: The socket has no connection.

ENOTSOCK
: Descriptor *s* references a file and not a socket.

See also

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

getprotoent(), getprotobynumber(), getprotobyname(), setprotoent(), endprotoent() - get information about protocols

```
#include <netdb.h>

struct protoent *getprotoent(void);

struct protoent *getprotobyname(const char *name);

struct protoent *getprotobynumber(int proto);

void setprotoent(int stayopen);

void endprotoent(void);
```

Description

The *getprotobyname()*, *getprotobynumber()* and *getprotoent()* functions return information about the available services. These functions access the UFS */etc/inet/protocols* file. The interface is provided for compatibility reasons. The contents of the file are coded in EBCDIC.

The *getprotobyname()*, *getprotobynumber()* and *getprotoent()* functions return a pointer to an object with the *protoent* structure described below.

The *protoent* structure corresponds to the fields of a line in the protocol */etc/inet/protocols* database and is declared as follows:

```
struct protoent {
   char *p_name;           /* Official name of the protocol*/
   char **p_aliases;       /* Alias list */
   int p_proto;            /* Protocol number */
};
```

protoent components:

p_name
: Name of the protocol

p_aliases
: A list of alternative (alias) names for the protocol, terminated with null

p_proto
: Number of the protocol

getprotoent() reads the next line from the file. If necessary, *getprotoent()* opens the file first.

getprotoent() opens the file and resets it to the start. If the *stayopen* flag is not equal to zero, the database is not closed after any *getprotoent()* call (neither directly nor indirectly via one of the other *getproto...()* calls).

endprotoent() closes the file.

getprotobyname() and *getprotobynumber()* search sequentially through the file from the start until

– a matching protocol name is found or
– the matching protocol number is found or
– the end of the file is reached.

Return value

The null pointer is returned if the search reaches the end of the file.

Note

All information is in a static area and therefore has to be copied if it is to be saved.

See also

/etc/inet/protocols

getservent(), getservbyport(), getservbyname(), setservent(), endservent() - get information about services

```
#include <netdb.h>

struct servent *getservent(void);

struct servent *getservbyname(const char *name, const char *proto);

struct servent *getservbyport(int port, const char *proto);

void setservent(int stayopen);

void endservent(void);
```

Description

The *getservbyport()*, *getservbyname()* and *getservent()* functions return information about the available services. Each of these functions returns a pointer to an object with the *servent* structure described below.

The *servent* structure corresponds to the fields of a line in the service */etc/inet/services* database and is defined 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 */
};
```

servent components:

s_name
: Name of the service

s_aliases
: A list of alternative (alias) names 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.

getservent() reads the next line in the file. If necessary, *getservent()* opens the file first.

getservent() opens the file and resets it to the start. If the *stayopen* flag is not equal to zero, the database is not closed after any *getservent()* call (neither directly nor indirectly via one of the other *getserv...()* calls).

endservent() closes the file.

getservbyname() and *getservbyport()* search sequentially through the file from the start until

– a matching service name is found or
– the matching port number is found or
– The end of the file is reached.

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 information about services and their port numbers is not available in BCAM as they are components of OSI layer 7. Since the assignment of port numbers to services is static, the implementation can be solved by entering the services in a UFS */etc/inet/services* file (as is normal in UNIX systems).

All information is in a static area and therefore has to be copied if it is to be saved.

See also

getprotoent(), /etc/inet/services

getsockname() - get the name of a socket

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

int getsockname(int s, struct sockaddr *name, size_t *namelen);
```

Description

The *getsockname()* function returns the current name for socket *s*.

name points to a memory area. After successful execution of *getsockname()*, **name* contains the name (address) of socket *s*. The actual format of the *sockaddr* structure depends on the address family involved and is described in the section "Socket addressing" on page 13.

The *size_t* variable to which the *namelen* parameter points initially indicates the size of the memory area referenced by *name*. When the function returns, **name* 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

EBADF
: The *s* parameter is not a valid descriptor.

EFAULT
: The *name* parameter points to an area outside the process address range.

ENOTSOCK
: Descriptor *s* references a file and not a socket.

See also

bind(), getpeername(), socket()

getsockopt(), setsockopt() - get and set socket options

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

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

int setsockopt(int s, int level, int optname, const void *optval,
                                                        size_t optlen);
```

Description

The *getsockopt()* and *setsockopt()* functions access options which are defined for a socket. Options can exist on various protocol levels but they always exist on the highest protocol level.

The name *optname* of the option and the protocol level *level* on which the protocol is interpreted must always be specified for accessing a socket option. The user must specify SOL_SOCKET or IPPROTO_IPV6, as appropriate, for *level* to access options on the socket level.

With the *setsockopt()* function, the user can access option values via the *optval* and *optlen* parameters. With the *getsockopt()* function, *optval* and *optlen* identify a buffer in which the values of the desired option(s) are returned. With *getsockopt()*, **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. **optval* contains the value 0 if the option has no value that can be returned.

optname and all specified options are passed without conversion to the relevant protocol module for interpretation. The <sys/socket.h> header file contains definitions for the socket level options. The options are described on page 107.

optval is a pointer to a parameter of type integer for most of the socket level options. The *optval* parameter may not be set to the null pointer if a boolean value is to be allowed for *setsockopt()*. If a boolean operation is not to be allowed, the *optval* parameter must be set to the null pointer. The SO_LINGER option uses a parameter of data type *struct linger*, which is defined in the <sys/socket.h> header file. This parameter specifies the desired option status and the delay interval (see page 107).

The options described below are known on the socket level. You can get and set each option with *getsockopt()* and *setsockopt()* respectively unless something different is stated in the option description.

Socket options

SO_KEEPALIVE
Specifies whether connections are to be kept open or not.
SO_KEEPALIVE causes regular transfer of control messages over a connected socket. If the connected partner end system cannot answer the message, the connection is taken to be broken. A process which is waiting to write to the socket receives a SIGPIPE signal and the write operation returns an error. By default, a process terminates when it receives a SIGPIPE signal. A read operation to the socket returns an error but does not generate a SIGPIPE signal. If the process waits for a *select()* call with an interrupted connection, *select()* returns the value *true* for all read or write events that are selected for the socket.

SO_LINGER
Defines whether socket closing after a *close()* call is delayed if data is still pending to be transferred on the socket. SO_LINGER controls the action that is triggered when non-transferred messages are waiting in the socket queue and the *close()* function is called. If the socket ensures secured data transfer and SO_LINGER is set, the system blocks the process that is trying to close the socket. The timeout interval is defined when *setsockopt()* is called if SO_LINGER is enabled. If SO_LINGER is disabled when the *close()* function is called, the system executes *close()* immediately and the process can be continued as quickly as possible.

SO_BROADCAST
Defines whether broadcast messages may be transferred or not.
Since broadcast messages may always be sent in BS2000, this option has no functional significance.
However, it must be noted that the reception of broadcast messages can be disallowed (see the BCAM BCOPTION command in the “openNet Server V3.0 (BS2000/OSD)” manual).

SO_REUSEADDR
Specifies that the rules for the validity check on the addresses specified for *bind()* should permit the reuse of local addresses provided this is supported by the protocol. An integral value (*int*) is required for this option.

SO_TYPE
Gets the socket type.
SO_TYPE is only used by *getsockopt()*. SO_TYPE returns the type of the socket, i.e. either SOCK_STREAM or SOCK_DGRAM. This can be useful for servers that inherit sockets when they are started.

SO_ACCEPTCONN
Indicates whether the socket is ready to receive connection requests.
SO_ACCEPTCONN can only be used with *getsockopt()*.

Return value

0:
: If successful.

-1:
: If errors occur. *errno* is set to indicate the error.

Errors

EBADF
: The *s* parameter is not a valid descriptor.

EFAULT
: *optval* does not point to a valid part of the process address range in the length *optlen*.

EINVAL
: One of the parameters *level* , *optval* or *optlen* has an illegal value.

ENOPROTOOPT
: The option is not known to the designated level.

ENOTSOCK
: Descriptor *s* does not reference a socket.

EOPNOTSUPP
: The option is not supported.

See also

socket(), getprotoent()

inet_addr(), inet_network(), inet_makeaddr(), inet_lnaof(), inet_netof(), inet_ntoa() - manipulate IPv4 Internet address

```
#include <arpa/inet.h>

in_addr_t inet_addr(const char *cp);

in_addr_t inet_lnaof(struct in_addr in);

struct in_addr inet_makeaddr(in_addr_t net, in_addr_t lna);

in_addr_t inet_netof(struct in_addr in);

in_addr_t inet_network(const 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 period notation into 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 period notation into 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 period 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 period 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 three-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 period 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, (*in_addr_t*)-1 is returned.

After successful execution, *inet_network()* returns the converted Internet number.
Otherwise, (*in_addr_t*)-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 period notation.

Errors

No errors are defined.

Note

The return value of *inet_ntoa()* may point to static data which can be overwritten by subsequent *inet_ntoa()* calls.

See also

gethostent(), getnetent()

inet_ntop(), inet_pton() - manipulate Internet addresses

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

char *inet_ntop(int af,void *addr,char *dst,size_t size);
int inet_pton(int af,char *addr,void *dst);
```

Description

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:

– AF_INET: a binary IPv4 address is converted.
– AF_INET6: a binary IPv6 address is converted.

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). 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:

– AF_INET: an IPv4 address is converted.
– AF_INET6: 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.

Note

If the output of *inet_pton()* is to be 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_ntop()* 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_pton() 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
    Invalid *af* parameter specified, or the function is not supported on this system. See also the section “Dependencies of the BS2000/OSD BCAM transport system” on page 302 for more information.

ENOSPC
    The result buffer is too small.

listen() - test a socket for pending connections

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

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

Description

The *listen()* function authorizes socket *s* for accepting 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 defines the maximum number of connection requests that the queue can hold with the *backlog* parameter. The value of *backlog* is limited to a maximum of 50.
The *listen()* function can only be called for type SOCK_STREAM sockets.

The following steps are required to enable a process to communicate over a socket with the partner who 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()*.

If a connection request arrives and the queue is full, the socket that sent the connection request receives the error message ECONNREFUSED or ETIMEDOUT.

Return value

0:
  If successful.

-1:
  If errors occur. *errno* is set to indicate the error.

Errors

EBADF
: The *s* parameter is not a valid descriptor.

ENOTSOCK
: Descriptor *s* references a file and not a socket.

EOPNOTSUPP
: The socket type is not supported by *listen()*.

See also

accept(), connect(), socket()

recv(), recvfrom(), recvmsg() - receive a message from a socket

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

ssize_t recv(int s, void *buf, size_t len, int flags);

ssize_t recvfrom(int s, void *buf, size_t len, int flags,
                                    struct sockaddr *from, size_t *fromlen);

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

Description

The *recv()*, *recvfrom()* and *recvmsg()* functions receive messages from a socket.
recv() can only receive messages from a socket over which a connection is set up (see the *connect()* function on page 82).
recvfrom() and *recvmsg()* can receive messages from a socket with or without a connection.

The *s* parameter designates the socket from which the message is received.
If the *from* parameter is not the null pointer, the address of the message sender is stored in the address area referenced by *from*.
fromlen is a result parameter. The *size_t* variable, to which *fromlen* points, initially holds 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 complete message must be read in a single operation for a datagram socket. If the specified message buffer is too small and MSG_PEEK is not set in the *flags* parameter, data extending beyond the buffer size is deleted.
Message limits are ignored with a stream socket. A 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 *ioctl()* on page 132). In this case, -1 is returned and the *errno* variable is set to the value EWOULDBLOCK.

The *poll()* or *select()* functions can be used to determine when further data arrives.

If the process which calls *recv()*, *recvfrom()* or *recvmsg()* receives a signal before any data is available, the function concerned is recalled in a standard case. However, the function is not recalled if the calling process uses *sigaction()* to signal interruption of this call (see also the manual “C Library Functions (BS2000/OSD) for POSIX Applications”).

The *flags* parameter indicates the type of message reception.

MSG_PEEK
: Receives an incoming message. However, the data is handled as unread and the next receive function also returns this data.

MSG_WAITALL
: The function blocks until the entire data that was requested can be returned.
A smaller amount of data can be returned in the following cases:

- A signal arrives.
- The connection is closed.
- An error condition occurs.

The *recvmsg()* function uses the *msghdr* structure to reduce the number of directly supplied parameters. The *msghdr* structure is declared as follows in the <sys/socket.h> header file:

```
struct msghdr {
   void          *msg_name;            /* Optional address */
   size_t         msg_namelen;         /* Length of address */
   struct iovec  *msg_iov;             /* Scatter/gather fields */
   int            msg_iovlen;          /* Number of members in msg-iov */
   caddr_t        msg_accrights;       /* Send/receive access rights */
   int            msg_accrightslen;
   void          *msg_control;         /* Auxiliary data */
   size_t         msg_controllen;      /* Length of auxiliary data buffer */
   int            msg_flags;           /* Flag for received message */
};
```

The members *msg_name* and *msg_namelen* contain the sending address and the address length of the partner if the socket has no connection set up. The partner address is a *sockaddr* structure. The actual format of the *sockaddr* structure depends on the address family involved and is described in the section “Socket addressing” on page 13. If the socket has a connection set up, *msg_name* can be passed as a null pointer.

The members *msg->msg_iov* and *msg->msg_iovlen* describe the scatter and gather fields. Sending and receiving access rights is not supported.

Return value

>0:
: If successful. The value indicates the number of received bytes.

=0:
: If successful. No more data can be received. The partner has closed his connection correctly (only with type SOCK_STREAM sockets).

-1:
If errors occur. *errno* is set to indicate the error.

Errors

EBADF
The *s* parameter is not a valid descriptor.

ECONNRESET
The connection to the partner was interrupted (only with type SOCK_STREAM sockets).

EFAULT
The data is to be received in a non-existent or protected part of the process address range.

EINTR
The calling process has received a signal before any data could be received. The signal to interrupt the function call is set.

EINVAL
More than MSG_MAXIOVLEN scatter/gather fields were specified.

EIO
User data has been lost.

ENETDOWN
The connection to the network is down.

ENOTCONN
No connection exists for the socket.

ENOTSOCK
Descriptor *s* references a file and not a socket.

EOPNOTSUPP
The *flags* parameter contains an illegal value or *msg->msg_accrights* was specified.

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

See also

connect(), getsockopt(), send(), socket();
fcntl(), ioctl(), read(), select() [1]

send(), sendto(), sendmsg() - send a message from socket to socket

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

ssize_t send(int s, const void *msg, size_t len, int flags);

ssize_t sendto(int s, const void *msg, size_t len, int flags,
                                const struct sockaddr *to, size_t tolen);

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

Description

The *send()*, *sendto()* and *sendmsg()* functions send messages from one socket to another. *send()* can only be used with a socket over which a connection is set up (see the *connect()* function on page 82). *sendto()* and *sendmsg()* can always be used.

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 EMSGSIZE is returned and the message is not transferred.

The *flags* parameter is currently not supported. A value not equal to 0 leads to an error and the *errno* variable is set to the value EOPNOTSUPP.

If the process which calls *send()*, *sendmsg()* or *sendto()* receives a signal before any send data is buffered, the function concerned is normally called again. However, the function is not recalled if the calling process has used *sigaction()* to set the signal to interrupt this call (see also the manual “C Library Functions (BS2000/OSD) for POSIX Applications”).

The *sendmsg()* function uses the *msghdr* structure to reduce the number of directly supplied parameters.

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

```
struct msghdr {
   void          *msg_name;          /* Optional address */
   size_t         msg_namelen;       /* Length of the address */
   struct iovec *msg_iov;           /* Scatter/gather fields */
   int            msg_iovlen;        /* Number of members in msg-iov */
   caddr_t        msg_accrights      /* Send/receive access rights */
   int            msg_accrightslen;
   void          *msg_control;       /* Auxiliary data */
   size_t         msg_controllen;    /* Length of auxiliary data buffer */
   int            msg_flags;         /* Flag for received message */
};
```

msg->msg_name and *msg->msg_namelen* specify the destination address if the socket has no connection set up. *msg->msg_name* can be passed as a null pointer if no names are desired or requested. You will find a description of how to assign the socket an address in the section "Socket addressing" on page 13.
msg->msg_iov and *msg->msg_iovlen* describe the scatter and gather fields. Sending and receiving access rights is not supported.

Return value

≥0:
: If successful. The value indicates the number of sent bytes.

-1:
: If errors occur. *errno* is set to indicate the error. The descriptor sets are then not changed.

Errors

EBADF
: The *s* parameter is not a valid descriptor.

ECONNRESET
: The connection to the partner was interrupted (only with type SOCK_STREAM sockets).

EDESTADDRREQ
: The socket is not connection-oriented, a permanent partner was not defined and no partner was specified in the call.

EFAULT
: The data is to be stored in a non-existent or protected part of the process address range.

EHOSTUNREACH
The destination host cannot be reached.

EINTR
The calling process received a signal before any data could be buffered for sending and the signal to interrupt the function call is set.

EINVAL
A parameter specified an illegal value.

EMSGSIZE
The message is too long to be sent in one piece.

ENETDOWN
The connection to the network is down.

ENOBUFS
The output queue for a network interface is full. This generally leads to the interface stopping sending, but can also be due to a temporary jam.

ENOTCONN
No connection exists for the socket.

ENOTSOCK
The descriptor *s* does not reference a socket.

EOPNOTSUPP
The *flags* or *msg->msg_accrights* parameter was specified, and this is not supported.

EPIPE
The socket is not enabled for writing or the socket is connection-oriented and the partner has shut the connection down.
If the socket is of type SOCK_STREAM, the SIGPIPE signal is generated for the calling process.

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

EAFNOSUPPORT
Addresses of the address family specified for *sendto()* or *sendmsg()* cannot be used with this socket.

If the socket address family is AF_UNIX, execution of send(), sendto() and sendmsg() can also lead to an error for the following reasons:

EACCES
: Access rights are refused for a path name component or write rights to the specified socket are refused.

ENAMETOOLONG
: A path name component exceeds NAME_MAX characters or the complete path name is longer than PATH_MAX characters.

ENOENT
: A path name component refers to a non-existent file or the path name is blank.

ENOTDIR
: A path name component is not a directory.

See also

connect(), getsockopt(), recv(), socket();
fcntl(), select(), write() [1]

shutdown() - close full duplex connection

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

int shutdown(int s, int how);
```

Description

The *shutdown()* function causes either one or both ends of a full duplex connection over a socket to be shut down. The *s* parameter designates the socket concerned. *shutdown()* has the following effects, depending on the value of the *how* parameter:

– If the *how* parameter has the value SHUT_RD, *shutdown()* prevents further messages from being received..

– If the *how* parameter has the value SHUT_WR, *shutdown()* prevents messages being sent.

– If the *how* parameter has the value SHUT_RDWR, *shutdown()* prevents messages being either sent or received.

Return value

0:
If successful

-1:
If errors occur. *errno* is set to indicate the error.

Errors

EBADF
The *s* parameter is not a valid descriptor.

ENOTSOCK
Descriptor *s* references a file and not a socket.

ENOTCONN
The socket has no connection.

See also

connect(), socket()

socket() - create socket

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

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. The protocol family generally corresponds to 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_UNIX protocol families are supported.

The *type* parameter defines the type of the socket and the semantics of the communications. The two following socket types are currently defined:

- SOCK_STREAM
- SOCK_DGRAM

The type SOCK_STREAM socket provides a sequential, secured, bidirectional connection. A socket of type SOCK_DGRAM supports the transfer of datagrams, which are connectionless, unsecured messages with a fixed maximum length.

The *protocol* parameter defines a specific protocol that is to be used for the socket. Since this implementation only supports the TCP/IP protocol family, only the values 0 (standard protocol), IPPROTO_IP, IPPROTO_IPV6, IPPROTO_TCP and IPPROTO_UDP are valid here.

Sockets of type SOCK_STREAM are full duplex data streams, similar to pipes. A stream socket must be in a connected state before any data can be sent or received over it. A connection to another socket is set up with the *connect()* function. Once two sockets are connected together, data can be transferred with *read()* and *write()* calls or similar calls such as *send()* and *recv()*. The user should call the *close()* function when a session is finished.

The communications protocols used for implementing a type SOCK_STREAM socket ensure that data is not lost or duplicated.

Type SOCK_DGRAM sockets allow the connectionless sending and receiving of datagrams with *sendto()* and *recvfrom()* or *sendmsg()* and *recvmsg()*. When these functions are called, the address of the communications partner is passed as a parameter.

The user can specify a process group with the *fcntl()* function to receive a SIGIO signal when input/output operations or connection setup requests arrive.

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.

Return value

≥0:
: Designates a non-negative descriptor if successful.

-1:
: If errors occur. *errno* is set to indicate the error.

Errors

EACCES
: No allowance is granted for creating a socket of the specified type or protocol.

EMFILE
: The table of descriptors per process is full.

ENFILE
: The system file table is full.

ENOBUFS
: Not enough space in the buffer. The socket cannot be created until enough storage resources are freed.

EPROTONOSUPPORT
: The protocol type or the specified protocol is not supported in this domain.

EPROTOTYPE
: Wrong protocol type for the socket.

EAFNOSUPPORT
: The address family specified in the *domain* parameter is not supported on this system. See also section “Dependencies of the BS2000/OSD BCAM transport system” on page 302 for more information.

See also

accept(), bind(), connect(), getsockname(), getsockopt(), listen(), recv(), send(), shutdown(), socketpair();
close(), fcntl(), ioctl(), read(), select(), write() [1]

socketpair() - create a pair of connected sockets

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

int socketpair(int domain, int type, int protocol, int sv[2]);
```

Description

The *socketpair()* function creates a pair of sockets that are connected with each other but have no names.

socketpair() creates the socket-pair in the address family specified with the *domain* parameter (AF_INET or AF_UNIX), of type *type* (SOCK_STREAM or SOCK_DGRAM) and using the optionally specified protocol *protocol*. The *protocol* parameter defines a specific protocol that is to be used for the socket. Since this implementation only supports the TCP/IP protocol family, only the values 0 (standard protocol), IPPROTO_IP, IPPROTO_TCP and IPPROTO_UDP are valid here.

The descriptors of the new socket are returned in the *sv[0]* and *sv[1]* parameters. The two sockets cannot be distinguished between.

Return value

0:
: If successful

-1:
: If errors occur. *errno* is set to indicate the error.

Errors

EAFNOSUPPORT
: The specified address family is not supported on this system.

EFAULT
: The address *sv* does not specify a valid part of the process address range.

EMFILE
: The table of descriptors per process is full.

ENFILE
: The system file table is full.

EOPNOTSUPP
The specified protocol does not support creating socket-pairs.

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

ENOMEM
An internal resource bottleneck has occurred.

See also

pipe(), read(), write() [1]

6.4 Using standard POSIX functions for sockets

The functions described in this section are standard POSIX library functions. The functions concerned are

- close()
- fcntl()
- ioctl()
- poll()
- read()
- readv()
- select()
- write()
- writev()

Only the particulars for using them with sockets are described in this section.

close() - close socket

```
int close(int s);
```

Description

close() closes socket *s*, depending on the SO_LINGER option (see the *setsockopt()* function on page 106).

Return value

0:
: If successful.

-1:
: If errors occur. *errno* is set to indicate the error.

Errors

EBADF
: The *s* parameter is not a valid descriptor.

fcntl() - control sockets

```
#include <fnctl.h>

int fcntl(int s, int cmd, int arg);
```

Description

The *fcntl()* function also executes control functions for sockets.
s designates the socket descriptor and *cmd* selects the control function to be executed.

The following control functions are supported for sockets:

F_DUPFD
: Duplicates a socket descriptor.

F_GETFD
: Gets the “close with exec” bit that belongs to the descriptor *s*. If the least significant bit is 0, the socket remains open when *exec()* is called, otherwise, the socket is closed when *exec()* is called.

F_SETFD
: Sets the “close with exec” bit belonging to *s* to the least significant bit of the integer value passed as the third parameter (0 or 1 as above).

F_GETFL
: Gets the file status bit for *s*.

F_SETFL
: Sets the file status bit for *s* to the integer value passed as the third parameter. Only specific bits can be set (e.g. O_NONBLOCK for non-blocking sockets).

F_SETOWN
: The process ID can be set for the specified socket, causing a SIGIO signal to be supplied when a message arrives from the process.

F_GETOWN
: Returns the process ID set for the socket.

Return value

0:
If successful

-1:
If errors occur. *errno* is set to indicate the error.

Errors

EBADF
The *s* parameter is not a valid descriptor.

EINVAL
cmd or *arg* are not valid for this device.

EIO
A physical input/output error has occurred.

EMFILE
cmd is F_DUPFD and the number of open file descriptors in the calling process equals the maximum value of open files for each user specified in the configuration.

ioctl() - control sockets

```
#include <sys/sockio.h>
#include <net/if.h>
#include <sys/filio.h>

int ioctl(int s, unsigned long request, char *arg);
```

Description

The *ioctl()* function also executes control functions for sockets.
s designates the socket descriptor. The data type of the object passed as the current parameter for *arg* depends on the control function concerned and is a pointer to either an integer variable (*int*) or a special data structure.Type conversion to “pointer to char” is therefore required when calling *ioctl()*.

The following control functions are supported for sockets:

Request	***arg**	**Function**
FIONBIO	int	Enable/disable blocking mode
FIONREAD	int	Get message length
FIOSETOWN	int	Set process ID
FIOGETOWN	int	Get process ID
SIOCSPGRP	as FIOSETOWN	
SIOCGPGRP	as FIOGETOWN	
SIOCGLIFNUM	struct lifnum	Get number of interfaces
SIOCGLIFCONF	struct lifconf	Get interface configuration
SIOCGLIFADDR	struct lifreq	Get Internet address of the interface
SIOCGLIFINDEX	struct lifreq	Get index of the interface
SIOCGLIFBRDADDR	struct lifreq	Get broadcast address of the interface
SIOCGLIFNETMASK	struct lifreq	Get subnetwork mask of the interface
SIOCGLIFFLAGS	struct lifreq	Get flags of the interface
SIOCGIFNUM	Int	Get interface number (only IPv4)
SIOCGIFCONF	struct ifconf	Get interface configuration (only IPv4)
SIOCGIFADDR	struct ifreq	Get interface Internet address (only IPv4)
SIOCGIFINDEX	struct ifreq	Get index of the interface (only IPv4)

Request	*arg	Function
SIOCGIFBRDADDR	struct ifreq	Get interface broadcast address (only IPv4)
SIOCGIFNETMASK	struct ifreq	Get subnetwork mask of the interface (only IPv4)
SIOCGIFFLAGS	struct ifreq	Get interface flags

FIONBIO
This option affects the execution behavior of socket functions when data flow control is triggered.

– **arg* == 0:
  Socket functions block until the function can be executed.

– **arg* != 0:
  Socket functions return with the *errno* code EWOULDBLOCK if the function cannot be executed immediately due to data flow control.

FIONREAD
Returns the length of the message currently in the input buffer.

FIOSETOWN
The process ID can be set for the specified socket, causing a SIGIO signal to be supplied when a message arrives from the process.

SIOCSPGRP
as FIOSETOWN

FIOGETOWN
Returns the process ID set for the socket.

SIOCGPGRP
as FIOGETOWN

SIOCGLIFNUM
The number of interfaces is returned in the *lifn_count* member. Only the intefaces which belong to the adress family (AF_UNSPEC, AF_INET or AF_INET6) specified in the *lifn_family* member are counted.

SIOCGLIFCONF
A list of the network configuration is returned. For each interface belonging to the address family specified in the *lifc_family* member and for which the flags specified in the *lifc_flags* member are set, an entry of the type *struct lifreq* is written to the area which is addressed by the *lifc_buf* member. If the length of this area (*lifc_len*) is not sufficient, the EINVAL error is reported.
The *lifconf* and *lifreq* structures are defined in the include file <net/if.h>.

SIOCGLIFADDR
The Interface address is returned in the *lifr_addr* member for the interface specified with the *lifr_name* member.

SIOCGLIFINDEX
The index (the interface number) is returned in the *lifr_index* member for the interface specified with the *lifr_name* member.

SIOCGLIFBRDADDR
The broadcast address is returned in the *lifr_broadaddr* member for the interface specified with the *lifr_name* member. For IPv4 interfaces without broadcast capability and for IPv6 interfaces, the EADDRNOTAVAIL error is reported.

SICGLIFNETMASK
The subnetwork mask is returned in the *lifr_addr* member for the interface specified with the *lifr_name* member. For IPv6 interfaces the EADDRNOTAVAIL error is reported.

SIOCGLIFFLAGS
The interface flags are returned in the *lifr_flags* member for the interface specified with the *lifr_name* member. The possible flags are IFF_UP, IFF_LOOPBACK and IFF_BROADCAST.

The folllowing options are supported for reasons of compatibility. However, they only return information on IPv4 interfaces:

SIOCGIFNUM
The number of IPv4 interfaces is returned in the argument.

SIOCGIFCONF
A list of the IPv4 network configuration is returned. For each IPv4 interface an entry of the type *struct ifreq* is written into the area which is addressed by the *ifc_buf* member. If the length of this area (*ifc_len*) is not sufficient, the EINVAL error is reported. The *ifconf* and *ifreq* structures are defined in the include file <net/if.h>.

SIOCGIFADDR
The Internet address is returned in the *ifr_addr* member for the interface specified with the *ifr_name* member.

SIOCGIFINDEX
The index (the interface number) is returned in the *ifr_index* member for the interface specified with the *ifr_name* member.

SIOCGIFBRDADDR
The broadcast address is returned in the *ifr_broadaddr* member for the interface specified with the *ifr_name* member. If the interface does not have broadcast capability, the EADDRNOTAVAIL error is reported.

SIOCGLIFNETMASK
: The subnet mask is returned in the *if_addr* element for the interface specified with the *ifr_name* element. For IPv6 interfaces the EADDRNOTAVAIL error is reported.

SIOCGIFFLAGS
: The interface flags are returned in the *ifr_flags* member for the interface specified with the *ifr_name* member. The possible flags are IFF_UP, IFF_LOOPBACK and IFF_BROADCAST.

Return value

0:
: If successful

-1:
: If errors occur. *errno* is set to indicate the error.

Errors

EFAULT
: *request* requests data transfer to or from the buffer to which *arg* points. However, part of the buffer lies outside the address range assigned to the process.

EINVAL
: *request* or *arg* are not valid.
The interface name specified (in *lifr_name* or *ifr_name*) is not valid.
The address family specified (in *lifn_family* or *lifc_family*) is not valid.
The length of the output area (*lifc_len* or *ifc_len*) specified in SIOCGLIFCONF or SIOCGIFCONF is not large enough.

EIO
: A physical input/output error occurred.

EOPNOTSUPP
: *request* is not supported.

EADDRNOTAVAIL
: *request* is not possible for this interface.

Example

Get all interface names and addresses with SIOCGLIFCONF.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/sockio.h>
```

```
#include <net/if.h>
#include <netdb.h>
#define  err_exit(a) {perror((a)); exit(1);}

int main(int argc, char **argv)
{
  int                    soc, cnt, af;
  struct lifconf         lifc;
  struct lifreq          *plifr;
  struct lifnum          lifn;
  char                   addr_str[INET6_ADDRSTRLEN + 1];
  char                   af_str[64];

  soc = socket(AF_INET, SOCK_DGRAM, 0);

  lifn.lifn_family = AF_UNSPEC;
  lifn.lifn_flags  = 0;
  if (ioctl(soc, SIOCGLIFNUM, &lifn) < 0) {
    err_exit("ioctl(SIOCGLIFNUM)");
  }
  lifc.lifc_len =        lifn.lifn_count * sizeof(struct lifreq);
  lifc.lifc_buf = malloc(lifn.lifn_count * sizeof(struct lifreq));
  if (lifc.lifc_buf == NULL) {
    err_exit("malloc");
  }
  lifc.lifc_family = AF_UNSPEC;
  lifc.lifc_flags  = 0;
  if (ioctl(soc, SIOCGLIFCONF, &lifc) < 0) {
    err_exit("ioctl(SIOCGLIFCONF)");
  }
  plifr = lifc.lifc_req;
  cnt = lifc.lifc_len / sizeof (struct lifreq);
  for (; cnt>0; cnt--, plifr++) {
    af = plifr->lifr_addr.ss_family;
    switch (af) {
    case AF_INET:
      sprintf(af_str, "AF_INET");
      inet_ntop(af, &((struct sockaddr_in *)&plifr->lifr_addr)->sin_addr,
                addr_str, INET6_ADDRSTRLEN);
      break;
    case AF_INET6:
      sprintf(af_str, "AF_INET6");
      inet_ntop(af, &((struct sockaddr_in6 *)&plifr->lifr_addr)->sin6_addr,
                addr_str, INET6_ADDRSTRLEN);
      break;
    default:
      sprintf(af_str, "af=%d", af);
      strcpy(addr_str, "???");
```

```
    }
    printf ("%-15s  %-8s  %s\n", plifr->lifr_name, af_str, addr_str);
  }
  free(lifc.lifc_buf);
  return 0;
}
```

The data structures *lifconf*, *lifreq* and *lifnum* are defined 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
};

struct  lifreq {
#define LIFNAMSIZ 32
    char                        lifr_name[LIFNAMSIZ];
    union {
        int                     lifru_addrlen;
        ...
    } lifr_lifru1;
#define lifr_addrlen    lifr_lifru1.lifru_addrlen
...
    unsigned int                lifr_movetoindex;
    union {
        struct sockaddr_storage  lifru_addr;
        ...
    } lifr_lifru;
#define  lifr_addr       lifr_lifru.lifru_addr
....
};

struct lifnum {
    sa_family_t     lifn_family;
    int             lifn_flags;
    int             lifn_count;
};
```

poll() - multiplex input/output

```
#include <poll.h>

int poll(struct pollfd fds[], unsigned long nfds, int timeout);
```

Description

The *poll()* function provides the user with a mechanism for multiplexing the input/output via a set of file descriptors which refer to open files.
poll() identifies the descriptors on which

- the user can receive messages,
- the user can send messages or
- specific events have occurred.

The *fds* parameter defines the descriptors to be tested and the events which are of interest for each descriptor. *fds* is a pointer to a vector with one member for each open descriptor.

The *pollfd* structure is declared as follows:

```
struct pollfd {
    int fd;               /* file ID */
    short events;         /* requested events */
    short revents;        /* reported events */
};
```

The file descriptor *fd* designates a socket. The members *events* (events to be queried for the socket) and *revents* (events returned for the socket) are bit masks which can be built up from ORed combinations of the event indicators described below.

POLLIN
: Data cannot be read non-blocking if a connection request was received.

POLLOUT
: Data cannot be written non-blocking.

POLLRDNORM
: as POLLIN

POLLWRNORM
: as POLLOUT

POLLERR
: An error was reported for the socket. The option is only valid in the *revents* bit mask and is not used in an *events* field.

POLLHUP

A hangup has occurred in the connection. POLLHUP and POLLOUT are mutually exclusive. If a hangup has occurred, a socket can never be written to. However, POLLHUP and POLLIN are not mutually exclusive. This option is only valid in the *revents* bit mask and is not used in an *events* field.

POLLNVAL

The specified *fds->fd* value does not belong to an open file. This option is only valid in *fds->revents* and is not used in *fds->events*.

For each member of the vector to which *fds* points, *poll()* tests the specified file descriptor for the event(s) specified in *fds->events*. The number of file descriptors to be tested is specified by *fds*.

If *fds->fd* is less than zero, *fds->events* is ignored and *fds->revents* is reset to zero in this entry when *poll()* returns.

The results of the *poll()* call are stored in *fds->revents*. Bits are set in the *fds->revents* bit mask to indicate which of the requested events are true. If no events are true, none of the bits in *fds->revents* are set when the *poll()* call returns. The event indicators POLLHUP, POLLERR and POLLNVAL are always set in *fds->revents* if the conditions indicated by them are true. This is also the case if these options were not in *fds ->events*.

If none of the defined events occurs with any of the selected file descriptors, *poll()* waits at least *timeout* milliseconds for the occurrence of an event with one of the selected file descriptors. On a host which cannot measure to millisecond accuracy, *timeout* is rounded up to the next permissible value available in the system.

poll() returns immediately if the value of *timeout* is 0. If the value of *timeout* is INFTIM (or -1), *poll()* blocks until a requested event occurs or the call is interrupted. *poll()* is not affected by the O_NDELAY and O_NONBLOCK switches.

Return value

0:

Indicates that the time for the call has expired and no file descriptors were selected.

>0:

A positive number indicates the total number of currently selected file descriptors (i.e. file descriptors for which *fds->revents* is not equal to zero).

-1:

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

Errors

EAGAIN
The assignment of the internal data structures failed, but the request should be retried.

EFAULT
A parameter refers to an address outside the assigned address range.

EINTR
A signal was trapped during the *poll()* call.

EINVAL
The *nfds* parameter is less than zero or greater than OPEN_MAX.

See also

accept(), listen();
read(), select(), write() [1]

read(), readv() - receive a message from a socket

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

#include <sys/uio.h>


ssize_t read(int s, char *buf, int len);

ssize_t readv(int s, const struct iovec *iov, int iovcnt);
```

Description

The *read()* and *readv()* functions receive messages from a socket. *read()* and *readv()* can only be used with a socket over which a connection is set up. The length of the message is returned.

The *s* parameter designates the socket from which the message is received.

For *read(),* the *buf* parameter points to the first byte of the receive buffer. The *len* parameter specifies the length (in bytes) of the receive buffer, and thus the maximum message length.

For *readv(),* the received data is placed in a vector with the members *iov[0],... ,iov[iovcnt-1]*. The vector members are objects of the type *struct iovec*. The address of the vector is passed in the *iov* parameter. Each vector member contains the address and length of a storage area into which *readv()* reads the data received from socket *s*. *readv()* fills one area after the other with data. *readv()* does not move on to the next area until the current area is completely filled with data.

The *struct iovec* structure is declared as follows:

```
struct iovec {
caddr_t iov_base;  /* buffer for data */
size_t iov_len;    /* length of buffer */
};
```

iovcnt indicates the number of vector members.
If no messages are available on the socket, the receive call waits for an incoming message, unless the socket is non-blocking. See the section “ioctl() - control sockets” on page 132 for more information. In this case, *read()* and *readv()* return the value -1, and the *errno* variable is set to the value EWOULDBLOCK.

The *poll()* or *select()* function can be used to determine when further data arrives.

If the process which calls *read()* or *readv()* receives a signal before any data is available, the call is repeated. The call is not repeated if the calling process uses *sigaction()* to set the signal to interrupt this call (see also the manual “C Library Functions (BS2000/OSD) for POSIX Applications”).

Return value

>0:
: If successful

-1:
: If errors occur. *errno* is set to indicate the error.

Errors

EBADF
: The *s* parameter is not a valid descriptor.

ECONNRESET
: The connection to the partner was interrupted (only with type SOCK_STREAM sockets).

EFAULT
: The data is to be received in a non-existent or protected part of the process address range.

EINTR
: The calling process received a signal before any data could be received and the signal to interrupt the function call is set.

EIO
: User data has been lost.

ENETDOWN
: The connection to the network is down.

ENOTCONN
: No connection exists for the socket.

ENOTSOCK
: Descriptor *s* references a file and not a socket.

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

See also

connect(), getsockopt(), recv(), send(), socket();
fcntl(), ioctl(), read(), select(), write() [1]

select() - multiplex input/output

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

int select(int width, 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 descriptor sets are stored as bit fields in Integer strings. The size of the bit fields (and descriptors) is defined by the FD_SETSIZE constant. FD_SETSIZE is defined in <sys/select.h> with a value which is by default at least as large as the maximum number of descriptors supported by the system.

The *width* parameter specifies the number of bits to be tested in each bit mask. *select()* tests bits 0 to *width*-1 in the separate bit masks. *width* normally has the value supplied by the *ulimit()* function as the maximum number of socket descriptors. The *ulimit()* function is described in the manual “C Library Functions (BS2000/OSD) for POSIX Applications”.
select() replaces the descriptor sets passed with the call with corresponding subsets. These subsets each contain all descriptors that are ready for the operation concerned.

You can use the following macros to manipulate bit masks or descriptor sets:

FD_ZERO(&fdset)
Initializes the descriptor set *fdset* as an empty set.

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*.

The behavior of these macros is undefined if the descriptor value is <0 or ≥ FD_SETSIZE.

The *timeout* parameter defines the maximum time that the *select()* function has for complete selection of the ready descriptors. If *timeout* is the null pointer, *select()* blocks for an undefined time.
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.

-1:
If errors occur. *errno* is set to indicate the error. The descriptor sets are then not changed.

Errors

EBADF
One of the descriptor sets specified an invalid descriptor.

EFAULT
One of the pointers that were passed points to a non-existent area in the process address range.

EINTR
A signal was received before one of the selected events arrived or before the time limit expired.

EINVAL
A component of the specified time limit is outside the valid range.
The valid range is defined as follows:

– $0 \leq t\_sec \leq 10^8$
– $0 \leq t\_usec < 10^6$

Note

In rare circumstances, *select()* can indicate that a descriptor is ready for writing while 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 *fcntl()* call.

See also

accept(), connect(), listen(), recv(), send();
fcntl(), read(), ulimit(), write() [1]

write(), writev() - send a message from socket to socket

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

ssize_t write(int s, char *buf, int len);
ssize_t writev(int s, const struct iovec *iov, int iovcnt);
```

Description

The *write()* and *writev()* functions send messages from one socket to another. *write()* and *writev()* can only be used with a socket over which a connection is set up.

The *s* parameter designates the socket over which the message is sent.

For *write()*, the *buf* parameter points to the first byte of the send buffer, and *len* specifies the length of the message in the send buffer in bytes.

For *writev()*, the data to be sent is supplied in the vector with the members *iov[0], ... ,iov[iovcnt-1]*. The vector members are objects of the type *struct iovec*. The address of the vector is passed in the *iov* parameter. Each vector member contains the address and length of a storage area from which *writev()* reads the data to be sent.

The *struct iovec* structure is declared as follows:

```
struct iovec {
    caddr_t iov_base;      /* buffer for data */
    size_t iov_len;        /* length of buffer */
};
```

iovcnt indicates the number of vector members.

If the message is too long to be transported completely by the underlying protocol level, error EMSGSIZE is returned and the message is not transferred.

If the process which calls *write()* and *writev()* receives a signal before any send data is buffered, the call is repeated.
The call is not repeated if the calling process uses *sigaction()* to set the signal to interrupt this call (see also the description of SA_RESTART at *sigaction()* in the manual “C Library Functions (BS2000/OSD) for POSIX Applications“).

Return value

0:
If successful (number of bytes actually sent)

-1:
If errors occur. *errno* is set to indicate the error. The descriptor sets are then not changed.

Errors

EBADF
The *s* parameter is not a valid descriptor.

ECONNRESET
The connection to the partner was interrupted (only with type SOCK_STREAM sockets).

EFAULT
The data is to be sent to a non-existent or protected part of the process address range.

EINTR
The calling process received a signal before any data could be buffered for sending and the signal for interrupting the function call is set.

EINVAL
A parameter specifies an illegal value.

EMSGSIZE
The message is too long to be sent in one piece.

ENETDOWN
The connection to the network is down.

ENOBUFS
The system could not provide an internal buffer. The operation can succeed if memory is freed.
The output queue for a network interface is full. This generally leads to the interface stopping sending, but can also be due to a temporary jam.

ENOTCONN
No connection exists for the socket.

ENOTSOCK
Descriptor *s* references a file and not a socket.

EPIPE
The socket is not activated for writing or the socket is connection-oriented and the partner has shut the connection down.
If the socket is of type SOCK_STREAM, the SIGPIPE signal is generated for the calling process.

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

See also

connect(), getsockopt(), recv(), socket();
fcntl(), select(), write() [1]

7 XTI(POSIX) basics

X/Open Transport Interface (XTI) is the standard defined by X/Open for a number of programming interfaces which allow the application to access network levels, similarly to the socket interface.

XTI offers two types of services:

- connection-oriented service
- connectionless service

XTI appears to the user as a finite, event-controlled state machine.
This means:

- for a transport endpoint, there is a finite number of defined states
- each of these states can only be reached via specific events
- in each state, only specific functions can be executed.

7.1 Connection-oriented service

The connection-oriented service transports data over a one-time “virtual connection”. This service is tailored for applications which require a secure, data flow-oriented connection.

7.1.1 Connection-oriented service phases

The connection-oriented service comprises three phases:

– local management
– connection setup
– data transfer
– connection shutdown

Local management

The local management defines functions between the transport user, the transport provider and other instances which control connection setup.

Examples of local functions:

– The user has to set up a communications channel to the transport provider. Each channel between the user and transport provider is called a *transport endpoint*. The user selects a special transport provider and sets up a transport endpoint with the *t_open()* function.

– Each user can manage one or more transport endpoints, which he has to identify to the transport provider. For this, the user assigns each transport endpoint a transport address, which is unique throughout the network, with the *t_bind()* function, i.e. he *binds* a transport address to the transport endpoint. The structure of the transport address is defined by the transport provider concerned.

In addition to *t_open()* and *t_bind(),* further functions exist for supporting the local transport interface management. These are summarized in table 2.

Function	Description
t_alloc()	Reserves memory for the transport interface
t_bind()	Binds an address to a transport endpoint.
t_close()	Closes a transport endpoint.
t_error()	Prints an error message from the transport provider.
t_free()	Releases the memory area reserved with *t_alloc()*.
t_getinfo()	Returns the parameter set of the current transport provider.
t_getstate()	Returns the state of the transport endpoint.
t_look()	Returns the current events of the transport endpoint.
t_open()	Sets up a transport endpoint that is to be bound to a specific transport provider.
t_optmgmt()	Negotiates protocol-specific options with the transport provider.
t_sync()	Synchronizes the transport endpoint with the transport provider.
t_unbind()	Unbinds an address from a transport endpoint.

Table 2: Functions for local management of the transport interface

Connection setup

In this phase, a communication connection is set up between two users.
The connection setup can be illustrated using the example of two transport users who have a client/server relationship with each other: one transport user (server) makes a number of services available to a group of users (clients) and then waits for requests from these clients. Each client can request a service after setting up a connection to the server.

The client requests a connection with the *t_connect()* function. One parameter of *t_connect()*, the address, identifies the server that the client wishes to reach. The server has to use the *t_listen()* function to be informed of all incoming connection requests. The server accepts a request to set up a connection with *t_accept()*. The transport connection is then set up.

Table 3 shows the connection setup functions.

Function	Description
t_accept()	Accepts a request to set up a connection.
t_connect()	Requests a connection with a particular user at a specified address.
t_listen()	Waits for a request to set up a connection from another user.
t_rcvconnect()	Confirms a connection setup request if *t_connect()* was called in asynchronous mode.

Table 3: Functions for connection setup

Data transfer

Data transfer allows two users to exchange data in both directions over an established connection. The *t_snd()* and *t_rcv()* functions send and receive data respectively over this connection. It is ensured that the data sent arrives at the receiver in the same order as sent.

Table 4 shows the functions for connection-oriented data transfer.

Function	Description
t_rcv()	Receives data.
t_snd()	Sends data.

Table 4: Functions for connection-oriented data transfer

Connection shutdown

The user sends the transport provider a request to shut an established connection down. There are two different types of connection shutdown:

- Abortive connection release:
  The abortive connection release directs the transport provider to terminate the connection immediately, whereby all previously sent data that has not reached the receiver may be lost. The transport user can initiate such a connection release with the *t_snddis()* function. The communication partner affected by this shutdown can query the cause of the shutdown with the *t_rcvdis()* function. The *t_rcvdis()* function handles incoming requests after a connection has been aborted.

- Orderly connection shutdown:
  In addition to the abortive connection release, some transport providers also enable a connection to be shut down in an orderly manner, where no data is ever lost. The *t_sndrel()* and *t_rcvrel()* functions implement an orderly connection shutdown.

  The orderly connection shutdown between two users, user1 and user2, always progresses in the following steps:

  1. User1, who is the first who wishes to shut the connection down, uses the *t_sndrel()* function to send user2 a request to shut the connection down. *t_sndrel()* informs user2 that user1 will send no further data.

  2. After receiving such a message with the *t_rcvrel()* function, user2 can still send data to user1.

  3. After transferring all data, user2 must also call *t_sndrel()*. This informs user1 that user2 is now ready to shut the connection down.

  4. The connection is shut down as soon as user1 receives the message from user2 with *t_rcvrel()*.

Table 5 shows the functions for shutting a connection down.

Function	Description
t_rcvdis()	Informs about an abortive connection release.
t_rcvrel()	Indicates that the communications partner wants an orderly connection shutdown.
t_snddis()	Requests an abortive connection release or refuses a connection request.
t_sndrel()	Requests an orderly connection shutdown.

Table 5: Functions for connection shutdown

Interaction between the connection-oriented service functions

Figure 4 illustrates the interaction between the XTI functions which implement the separate phases of the connection-oriented service. The separate XTI functions are described in detail in chapter “XTI(POSIX) library functions” on page 223.

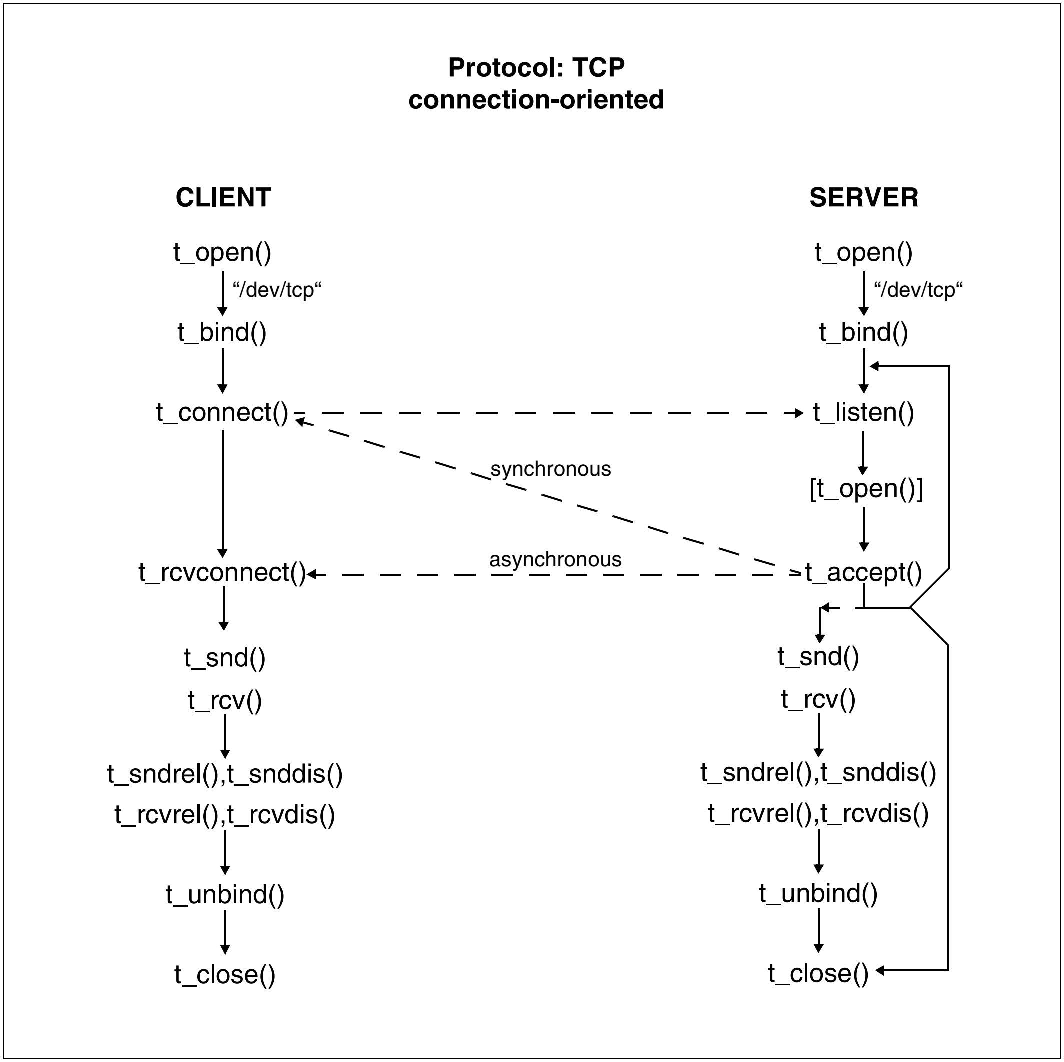

Figure 4: Interaction between the connection-oriented service functions

7.1.2 Connection-oriented client/server model

This section provides a detailed description of the separate phases of the connection-oriented service using an example program for the following simple client/server application:

1. The client and server carry out their local management tasks.
2. A connection is set up between the client and server.
3. The server transfers a file to the client. The client receives the file from the server and outputs it to its standard output.
4. The client and server shut the connection down.

The example program is described in separate program sections, where two program sections explain each phase of the connection-oriented service. One program section takes on the role of the client and the other the role of the server.

The program code used in the examples in this section is shown completely and coherently in section “Client in the connection-oriented service” on page 202 and section “Server in the connection-oriented service” on page 204.

Local management using the example client/server model

Before the client and server can set up a communications connection, they must first each set up a local channel to the transport provider with *t_open()*. After this, they must each use *t_bind()* to make a local address known under which each can be reached via its assigned transport endpoint.
The user gets the various services offered by the transport interface with the *t_open()* call.

The services are built up as follows:

Address	Maximum size of an address
Options	Maximum number of bytes for protocol-specific options which the user can exchange with the transport provider
tsdu	Maximum message size which can be transferred in the connection-oriented or connectionless services
etsdu	Maximum number of bytes for expedited data which can be sent over a connection
Connection setup (connect)	Maximum number of bytes for user data which can be exchanged during connection setup
Connection shutdown (discon)	Maximum number of bytes of user data which can be transferred during connection shutdown
Service type	Type of the service supported by the transport provider

Three service types are defined:

T_COTS	The transport provider supports the connection-oriented service but does not allow an orderly connection shutdown. The connection can only be aborted.
T_COTS_ORD	The transport provider supports the connection-oriented service and provides the option of an orderly connection shutdown (standard for XTI(POSIX) in the connection-oriented service).
T_CLTS	The transport provider supports the connectionless service.

The user receives the preset features of the transport endpoint with *t_open()*. If these are dynamic features, they may subsequently change. The user can obtain information on the current features of the transport endpoint with *t_getinfo()*.

Once a user has set up a transport endpoint, he must pass the transport provider the address under which he can be reached via this endpoint. As described above, the user passes the transport endpoint address to the transport provider with *t_bind()*. With server stations, *t_bind()* also ensures that incoming connection requests can be processed by the transport provider and forwarded to the transport endpoint.

One additional function is available while the transport endpoint is being set up: the user can change features with *t_optmgmt()*. Each transport protocol is expected to provide its own set of changeable features. These can, for example, be parameters that affect the service quality. Because of the protocol-specific nature of these parameters, only applications for a special protocol environment will use this option.

The local management tasks are shown below using a client and a server as examples. The two examples contain the definitions and calls.

Local management by the client

```
#include <xti.h>
#include <stdio.h>
#include <fcntl.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define SRV_ADDR 0x7F000001
#define SRV_PORT 8888

main()
{
   int fd;
   int nbytes;
   int flags = 0;
   char buf[1024];
   struct t_call *sndcall;
   struct sockaddr_in *sin;

   if ((fd = t_open("/dev/tcp", O_RDWR, NULL)) < 0)
   {
      t_error("t_open() failed");
      exit(1);
   }
   if (t_bind(fd, NULL, NULL) < 0)
   {
      t_error("t_bind() unsuccessful");
      exit(2);
   }
```

The first parameter of *t_open()* is the path name of the device which provides the requested transport service. In this example, */dev/tcp* is a device file and provides a connection-oriented transport protocol. This transport protocol is opened for read/write accesses by the second parameter. The user can employ the third parameter to get information on the available features. This information is required to create programs that are independent of protocols. To keep the example simple, this information is not accessed.

The client and server assume that the transport provider has the following features:

– Support for service type T_COTS_ORD, which is used in the example for the orderly connection shutdown.

– User data cannot be exchanged during connection setup or shutdown.

– No protocol-specific features are provided.

Since these features are not needed by the user, NULL is passed as the third parameter in the *t_open()* call. A different device file must be opened if the user requires a service type other than T_COTS_ORD. An example for T_CLTS is shown in section “Connectionless service using an example transaction system” on page 175.

t_open() returns an integer value, which is required in all further transport provider calls to identify the transport endpoint set up with *t_open()*. This integer value is a file descriptor.

After the transport endpoint has been set up, the user calls *t_bind()* to assign the transport endpoint an address. The first parameter of *t_bind()* identifies the transport endpoint and the second parameter describes the address which is to be bound to the transport endpoint. When *t_bind()* returns, the third parameter contains the actually bound address.

In contrast to the address of a server transport endpoint, which is needed by all clients to access the server, the address of a client does not have to be generally known. As no other process will try and access the address of a client, the client does not normally bother with its own address. This is shown in the above example in the *t_bind()* call where NULL is passed as the second and third parameters. If the second parameter is NULL, the transport provider assigns an address. The third NULL parameter means that the client is “not interested” in the address assigned by the transport provider.

If either *t_open()* or *t_bind()* is unsuccessful, *t_error()* is called to output an appropriate error message to *stderr*. If any of the transport provider functions should fail, the global integer variable *t_errno* is set to a corresponding value that indicates the error more closely. A number of such error values, and the *t_errno* variable itself, are defined in <xti.h> for the transport provider. *t_error()* outputs an error message according to the value of *t_errno*. This function works in the same way as the *perror()* function which outputs an error message according to the value of *errno*. If the error in the transport provider is a system error, *t_errno* is set to the value TSYSERR and *errno* is set to the appropriate system error value.

Local management by the server

The server in this example has to proceed in a similar manner before communications can be started. The server has to set up a transport endpoint which waits continuously for connection requests.

The definitions and calls required are as follows:

```
#include <xti.h>
#include <stropts.h>
#include <fcntl.h>
#include <stdio.h>
#include <signal.h>
#include <netinet.in.h>
#include <sys/socket.h>

#define FILENAME "/etc/services"
#define DISCONNECT -1
#define SRV_ADDR 0x7F000001
#define SRV_PORT 8888

int conn_fd; /* For the connection file descriptor */

main()
{

   int listen_fd;              /*   File descriptor for
                                *   connection request
                                */
   struct t_bind *bind;
   struct t_call *call;
   struct sockaddr_in *sin;


   if ((listen_fd = t_open("/dev/tcp", O_RDWR, NULL)) < 0)
   {
      t_error("t_open() call for listen_fd failed.");
      exit(1);
   }


   if ((bind = (struct t_bind *)t_alloc(listen_fd, T_BIND, T_ALL)) == NULL)
   {
      t_error("t_alloc() for t_bind structure failed.");
      exit(2);
   }
```

```
bind->qlen = 1;
bind->addr.len=sizeof(struct sockaddr_in);
sin=(struct sockaddr_in *)bind->addr.buf;
sin->sin_family=AF_INET;
sin->sin_port=htons(SRV_PORT);
sin->sin_addr.s_addr=htonl(SRV_ADDR);

if (t_bind(listen_fd, bind, bind) < 0)
{
   t_error("t_bind() for listen_fd failed.");
   exit(3);
}
```

Analogous to the client, the server also calls *t_open()* to set up a connection to the desired transport provider, i.e. the server sets up a transport endpoint (*listen_fd*). The server will use this transport endpoint *listen_fd* later when it calls the *t_listen()* function to wait for connection requests.

Before the server can use the *t_bind()* function to bind an address to the transport endpoint *listen_fd*, the server has to provide this address. The address is passed with the second parameter (*bind*) when *t_bind()* is called.

The *bind* parameter is a pointer to an object of data type *struct t_bind*. All structures and constants of the transport provider are declared/defined in <xti.h>.

The *t_bind* structure is declared in <xti.h> as follows:

```
struct t_bind {
   struct netbuf addr;
   unsigned qlen;
};
```

bind->qlen defines the maximum number of allowed connection requests.
If the value of *bind->qlen* is greater than 0, incoming connection requests can be processed with this transport endpoint. The server then puts incoming connection requests for the address provided in *bind->addr* into a queue. *bind->qlen* also defines the maximum number of requests that the server can process simultaneously. The server must reply to all requests by either accepting or refusing them. A connection request is pending if the server has not replied to it.
A server will often completely process one connection request and then the next. In this case, *qlen* should be set to the value 1. If a server wants to process several requests simultaneously, *bind->qlen* specifies the maximum number of requests which can be processed simultaneously.
Since the server in the example processes one connection request after the other, *bind->qlen* must be assigned the value 1. An example of a server that processes several requests simultaneously is shown in section “Managing multiple connections simultaneously and event- controlled operation” on page 193.

addr has the data type *struct netbuf* and describes the address to be bound.
The *netbuf* structure is declared in <xti.h> as follows:

```
struct netbuf {
   unsigned int maxlen;
   unsigned int len;
   char *buf;
};
```

buf is a pointer to a data buffer, *len* specifies the number of bytes in the buffer and *maxlen* specifies the maximum number of bytes that can be written into the buffer. The last entry is only required if data is transported from the transport provider to the user.

Calling *t_alloc()* reserves memory dynamically for a *t_bind* object. The first parameter of *t_alloc()* names the file descriptor which identifies the transport endpoint. The second parameter specifies the transport provider structure to be created, i.e. *t_bind* in this case. The third parameter specifies which components of this structure are to be created. T_ALL means that memory is to be reserved for all the components of the structure. This creates the *addr* buffer in the above example. The size of the buffer is determined by the transport provider, who defines a maximum address length. This length is in the *maxlen* component of the *netbuf* structure.
Using *t_alloc()* ensures compatibility with future versions of the transport provider.

The data is interpreted as an address with objects of type *struct t_bind*. It is generally assumed that the structure of an address differs from protocol to protocol. The structure of *netbuf* is created such that all protocols can be supported.

Finally, the address information is assigned to the new *t_bind* object. In the example, the address itself is structured according to the Internet communications domain address structure (see *struct sockaddr_in* on page 14).

The server now binds the address created above to the transport endpoint *listen_fd* with the *t_bind()* function. After the *t_bind()* call has been successfully executed, the server can be accessed by any client via this address. The transport provider puts incoming connection requests into a queue and this initiates the next phase of the connection setup protocol, the actual connection setup.

Connection setup using the example client/server model

The connection setup illustrates the difference between the client and server. The transport provider makes different, special functions available to each of them. The client calls *t_connect()* to request a connection while the server uses *t_listen()* to wait for connection requests. The server can either accept a connection with the *t_accept()* function or refuse it with *t_snddis()*. The client is informed of the decision of the transport provider when the *t_connect()* function terminates.

Connection request by the client

To continue with the client/server example, the following steps are required for connection setup from the viewpoint of the client:

```
if ((sndcall = (struct t_call *)t_alloc(fd, T_CALL, T_ADDR)) == NULL) {
   t_error("t_alloc() failed");
   exit(3);
}
sndcall->addr.len=sizeof(struct sockaddr_in);
sin=(struct sockaddr_in *)sndcall->addr.buf;
sin->sin_family=AF_INET;
sin->sin_port=htons(SRV_PORT);
sin->sin_addr.s_addr=htonl(SRV_ADDR);

if (t_connect(fd, sndcall, NULL) < 0) {
   t_error("t_connect() for fd failed");
   exit(4);
}
```

Before the client can send a connection request to the server with *t_connect()*, the client must specify the address of the server. This address is then passed as the second parameter (*sndcall*) with the *t_connect()* call.

The *sndcall* parameter is a pointer to an object of data type *struct t_call*.

The *t_call* structure is declared in <xti.h> as follows:

```
struct t_call {
   struct netbuf addr;
   struct netbuf opt;
   struct netbuf udata;
   int sequence;
};
```

t_alloc() is used in the example to set up a *t_call* object dynamically. No features or user data are specified in the above example. Only the server address is used. T_ADDR is selected as the third parameter of *t_alloc()* to set up an appropriate buffer for the address information.

After *t_alloc()* has been successfully executed, the server deposits the server address and its length into the memory area reserved by *t_alloc()*. The server address is thereby structured according to the address structure of the Internet communications domain (see *struct sockaddr_in* on page 14).

The *t_connect()* call sends a connection request to the server. The first parameter of the call is the transport endpoint over which the connection is to be set up. The address of the desired server is passed with the second parameter (*sndcall*). The third parameter is also a pointer to an object of type *struct t_call*. This *t_connect()* parameter is used to get information on the established connection. Since this information is not needed here, NULL is passed as the third parameter in the example. If *t_connect()* is successful, the connection is set up. If the server refuses the connection request, *t_errno* is set to the value TLOOK.

The TLOOK error has a special significance for the transport interface: TLOOK informs the user if an interface function was interrupted by an unexpected asynchronous event on the specified transport endpoint. TLOOK therefore does not indicate an interface error, but only that the called function is not executed because of the pending event. The defined transport interface events are described on page 182.

The user can determine which event has occurred when a TLOOK error is reported, with the *t_look()* function. If the connection request is refused in the above example, the client receives a message about the aborted connection. The program is terminated in this case.

Connection acceptance by the server

When the client requests a connection with *t_connect()*, a corresponding event is set at the transport endpoint of the server. The steps required for handling this event are shown below. For each client, the server accepts the request and creates a new process to manage the connection.

```
if ((call = (struct t_call *)t_alloc(listen_fd, T_CALL, T_ADDR)) == NULL){
   t_error("t_alloc() for t_call structure failed");
   exit(5);
}

while (1) {
   if (t_listen(listen_fd, call) < 0) {
      t_error("t_listen for listen_fd failed");
      exit(6);
   }

   if ((conn_fd = accept_call(listen_fd, call)) != DISCONNECT)
      run_service(listen_fd);
   }
}
```

The server uses *t_alloc()* to set up an object of type *struct t_call* that is required by *t_listen()*. The third parameter of *t_alloc()*, T_ADDR, causes the buffer for the address of the client to be created.
The value of *maxlen* in a *netbuf* object specifies the actual length of the created buffer.

The server runs in an endless loop and processes one incoming connection request in each loop run. The server thereby proceeds as follows:

1. The server calls the *t_listen()* function to wait for connection requests that arrive at the transport endpoint *listen_fd*. The transport address of the sender of a connection request is stored by *t_listen()* in the *t_call* object to which the pointer variable *call* points. If no connection requests are pending, the *t_listen()* function blocks the process until a connection request arrives.
2. When a connection request arrives, the server calls the user-defined *accept_call()* function to confirm the connection. *accept_call()* accepts the connection request on a new transport endpoint and returns the relevant file descriptor as the result. This file descriptor is stored in the global *conn_fd* variable. Since the connection is set up on a new transport endpoint, the server can wait for further requests on the old transport endpoint. The *accept_call()* function is described in detail below.
3. If the connection acceptance was successful, the *run_service()* function creates a new process to manage the connection. The user-defined *run_service()* function is described in detail on page 168.

The transport interface supports an asynchronous mode and this is described in chapter “Advanced XTI(POSIX) concepts” on page 191.

The *accept_call()* function, which the server calls to accept a connection request, is defined as follows:

```
accept_call(listen_fd, call)
int listen_fd;
struct t_call *call;
{
   int resfd;
   struct t_call *refuse_call;

   if ((resfd = t_open("/dev/tcp", O_RDWR, NULL)) < 0) {
      t_error(„t_open() call for accept failed”);
      exit(7);
   }
   while (t_accept(listen_fd, resfd, call) < 0) {
      if (t_errno == TLOOK) {
         if (t_look(listen_fd) == T_DISCONNECT) { /* Connection abort */
               if (t_rcvdis(listen_fd, NULL) < 0) {
                  t_error("t_rcvdis() failed for listen_fd");
                  exit(9);
               }
```

```
            if (t_close(resfd) < 0) {
               t_error("t_close failed for responding fd");
               exit(10);
            }
            /* Terminate call and wait for further calls */
            return(DISCONNECT);
         } else { /* new T_LISTEN; delete event */
            if ((refuse_call =
               (struct t_call *)t_alloc(listen_fd,T_CALL,0)) == NULL) {
               t_error("t_alloc() for refuse_call failed");
               exit(11);
            }

            if (t_listen(listen_fd, refuse_call) < 0) {
               t_error("t_listen() for refuse_call failed");
               exit(12);
            }



            if (t_snddis(listen_fd, refuse_call) < 0) {
               t_error("t_snddis() for refuse_call failed");
               exit(13);
            }

            if (t_free((char *)refuse_call, T_CALL) < 0) {
               t_error("t_free() for refuse_call failed");
               exit(14);
            }
         }
      } else {
         t_error("t_accept() failed");
         exit(15);
      }
   }
   return(resfd);
}
```

The *accept_call()* call needs two parameters:

– *listen_fd* specifies the transport endpoint on which the connection request arrived.

– *call* is a pointer to an object of data type *struct t_call* that contains all the information for these requests.

The *t_call()* function first creates an additional transport endpoint. This new transport endpoint *resfd* is used to accept the connection request.

The *t_accept()* function accepts the connection request. The first parameter of the *t_accept()* function specifies the transport endpoint on which the request was received. The second parameter specifies the transport endpoint on which the request is to be confirmed.
A request can be confirmed on the same transport endpoint on which it was received.

In this case, other clients cannot make any requests for the duration of this connection. The third parameter of *t_accept()* points to the *t_call* object of the currently processed connection request. This object should contain the address of the calling client and the sequential number of the *t_listen()* call. The value of *call->sequence* is significant if the server manages several connections. You will find an appropriate example in section "Event-controlled server" on page 210.

To keep this example simple, the server terminates the program if the *t_open()* call fails. *exit(2)* closes the transport endpoint assigned to *listen_fd*. The transport provider thereby sends the client a message to the effect that the connection was aborted and the connection request was unsuccessful. The *t_connect()* call fails and *t_errno* is set to TLOOK.

t_accept() execution can fail if an asynchronous event occurs on the receiving transport endpoint before the connection is accepted. *t_errno* is then set to TLOOK. Table 9 on page 183 shows that precisely one of the two following events can arrive:

- An abort message has arrived for the previously reported connection request, i.e. the client who sent the connection request wants to abort the connection.

  When an abort request arrives, the server must immediately use a *t_rcvdis()* call to analyze the reason for the request. The *t_rcvdis()* function has a parameter which is a pointer to an object of data type *t_discon* (see page 270). The *t_discon* object is required to store the abort condition. The reason for the abort is not queried in this example and the parameter is therefore set to NULL. After the abort condition is received, *accept_call()* closes the transport endpoint and returns a DISCONNECT as its result. This informs the server that the connection was closed by the client.

- A new connection request arrived during execution of *t_accept()*.

  In this example, the server refuses this connection request in order to be able to accept the currently processed connection request without interruption. The server thereby proceeds as follows:

  1. The server creates a new object of type *struct t_call* with *t_alloc()* .
  2. The server then accepts the new connection request with *t_listen()* which returns a unique ID for the new connection request in the *refuse_call->sequence* field.
  3. The server refuses the new connection request with *t_snddis()*.
  4. The server repeats the *t_accept()* call after releasing the *t_call* object referenced by *refuse_call*.

The transport connection has been set up with the newly created transport endpoint. This allows the receive endpoint to handle new connection requests.

Data transfer using the example client/server model

Once the connection has been set up, the client and server can start exchanging data. They use the *t_snd()* and *t_rcv()* functions for this. From this point on, the transport provider does not distinguish between the client and server. Each user can send and receive data or close the connection. The transport provider offers secured data transfer and maintains the order of sending over an established connection.
In the example, the server sends one file to the client over the established connection.

Data sending by the server

The server organizes the data transfer by creating a new process which sends the data to the client. The parent process waits for further connection requests while the child process transfers the data.

The *run_service()* function is called to create this child process. The following extract from the definition of *run_service()* illustrates this procedure:

```
run_service(listen_fd)
int listen_fd;
{
   int nbytes;
   FILE *logfp;  /* Pointer to the protocol file */
   char buf[1024];

   switch (fork()) {

   case -1:
      perror("fork failed");
      exit(20);
      break;
   default: /* Parent process */

   /* Close conn_fd and terminate the function */
   if (t_close(conn_fd) < 0) {
      t_error("t_close() failed for conn_fd");
      exit(21);
   }
   return;


   case 0: /* Child */

      /* Close listen_fd and transfer the file */
      if (t_close(listen_fd) < 0) {
         t_error("t_close() failed for listen_fd");
         exit(22);
      }
```

```
if (t_look(conn_fd) != 0) { /* Has connection abort arrived? */
   fprintf(stderr, "t_look: unexpected event \n");
   exit(25);
}

while ((nbytes = fread(buf, 1, 1024, logfp)) > 0)
   if (t_snd(conn_fd, buf, nbytes, 0) < 0) {
      t_error("t_snd() failed");
      exit(26);
   }
```

After the *fork()*, the parent process returns to the main loop and waits for new connection requests.

The child process manages the new connection in the meantime. If the *fork()* call fails, *exit()* closes the established connection and sends an abort message to the client. This causes the *t_connect()* call of the client to fail.

The child process reads 1024 bytes of the protocol file and sends the data with the *t_snd()* call to the client. *buf* points to the start of the data buffer and *nbytes* defines the number of characters to be transferred.

If the user makes too much data available to the transport provider for transfer, the transport provider can refuse acceptance to ensure correct flow control. In this case, the *t_snd()* call blocks until the flow control is released again and the transfer can proceed. The *t_snd()* call is then not terminated until the transport provider is passed as many characters as defined by the *nbytes* variable.

The *t_snd()* function does not check whether an abort request arrived until the data is passed to the transport provider. Because the data flow is in just one direction it is also not possible for the user to handle incoming events. If, for example, the connection is interrupted, the user should be informed that data could be lost. The user can call *t_look()* before each *t_snd()* call to check whether incoming events arrived.

Data reception by the client

In the example, the server transfers a file to the client over the established connection. The client receives the file and directs it to the standard output. The client uses the following program section to receive the data:

```
while ((nbytes = t_rcv(fd, buf, 1024, &flags)) != -1)
   if (fwrite(buf, 1, nbytes, stdout) == 0) {
      fprintf(stderr, "fwrite failed \n");
      exit(5);
   }
}
```

The client calls the *t_rcv()* function to receive the incoming data. If no data is available, the process is blocked by the *t_rcv()* call until data is available. *t_rcv()* then returns the number of bytes in the receive buffer *buf* (maximum 1024). The client then writes the received data to the standard output. The data transfer is terminated when the *t_rcv()* call fails, which happens if a connection shutdown request is received. This is explained in more detail on the following page.

If the *fwrite()* call fails, the program is terminated and the transport endpoint is closed. Closing a transport endpoint (with *exit()* or *t_close()*) in the data transfer phase causes a connection abort and the communications partner receives an abort message.

Connection shutdown using the example client/server model

As already mentioned, there are two different forms of connection shutdown that can be supported by the transport provider.

– The abortive connection release terminates a connection immediately. This can lead to loss of data if all data has not reached the receiver.

  Any user can initiate such an abort by calling the *t_snddis()* function. If problems occur within the transport provider, the transport provider can also initiate a connection abort.

  When the abort message reaches the receiver, he has to call the *t_rcvdis()* function to receive the message. *t_rcvdis()* returns a value which defines the reason for the abort as a result. This value is dependent on the transport provider used and should not be interpreted by protocol-independent programs.

– The *orderly connection shutdown* terminates a connection only after all data has been transferred.

All transport providers must support the first variant, i.e. abortive connection release. In the example, it is implied that the transport provider also allows the orderly connection shutdown.

Connection shutdown by the server

Once all data has been transferred, the server can initiate an orderly connection shutdown as follows:

```
if (t_sndrel(conn_fd) < 0) {
   t_error("t_sndrel() failed");
   exit(27);
}
```

The connection is only shut down after both ends have sent a shutdown request and each has received a confirmation (see page 153).

Connection shutdown by the client

The connection shutdown progresses in the same way from the viewpoint of the client as it does from the viewpoint of the server. As already mentioned, the client receives data until the *t_rcv()* call fails. If the server calls either *t_snddis()* or *t_sndrel()*, the *t_rcv()* call fails and *t_errno* is set to T_LOOK. The client handles this condition as follows:

```
   if ((t_errno == TLOOK)  &&  (t_look(fd) == T_ORDREL)) {
      if (t_rcvrel(fd) < 0) {
         t_error("t_rcvrel() failed");
         exit(6);
      }
      if (t_sndrel(fd) < 0) {
         t_error("t_sndrel() failed");
         exit(7);
      }
      exit(0);
   }
   t_error("t_rcv() failed");
   exit(8);
}
```

When an event arrives at the transport endpoint of the client, the client checks whether the expected request for orderly shutdown has arrived. If it has, the client calls *t_rcvrel()* to receive the request. The client then calls *t_sndrel()*. This indicates to the server that the client is also ready to shut the connection down. At this point, the client program is terminated, also causing the transport endpoint to be closed.

If the transport provider does not support the orderly connection shutdown discussed above, the users must employ the abortive connection release. The users themselves are then responsible for ensuring that the connection shutdown does not cause data to be lost. For example, a specific combination of bytes can be used to indicate that the connection is to be terminated. There are many ways of preventing data loss. Each application and each higher protocol must have a mechanism that adjusts itself to the prevailing transport environment.

7.2 Connectionless service

The connectionless service is packet-oriented and supports the transfer of datagrams. Datagrams are fully addressed units of data which, from the viewpoint of the transport provider, have no logical relationship to each other.

Connectionless services are of interest to applications which

- only communicate briefly with a partner,
- can be dynamically configured,
- do not require guaranteed delivery of the data in the same order as sent.

Connectionless services are therefore preferably used for short request/reply dialogs as are, for example, typical for transaction systems.

7.2.1 Phases of the connectionless service

The connectionless service comprises the following two phases:

- local management
- data transfer

Local management

The same functions are needed for the local management as with a connection-oriented service (see section “Connection-oriented service” on page 150).

Data transfer

The data transfer allows the user to send datagrams to another user. Each datagram must contain the complete destination address. This message-based data exchange is supported by the *t_sndudata()* and *t_rcvudata()* functions.

Table 6 shows the functions for connectionless data transfer.

Function	Description
t_rcvudata()	Receives a message from another user.
t_rcvuderr()	Receives error information about a previously sent message.
t_sndudata()	Sends a message to a specific user.

Table 6: Functions for connectionless data transfer

Interaction of the connectionless service functions

Figure 5 illustrates the interaction between the XTI functions which implement the two phases of the connectionless service. The separate XTI functions are described in detail in chapter “XTI(POSIX) library functions” on page 223.

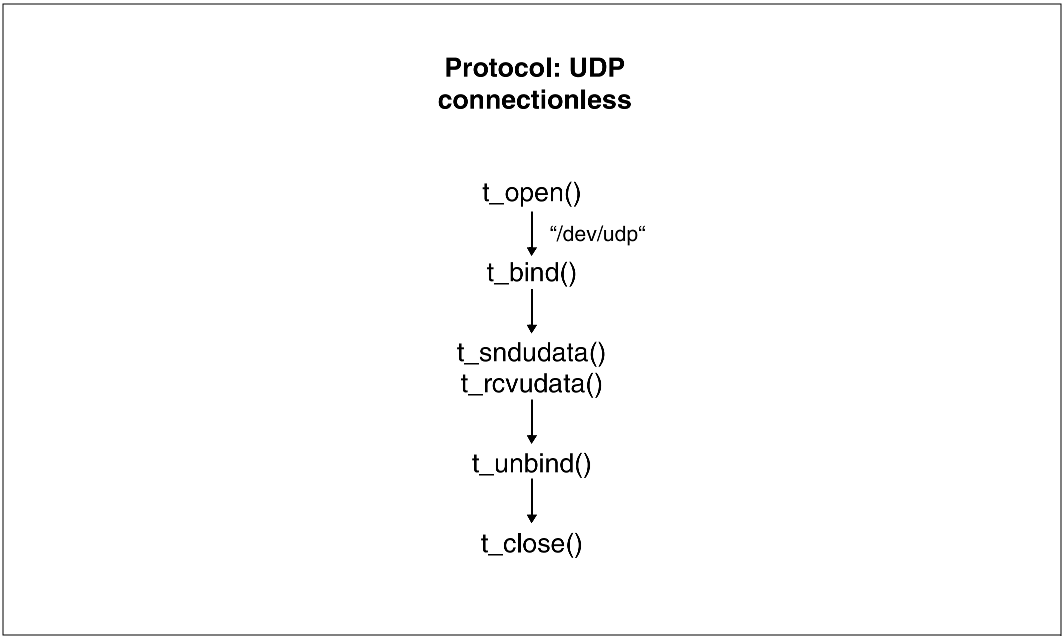

Figure 5: Interaction of the connectionless service functions

7.2.2 Connectionless service using an example transaction system

The connectionless service is explained in more detail using an example transaction system: The server waits for incoming requests and then processes and answers them.

Local management using an example transaction system

As with the connection-oriented service, the user has to execute the local management before transferring data. The user has to call an appropriate connectionless service with *t_open()* and then bind his address to the transport endpoint with *t_bind()*.

The user can employ the *t_optmgmt()* function to change the protocol features. As with the connection-oriented service, each transport provider has its own features. Using *t_optmgmt()* therefore makes the programs dependent on the protocol used.

The **server** executes the local management with the following definitions and calls:

```
#include <stdio.h>
#include <fcntl.h>
#include <xti.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define SRV_ADDR 0x7F000001
#define SRV_PORT 8888

main()
{
   int fd;
   nt flags;

   struct t_bind *bind;
   struct t_unitdata *ud;
   struct t_uderr *uderr;
   struct sockaddr_in *sin

   if ((fd = t_open("/dev/udp", O_RDWR, NULL)) < 0) {
      t_error("The transport provider cannot be opened");
      exit(1);
   }

   if ((bind = (struct t_bind *)t_alloc(fd,T_BIND, T_ADDR)) == NULL) {
      t_error("t_alloc() of the t_bind structure failed");
      exit(2);
   }
```

```
bind->addr.len=sizeof(struct sockaddr_in);
sin=(struct sockaddr_in *)bind->addr.buf;
sin->sin_family=AF_INET;
sin->sin_port=htons(SRV_PORT);
sin->sin_addr.s_addr=htonl(SRV_ADDR);

bind->qlen = 0;

if (t_bind(fd, bind, bind) < 0) {
   t_error("t_bind() failed");
   exit(3);
}
```

The server creates a transport endpoint by calling *t_open()*.

The server uses *t_bind()* to bind a specific address to the transport endpoint, to enable potential clients to recognize and access the server. The server uses *t_alloc()* to create an object of data type *t_bind* and supplies the *buf* and *len* components with appropriate values in the *addr* component of the *t_bind* object. The address itself is structured according to the address structure of the Internet communications domain.

One important difference between the connection-oriented and connectionless services is that the contents of the *t_bind qlen* component are meaningless in the connectionless service: all user datagrams can be received as soon as the *t_bind()* call has ended. During connection setup with the connection-oriented service, the transport provider defines a client/server relationship. Such a relationship does not exist with the connectionless mode. In this example, it is not the transport provider that defines a client/server relationship, but rather the application type.

Data transfer using an example transaction system

As soon as the user has bound an address to the transport endpoint, he can send and receive datagrams. Each message sent is accompanied by the address of the receiver.

The following series of calls show the **server** in the data transfer phase:

```
if ((ud = (struct t_unitdata *)t_alloc(fd,T_UNITDATA, T_ALL)) == NULL) {
   t_error("t_alloc() of the t_unitdata structure failed");
   exit(5);
}
```

```
    if ((uderr = (struct t_uderr *)t_alloc(fd, T_UDERROR, T_ALL)) == NULL) {
       t_error("t_alloc() of the t_uderr structure failed");
       exit(6);
    }
    for(;;) {
       if (t_rcvudata(fd, ud, &flags) < 0) {
          if (t_errno == TLOOK) {

                  /*
                   *  Error with a previously sent datagram
                   */

                  if (t_rcvuderr(fd, uderr) < 0) {
                  t_error(„t_rcverr failed“);
                  exit(7);
                  }
                  fprintf(stderr,
                  "Faulty datagram, error = %d \n",
                  uderr->error);
                  continue;
          }
       t_error("t_rcvudata() failed");
       exit(8);
       }

       /*
        * query() processes the request and writes the reply
        * in ud->udata.buf and the length in ud->udata.len
        */

       query(ud);

       if (t_sndudata(fd, ud, 0) < 0) {
          t_error("t_sndudata() failed");
          exit(9);
       }
    }
}

query()
{
/* Only an extract, for simplification reasons */

}
```

To store datagrams, the server must first create an object of data type *struct t_unitdata* with *t_alloc()*.

The *t_unitdata* structure is declared in <xti.h> as follows:

```
struct t_unitdata {
   struct netbuf addr;
   struct netbuf opt;
   struct netbuf udata;
};
```

addr contains the address of the sender for incoming datagrams and the address of the receiver for outgoing datagrams. *opt* specifies possible options of the employed protocol that are to be used on this datagram. *udata* contains the user data. *addr*, *buf* and *udata* must be provided with buffers of sufficient size to store incoming datagrams. As described in section "Connection-oriented service" on page 150, this is ensured by specifying T_ALL with the *t_alloc()* call. The *maxlen* component of each component (of type *struct netbuf*) of the created *t_unitdata* object is supplied with an appropriate value by *t_alloc()*.
The server also creates an object of type *struct t_uderr* for processing datagram errors (see page 179).

The server runs in an endless loop. It receives requests, processes them and replies to the clients. *t_rcvudata()* is called first to receive the next request. *t_rcvudata()* receives the next possible datagram. If no datagrams are available, *t_rcvudata()* blocks the process until a datagram is received. The second parameter of the *t_rcvudata()* call specifies the *t_unitdata* object in which the datagram is to be stored.

The third parameter (*flags*) must be a pointer to an integer value. This value can be set to T_MORE when *t_rcvudata()* is ended to indicate that the *udata* buffer was not large enough to accept the complete datagram. In this case, additional *t_rcvudata()* calls supply the remaining part of the datagram.
Since the buffer in this example was created with *t_alloc()*, this case cannot occur and the server does not need to test *flags*.

Once a datagram has been successfully received, the server calls *query()* to process the request.

Datagram errors

If the transport provider cannot process a datagram passed with *t_sndudata()*, a T_UDERR error is reported to the user. With this error, the datagram address and options are returned together with a protocol-dependent error value. The described condition can, for example, occur if the transport provider does not find the specified destination address.

It is expected that each protocol defines all causes for a datagram not being sent.

The error indication does not provide information as to whether the datagram was successfully sent. The transport protocol decides how the error indication is used. It must be emphasized once more at this point that the connectionless service does not guarantee reliable data delivery.

The server is informed of the error as soon as it tries to receive a datagram. The *t_rcvudata()* call fails and *t_errno* is set to TLOOK. If *t_errno* is set to TLOOK, only an T_UDERR can have occurred so the server calls *t_rcvuderr()* to determine the cause of the error. The second parameter of the *t_rcvuderr()* call is a previously created object of data type *struct t_uderr*. This object is supplied with values by the *t_rcvuderr()* call.

The *t_uderr* structure is declared in <xti.h> as follows:

```
struct t_uderr {
   struct netbuf addr;
   struct netbuf opt;
   long error;};
```

addr and *opt* specify the destination address and the options set for this datagram. *error* indicates a protocol-dependent error value which specifies why the datagram was not processed. The server outputs the error value and then returns to the normal loop.

7.3 States and state transitions

The following tables describe:

- transport interface states
- transport interface and *t_look()* events
- outgoing events
- incoming events
- state transitions caused by transport system user actions
- transport interface state transitions
- TLOOK error events

Transport interface states

The following table describes the states used to describe the state transitions of the transport interface.

State	Meaning	Service type
T_UNINIT	Not initialized. Start and end state of the interface	T_COTS, T_COTS_ORD, T_CLTS
T_UNBND	Initialized but not bound	T_COTS, T_COTS_ORD, T_CLTS
T_IDLE	No connection established	T_COTS, T_COTS_ORD, T_CLTS
T_OUTCON	Outgoing connection waiting for the server	T_COTS, T_COTS_ORD
T_INCON	Incoming connection waiting for the server	T_COTS, T_COTS_ORD
T_DATAXFER	Data transfer	T_COTS, T_COTS_ORD
T_OUTREL	Orderly connection shutdown (waiting for confirmation for orderly connection shutdown)	T_COTS_ORD
T_INREL	Incoming orderly connection shutdown (waiting for request for orderly connection shutdown)	T_COTS_ORD

Table 7: Transport interface states

Transport interface and t_look() events

The user can call the *t_look()* function to determine which event has occurred if a TLOOK error is reported. The TLOOK error has a special significance for the transport interface. TLOOK informs the user when a function of the interface was interrupted by an unexpected asynchronous event on the specified transport endpoint. An error indicated by TLOOK must therefore not be interpreted as an interface error. The called function is not executed because of the pending event.

The following transport interface events are defined:

Event	Meaning
T_LISTEN	A connection request arrived at the transport endpoint. T_LISTEN can only occur with a transport endpoint which is assigned an address with *qlen* > 0.
T_CONNECT	Confirmation of a previously sent connection request has arrived. The confirmation is sent when the server accepts a connection request.
T_DATA	User data has arrived.
T_DISCONNECT	A message reporting that a connection has been aborted or refused has arrived
T_ORDREL	The request for an orderly connection shutdown has arrived.
T_UDERR	The report about an error with a previously sent datagram has arrived.

Table 8: Transport interface events

Events at TLOOK error

Table 9 describes the asynchronous events which lead to terminating an XTI function with the TLOOK error.

XTI function	**Event**
t_accept()	T_DISCONNECT, T_LISTEN
t_connect()	T_DISCONNECT, T_LISTEN
t_listen()	T_DISCONNECT
t_rcv()	T_DISCONNECT, T_ORDREL
t_rcvconnect()	T_DISCONNECT
t_rcvrel()	T_DISCONNECT
t_rcvudata()	T_UDERR
t_snd()	T_DISCONNECT, T_ORDREL
t_sndudata()	T_UDERR
t_unbind()	T_LISTEN, T_DATA
t_sndrel()	T_DISCONNECT
t_snddis()	T_DISCONNECT

Table 9: TLOOK error events

If execution of an XTI function leads to a TLOOK error on a transport endpoint, subsequent calls to the same or other XTI functions affected by the same TLOOK return the TLOOK error until the causing event has been handled. You can identify the event causing the TLOOK error with the XTI *t_look()* function and then handle it with a suitable other XTI function.

Outgoing events

The outgoing events are described in table 10 on page 185. They correspond to the values returned by the specified transport functions, where the functions send a request or reply to the transport provider.

Some of the events (e.g. *accept*) in the table are discriminated according to the context in which they occur. The context depends on the values of the following variables:

– *ocnt*
  Number of pending connection requests

– *fd*
  File descriptor of the current transport endpoint

– *refsd*
  File descriptor of the transport endpoint on which a connection is accepted

Event	Meaning	Service type
opened	Successful termination of *t_open()*	T_COTS, T_COTS_ORD, T_CLTS
bind	Successful termination of *t_bind()*	T_COTS, T_COTS_ORD, T_CLTS
optmgmt	Successful termination of *t_optmgmt()*	T_COTS_ORD, T_CLTS
unbind	Successful termination of *t_unbind()*	T_COTS, T_COTS_ORD, T_CLTS
closed	Successful termination of *t_close()*	T_COTS, T_COTS_ORD, T_CLTS
connect1	Successful termination of *t_connect()* in synchronous mode	T_COTS, T_COTS_ORD
connect2	TNODATA error with *t_connect()* in asynchronous mode or TLOOK error caused by a connection shutdown request arriving at the communications endpoint	T_COTS, T_COTS_ORD
accept1	Successful termination of *t_accept()* with *ocnt* == 1, *fd* == *resfd*	T_COTS, T_COTS_ORD
accept2	Successful termination of *t_accept()* with *ocnt* == 1, *fd* != *resfd*	T_COTS, T_COTS_ORD
accept3	Successful termination of *t_accept()* with *ocnt* >1	T_COTS, T_COTS_ORD
snd	Successful termination of *t_snd()*	T_COTS, T_COTS_ORD
snddis1	Successful termination of *t_snddis()* with *ocnt* <=1	T_COTS, T_COTS_ORD
snddis2	Successful termination of *t_snddis()* with *ocnt* >1	T_COTS, T_COTS_ORD
sndrel	Successful termination of *t_sndrel()*	T_COTS_ORD
sndudata	Successful termination of *t_sndudata()*	T_CLTS

Table 10: Outgoing events

Incoming events

The incoming events correspond to the successful return values of the specified function, where these functions receive data or information about events from the transport provider. The only incoming event that is not connected directly to the return value of a function is *pass_conn*. The *pass_conn* event occurs when a user transfers a connection to another transport endpoint. This event occurs on a transport endpoint to which the connection was transferred, although no transport interface function was called for it. The *pass_conn* event describes the behavior when a user accepts a connection on another transport endpoint.

In the following table, the *rcvdis* events are discriminated according to the context in which they occur. The context depends on the value of *ocnt*. The value of *ocnt* specifies the number of pending connection requests on the transport endpoint.

Event	**Meaning**	**Service type**
listen	Successful termination of *t_listen()*	T_COTS, T_COTS_ORD
rcvconnect	Successful termination of *t_rcvconnect()*	T_COTS, T_COTS_ORD
rcv	Successful termination of *t_rcv()*	T_COTS, T_COTS_ORD
rcvdis1	Successful termination of *t_rcvdis()* with *ocnt* <= 0	T_COTS, T_COTS_ORD
rcvdis2	Successful termination of *t_rcvdis()* with *ocnt* = = 1	T_COTS, T_COTS_ORD
rcvdis3	Successful termination of *t_rcvdis()* with *ocnt* > 1	T_COTS, T_COTS_ORD
rcvrel	Successful termination of *t_rcvrel()*	T_COTS_ORD
rcvudata	Successful termination of *t_rcvudata()*	T_CLTS
rcvuderr	Successful termination of *t_rcvuderr()*	T_CLTS
pass_conn	Receive a transferred connection	T_COTS, T_COTS_ORD

Table 11: Incoming events

State transitions caused by transport system user actions

In the state tables listed under “State tables” (see page 188), some state transitions are accompanied by a series of actions that have to be carried out by the transport service user. These actions are identified with the notation “[*n*]”, where *n* is the number of the action to be executed.

The actions concerned are as follows:

1. Set the number of pending connection requests to 0.
2. Increment the number of pending connection requests.
3. Decrement the number of pending connection requests.
4. Transfer a connection to another transport endpoint, as specified in *t_accept()*.

State tables

The state transitions of the transport interface are described in the following tables. The transition to the next state is shown for a current state and an event. All actions are also defined which are to be executed by the transport system user. Such actions are identified with "[*n*]".

The contents of each box indicate the follow-up state. This is dependent on the current state (at the head of the column) and the current incoming or outgoing event (at the left in the line concerned). An empty box means that the state/event combination concerned is invalid. Each box can contain an action list (as described in the previous section) in addition to the follow-up state. The transport service user must execute the actions in the listed order.

You should note the following points when reading the state tables:

– The *t_close()* function is also handled in the state tables (see the *closed* event in table 12). However, *t_close()* can be called from any state to close a transport endpoint. If the address is bound to a transport endpoint, calling *t_close()* automatically releases the address.

– The transport provider detects when a transport service user calls a function outside the defined order. In this case, the transport provider refuses the function and sets *t_errno* to TOUTSTATE. The state does not change.

– If a different transport error occurs, the state does not normally change. An exception to this is a TLOOK or TNODATA error with *t_connect()*. Other exceptions are noted explicitly in the description of the functions in chapter "XTI(POSIX) library functions" on page 223. In the state tables, it is assumed that the transport interface is used correctly.

– The *t_getinfo()*, *t_getstate()*, *t_alloc()*, *t_free*(), *t_sync()*, *t_look()* and *t_error()* functions are not included in the state tables as they do not affect the state.

The state transitions in the following phases are each handled in a separate table:

– local management (connection-oriented and connectionless service)

– data transfer in the connectionless service

– connection setup, data transfer and connection shutdown in the connection-oriented service

Event	State		
	T_UNINIT	**T_UNBND**	**T_IDLE**
opened	T_UNBND		
bind		T_IDLE [1]	
optmgmt			T_IDLE
unbind			T_UNBND
closed		T_UNINIT	

Table 12: Local management state transitions

Event	State
	T_IDLE
sndudata	T_IDLE
rcvudata	T_IDLE
rcvuderr	T_IDLE

Table 13: Connectionless service state transitions

Event	State					
	T_IDLE	T_OUTCON	T_INCON	T_DATAXFER	T_OUTREL	T_INREL
connect1	T_DATAXFER					
connect2	T_OUTCON					
rcvconnect		T_DATAXFER				
listen	T_INCONN [2]		T_INCONN [2]			
accept1			T_DATAXFER [3]			
accept2			T_IDLE [3][4]			
accept3			T_INCON [3][4]			
snd				T_DATAXFER		T_INREL
rcv				T_DATAXFER	T_OUTREL	
snddis1		T_IDLE	T_IDLE [3]	T_IDLE	T_IDLE	T_IDLE
snddis2			T_IDLE [3]			
rcvdis1		T_IDLE		T_IDLE	T_IDLE	T_IDLE
rcvdis2			T_IDLE [3]			
rcvdis3			T_INCON [3]			
sndrel				T_OUTREL		T_IDLE
rcvrel				T_INREL	T_IDLE	
pass_conn	T_DATAXFER					

Table 14: Connection-oriented service state transitions

8 Advanced XTI(POSIX) concepts

Some of the most important advanced concepts of the transport interface are discussed in this chapter:

- asynchronous execution mode
- simultaneous management of multiple connections by the server and event-controlled operation of multiple connections by the server

8.1 Asynchronous execution mode

Many of the transport interface functions can block a process if they wait for specific events or block the process data flow. However, there are situations where the user will want to prevent this blocking. For example, time-critical applications should never be blocked. In another case, the process wants to continue working while waiting for a transport interface event.

Each function which could block the process can therefore be executed in a special non-blocking (asynchronous) mode. The *t_listen()* call normally blocks the calling process (server) until the connection is confirmed. However, the server could also use the non-blocking *t_listen()* call to periodically check whether the connection has been set up. The asynchronous mode is enabled with the O_NONBLOCK parameter for the file ID concerned. This can be done with *t_open()* when the transport endpoint is opened or with an *fcntl()* call before a prospective blocking transport interface function is called. *fcntl()* can be used at any time to enable/disable the asynchronous mode.
All program examples in this chapter use the default synchronous mode.

8.2 Managing multiple connections simultaneously and event-controlled operation

An example is used in the following section to illustrate two important concepts:

- Simultaneous management of multiple connections by the server:

  The server application shown in chapter "XTI(POSIX) basics" on page 149 can only process one connection request at a time. However, the transport interface also allows several connections to be processed simultaneously. This is, for example, meaningful in the following cases:

  – The server wishes to assign a priority to each client.

  – Several clients wish to set up a connection to a server which is currently processing a connection request. If the server can only process one connection request at any one time, the clients find the server in an occupied state. However, if the server can process several connections simultaneously, the clients will only find the server in an occupied state if the server is already processing the maximum possible number of clients requests.

- Programming event-controlled operation:

  The programmer can write event-controlled programs using the transport interface. With an event-controlled server, the process continuously polls a transport endpoint to check if events have been reported by the transport interface. The server then calls the interface function appropriate to the reported event.

The following example program uses the same definitions and calls for the local management as the example server in section "Connection-oriented service" on page 150. The program code used in the example is shown completely and coherently in section "Event-controlled server" on page 210.

```
#include <xti.h>
#include <fcntl.h>
#include <stdio.h>
#include <poll.h>
#include <stropts.h>
#include <signal.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```
#define FILENAME "/etc/services"
#define NUM_FDS 1
#define MAX_CONN_IND 4
#define SRV_ADDR 0x7F000001
#define SRV_PORT 8888

int conn_fd;/* Server transport endpoint */

/* For storing the connections */
struct t_call *calls[NUM_FDS][MAX_CONN_IND];

main()
{
   struct pollfd pollfds[NUM_FDS];
   struct t_bind *bind;
   struct sockaddr_in *sin;
   int i;

   /*
    * Open a transport endpoint and bind the address.
    * However, multiple endpoints are supported.
    */

   if ((pollfds[0].fd = t_open("/dev/tcp", O_RDWR, NULL)) < 0) {
   t_error("t_open() failed");
   exit(1);
   }

   if ((bind = (struct t_bind *)t_alloc(pollfds[0].fd,
                                          T_BIND, T_ALL)) == NULL) {
      t_error("t_alloc() of t_bind structure failed");
      exit(2);
   }
   bind->qlen = MAX_CONN_IND;
   bind->addr.len=sizeof(struct sockaddr_in);
   sin=(struct sockaddr_in *)bind->addr.buf;
   sin->sin_family=AF_INET;
   sin->sin_port=htons(SRV_PORT);
   sin->sin_addr.s_addr=htonl(SRV_ADDR);
   if (t_bind(pollfds[0].fd, bind, bind) < 0) {
      t_error("t_bind() failed");
      exit(3);
   }
```

The file ID returned by *t_open()* is stored in the first member of the *struct pollfd* vector *pollfds* (see the *pollfd* structure on page 138). The *pollfds* vector is used later when calling the POSIX *poll()* function to process incoming events. *poll()* is a general C library function which is described on page 138. It must be noted that just one transport endpoint is set up in this example. However, since the remaining part of the example is laid out for multiple connections, only minor changes have to be made to manage multiple communications connections with this program.

An important point for this example is that the server sets *bind->qlen* to a value >1 and passes it with the *t_bind()* call. This makes it possible for the server to receive multiple connection requests on one transport endpoint. In the examples in chapter “XTI(POSIX) basics”, the server always accepts and processes just one connection at a time. In contrast to this, the server can accept up to MAX_CONN_IND requests simultaneously in this example. However, the transport provider may reduce the value of *bind->qlen* if he cannot process the number of connections required by the server.

The server proceeds as follows after making its address known:

```
   pollfds[0].events = POLLIN;

   for(;;) {
      if (poll(pollfds, NUM_FDS, -1) < 0) {
         perror("poll() failed");
         exit(5);
      }

      for (i = 0; i < NUM_FDS; i++) {

         switch (pollfds[i].revents) {

         default:
                 perror("Poll returns an error message");
                 exit(6);
                 break;

         case 0:
                 continue;

         case POLLIN:
                 do_event(i, pollfds[i].fd);
                 service_conn_ind(i, pollfds[i].fd);
         }
      }
   }
}
```

The *events* component of the first member of the *pollfd* vector *pollfds* sets the server to POLLIN so that it is informed about all events arriving at the transport interface. The server then goes into an endless loop, waits for events at the transport endpoints with *poll()* and processes these events accordingly.

The *poll()* call blocks the process until an event arrives. After the end of the call, the server checks each entry (corresponding to one transport endpoint) to see if an event has occurred there. If *revents* is set to 0, no events arrived at this endpoint. If *revents* is set to POLLIN, an event has arrived at this endpoint. In this case, the server calls *do_event()* to process the event. If *revents* has a value other than POLLIN, an error has occurred at this endpoint and the program is terminated.

In each loop run in which an event is found, the server calls *service_conn_ind()* (see page 198) to process any pending requests. If a further request is pending while one is being processed, *service_conn_ind* is exited immediately, whereby the current request is stored for later processing with the *do_event()* function.

The *do_event()* function is called to process an incoming event and is defined as follows:

```
do_event(slot, fd)
{
   struct t_discon *discon;
   int i;

   switch (t_look(fd)) {

   default:
      fprintf(stderr,"t_look: unexpected event \n");
      exit(7);
      break;

   case -1:
      t_error("t_look() failed");
      exit(9);
      break;


   case 0:
      /* This should never happen if POLLIN is reported */
      fprintf(stderr,"t_look() reports no event\n");
      exit(10);


   case T_LISTEN:

      /*
       * Search for a free member in the calls field
       */
      for (i = 0; i < MAX_CONN_IND; i++) {
         if (calls[slot][i] == NULL)
                 break;
      }
```

```
      if ((calls[slot][i] = (struct t_call *)t_alloc(fd,
                                      T_CALL, T_ALL)) == NULL) {
         t_error("t_alloc() of t_call structure failed");
         exit(11);
      }

      if (t_listen(fd, calls[slot][i]) < 0) {
         t_error("t_listen() failed");
         exit(12);
      }
      break;

   case T_DISCONNECT:
      discon = (struct t_discon *)t_alloc(fd, T_DIS, T_ALL);

      if (t_rcvdis(fd, discon) < 0) {
         t_error("t_rcvdis() failed");
         exit(13);
      }
      /*
       * Find and delete request in calls field
       */
      for (i = 0; i < MAX_CONN_IND; i++) {
         if (discon->sequence == calls[slot][i]->sequence) {
            t_free(calls[slot][i], T_CALL);
            calls[slot][i] = NULL;
         }
      }
      t_free(discon, T_DIS);
      break;
   }
}
```

The *do_event()* function has two parameters: a number *slot* and a file ID *fd.*
slot indexes the vectors (submatrices) of the global *calls* matrix whose members are pointers to *t_call* objects. Each transport endpoint to be interrogated is represented by a vector in the *calls* matrix. The value of *slot* therefore specifies the transport endpoint to be processed. The vector members point to the *t_call* objects in which the incoming requests of the transport endpoint concerned are stored.

The *t_look()* call gets the event which occurred on the transport endpoint identified by *fd.* If a connection request (T_LISTEN) or an abort request (T_DISCONNECT) has arrived, it is processed accordingly. With other events, an appropriate error message is output and the program is terminated.

With a connection request, *do_event()* searches for a free entry in the *calls* field. A *t_call* object is now requested for this entry. The request is received with *t_listen()*. This field must always contain at least one free field as *t_bind()* specified the maximum number of requests than could be processed simultaneously when it created the field. The request is processed later.

An incoming abort request must belong to a connection request that arrived earlier. This is true if a client sends a connection request and then aborts it immediately. *do_event()* creates a *t_discon* structure to receive the information for the abort.

The *t_discon* structure is declared in <xti.h> as follows:

```
struct t_discon {
   struct netbuf udata;
   int reason;
   int sequence;
};
```

reason specifies the protocol-specific reason for the connection shutdown. *sequence* specifies the number of the connection request that is to be aborted.

The *t_rcvdis()* function is called to receive the above information. **calls* of the program in which the requests are managed is then searched for the request specified in the *sequence* component. Once the request is found, the memory is released and the entry set to NULL.

If any event has occurred at the transport endpoint, the *service_conn_ind()* function is called to process all requests pending on this endpoint as follows:

```
service_conn_ind(slot, fd)
{
   int i;

   for (i = 0; i < MAX_CONN_IND; i++) {
      if (calls[slot][i] == NULL)
         continue;

      if ((conn_fd = t_open("/dev/tcp", O_RDWR, NULL)) < 0) {
         t_error("t_open() failed");
         exit(14);
      }

      if (t_accept(fd, conn_fd, calls[slot][i]) < 0) {
         if (t_errno == TLOOK) {
            t_close(conn_fd);
            return;
         }
         t_error("t_accept() failed");
         exit(16);
      }
```

```
        t_free(calls[slot][i], T_CALL);
        calls[slot][i] = NULL;

        run_service(fd);
    }
}
```

The field is searched for requests for the specified endpoint (*slot*). For each request, the server opens a transport endpoint and accepts the request. If, in the meantime, another event (connection request or connection shutdown) has arrived, the *t_accept()* call fails and *t_errno* is set to TLOOK.

A user cannot accept any requests if other connection or abort requests are pending on this transport endpoint.

When this error occurs, *conn_fd* is closed immediately and the function is exited. The request remains intact in the field and can therefore be processed at a later time. The server process is now back in the main loop and the event can be handled with the next *poll()* call. This method allows multiple requests to be processed simultaneously.

Once all events have been processed, the *service_conn_ind()* function can set up the connections and call the *run_service()* function to transfer the data. The *run_service()* function is described in section “Connection-oriented client/server model” on page 168.

9 Examples for XTI(POSIX)

Sections of example programs are shown and explained in chapter “XTI(POSIX) basics” on page 149 and chapter “Advanced XTI(POSIX) concepts” on page 191. These example programs are shown again in full and coherently in this chapter.

9.1 Client in the connection-oriented service

The following client program in the connection-oriented service is described in detail in section “Connection-oriented client/server model” on page 156. The client sets up a transport connection to a server, receives data from the server and writes the data to its standard output. The connection is shut down using the orderly connection shutdown of the transport interface. The client can communicate with any of the connection-oriented servers described in the examples in this chapter.

```
#include <stdio.h>
#include <xti.h>
#include <fcntl.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define SRV_ADDR 0x7F000001
#define SRV_PORT 8888

main()
{
   int fd;
   int nbytes;
   int flags = 0;
   char buf[1024];
   struct t_call *sndcall;
   struct sockaddr_in *sin;

   if ((fd = t_open("/dev/tcp", O_RDWR, NULL)) < 0) {
      t_error("t_open() failed");
      exit(1);
   }

   if (t_bind(fd, NULL, NULL) < 0) {
      t_error("t_bind() failed");
      exit(2);
   }

   if ((sndcall = (struct t_call *)t_alloc(fd,T_CALL, T_ADDR)) == NULL) {
      t_error("t_alloc() failed");
      exit(3);
   }
   sndcall->addr.len=sizeof(struct sockaddr_in);
   sin = (struct sockaddr_in *)sndcall->addr.buf;
   sin->sin_family=AF_INET;
   sin->sin_port=htons(SRV_PORT);
   sin->sin_addr.s_addr=htonl(SRV_ADDR);
```

```
    if (t_connect(fd, sndcall, NULL) < 0) {
       t_error("t_connect() failed for fd");
       exit(4);
    }

    while ((nbytes = t_rcv(fd, buf, 1024, &flags)) != -1) {
       if (fwrite(buf, 1, nbytes, stdout) == 0) {
          fprintf(stderr, "fwrite() failed\n");
          exit(5);
       }
    }

    if ((t_errno == TLOOK)  &&  (t_look(fd) == T_ORDREL)) {
       if (t_rcvrel(fd) < 0) {
          t_error("t_rcvrel() failed");
          exit(6);
       }
       if (t_sndrel(fd) < 0) {
          t_error("t_sndrel() failed");
          exit(7);
       }
       exit(0);
    }
    t_error("t_rcv() failed");
    exit(8);
 }
```

9.2 Server in the connection-oriented service

The following server program for the connection-oriented service is described in detail in section “Connection-oriented client/server model” on page 156. The server sets up a transport connection to a client and then passes a protocol file to this client. The connection is shut down using the orderly connection shutdown of the transport interface. The client in the connection-oriented service described in the previous section can communicate with the server described here.

```
#include <xti.h>
#include <stropts.h>
#include <fcntl.h>
#include <stdio.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define FILENAME "/etc/services"
#define DISCONNECT -1
#define SRV_ADDR 0x7F000001
#define SRV_PORT 8888

int conn_fd;  /* Connection setup */

main()
{
   int listen_fd;  /* Monitor transport endpoint */
   struct t_bind *bind;
   struct t_call *call;
   struct sockaddr_in *sin;

   if ((listen_fd = t_open("/dev/tcp", O_RDWR, NULL)) < 0) {
      t_error("t_open() failed for listen_fd");
      exit(1);
   }

   if ((bind = (struct t_bind *)t_alloc(listen_fd,T_BIND, T_ALL)) == NULL) {
      t_error("t_alloc() of the t_bind structure failed");
      exit(2);
   }
   bind->qlen = 1;
   bind->addr.len=sizeof(struct sockaddr_in);
   sin = (struct sockaddr_in *)bind->addr.buf;

   sin->sin_family=AF_INET;
   sin->sin_port=htons(SRV_PORT);
   sin->sin_addr.s_addr=htonl(SRV_ADDR);
```

```
    if (t_bind(listen_fd, bind, bind) < 0) {
       t_error("t_bind() for listen_fd failed");
       exit(3);
    }

    if ((call = (struct t_call *)t_alloc(listen_fd,T_CALL, T_ALL)) == NULL) {
       t_error("t_alloc() of t_call structure failed");
       exit(5);
    }

    for(;;) {
       if (t_listen(listen_fd, call) < 0) {
          t_error("t_listen() for listen_fd failed");
          exit(6);
       }

       if ((conn_fd = accept_call(listen_fd, call)) != DISCONNECT)
          run_server(listen_fd);
    }
 }

 accept_call(listen_fd, call)
 int listen_fd;
 struct t_call *call;
 {
    int resfd;
    struct t_call *refuse_call;


    if ((resfd = t_open("/dev/tcp", O_RDWR, NULL)) < 0) {
       t_error("t_open() for responding fd failed");
       exit(7);
    }

    while (t_accept(listen_fd, resfd, call) < 0) {
       if (t_errno == TLOOK) {
          if (t_look(listen_fd) == T_DISCONNECT) { /* Connection abort */
                if (t_rcvdis(listen_fd, NULL) < 0) {
                   t_error("t_rcvdis() failed for listen_fd");
                   exit(9);
                }
                if (t_close(resfd) < 0) {
                   t_error("t_close failed for responding fd");
                   exit(10);
                }
```

```
                /* Terminate call and wait for further call */
                return(DISCONNECT);
            } else { /* New T_LISTEN; delete event */
                if ((refuse_call =
                   (struct t_call*)t_alloc(listen_fd,T_CALL,0)) == NULL) {
                   t_error("t_alloc() for refuse_call failed");
                   exit(11);
                }

                if (t_listen(listen_fd, refuse_call) < 0) {
                   t_error("t_listen() for refuse_call failed");
                   exit(12);
                }

                if (t_snddis(listen_fd, refuse_call) < 0) {
                   t_error("t_snddis() for refuse_call failed");
                   exit(13);
                }

                if (t_free((char *)refuse_call, T_CALL) < 0) {
                   t_error("t_free() for refuse_call failed");
                   exit(14);
                }
            }
         } else {
            t_error("t_accept() failed");
            exit(15);
         }
      }

      return(resfd);
   }

   run_server(listen_fd)
   int listen_fd;
   {
      int nbytes;
      FILE *logfp; /* File pointer to log file */
      char buf[1024];

      switch (fork()) {

      case -1:
         perror("fork failed");
         exit(20);
         break;
```

```
        default: /* Parent process */
           /* Close conn_fd and remain active as monitor again */
           if (t_close(conn_fd) < 0) {
              t_error("t_close() for conn_fd failed");
              exit(21);
           }
           return;

        case 0:  /* Child */

           /* Close listen_fd and execute service */
           if (t_close(listen_fd) < 0) {
              t_error("t_close() for listen_fd failed");
              exit(22);
           }
           if ((logfp = fopen(FILENAME, "r")) == NULL) {
              perror("Log file cannot be opened");
              exit(23);
           }

           if (t_look(conn_fd) != 0) { /* Was there an interruption? */
              fprintf(stderr, "t_look: unexpected event\n");
              exit(25);
           }

           while ((nbytes = fread(buf, 1, 1024, logfp)) > 0)
              if (t_snd(conn_fd, buf, nbytes, 0) < 0) {
                 t_error("t_snd() failed");
                 exit(26);
              }

           if (t_sndrel(conn_fd) < 0) {
              t_error("t_sndrel() failed");
              exit(27);
           }
           while(t_look(conn_fd) == 0) {
              sleep(1);
           }
           if(t_look(conn_fd) == T_DISCONNECT) {
              fprintf(stderr, "Connection aborted\n");
              exit(12);
           }
           exit(0);
        }
     }
```

9.3 Datagram-oriented transaction server

The following program for a transaction system in connectionless mode is described in detail in section “Connectionless service using an example transaction system” on page 175. The server waits for incoming requests for data packets, then processes each request and sends a reply.

```
#include <stdio.h>
#include <fcntl.h>
#include <xti.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define SRV_ADDR 0x7F000001
#define SRV_PORT 8888

main()
{
   int fd;
   int flags;
   struct t_bind *bind;
   struct t_unitdata *ud;
   struct t_uderr *uderr;
   struct sockaddr_in *sin;


   if ((fd = t_open("/dev/udp", O_RDWR, NULL)) < 0) {
      t_error("Not possible to open /dev/udp");
      exit(1);
   }

   if ((bind = (struct t_bind *)t_alloc(fd,
      T_BIND, T_ADDR)) == NULL) {
      t_error("t_alloc() of the t_bind structure failed");
      exit(2);
   }
   bind->addr.len=sizeof(struct sockaddr_in);
   sin=(struct sockaddr_in *)bind->addr.buf;
   sin->sin_family=AF_INET;
   sin->sin_port=htons(SRV_PORT);
   sin->sin_addr.s_addr=htonl(SRV_ADDR);
   bind->qlen = 0;

   if (t_bind(fd, bind, bind) < 0) {
      t_error("t_bind() failed");
      exit(3);
   }
```

```
    if ((ud = (struct t_unitdata *)t_alloc(fd,
       T_UNITDATA, T_ALL)) == NULL) {
       t_error("t_alloc() of t_unitdata structure failed");
       exit(5);
    }

    if ((uderr = (struct t_uderr *)t_alloc(fd,
       T_UDERROR, T_ALL)) == NULL) {
       t_error("t_alloc() of t_uderr structure failed");
       exit(6);
    }

    for(;;) {
       if (t_rcvudata(fd, ud, &flags) < 0) {
          if (t_errno == TLOOK) {
                   /*
                    * Error because of previous datagram
                    */

                   if (t_rcvuderr(fd, uderr) < 0) {
                      t_error("t_rcvuderr() failed");
                      exit(7);
                   }
                   fprintf(stderr,
                   "Datagram error, error = %d\n",
                   uderr->error);
                   continue;
          }
          t_error("t_rcvudata() failed");
          exit(8);
       }
       /*
        * query() processes the request and writes the reply in
        * ud->udata.buf and the length in ud->udata.len
        */
       query(ud);

       if (t_sndudata(fd, ud, 0) < 0) {
          t_error("t_sndudata() failed");
          exit(9);
       }
    }
}

query()
{
/* Only an extract, for simplification reasons */
}
```

9.4 Event-controlled server

The following server program for the connection-oriented service is described in detail on page 193ff in chapter “Advanced XTI(POSIX) concepts”. The server manages several connection requests in an event-controlled manner. All of the connection-oriented clients described earlier in this chapter can communicate with this server.

```
#include <xti.h>
#include <fcntl.h>
#include <stdio.h>
#include <poll.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define FILENAME "/etc/services"
#define NUM_FDS 1
#define MAX_CONN_IND 4
#define SRV_ADDR 0x7F000001
#define SRV_PORT 8888

int conn_fd; /* Connection to server */

/* For storing the connections */
struct t_call *calls[NUM_FDS][MAX_CONN_IND];

main()
{
   struct pollfd pollfds[NUM_FDS];
   struct t_bind *bind;
   struct sockaddr_in *sin;
   int i;

   /*
    * Only open and bind one transport endpoint,
    * although it would also be possible for more
    */
   if ((pollfds[0].fd = t_open("/dev/tcp", O_RDWR, NULL)) < 0) {
      t_error("t_open() failed");
      exit(1);
   }

   if ((bind = (struct t_bind *)t_alloc(pollfds[0].fd,
                                        T_BIND, T_ALL)) == NULL) {
      t_error("t_alloc() of the t_bind structure failed");
      exit(2);
   }
```

```
    bind->qlen = MAX_CONN_IND;
    bind->addr.len=sizeof(struct sockaddr_in);
    sin=(struct sockaddr_in *)bind->addr.buf;
    sin->sin_family=AF_INET;
    sin->sin_port=htons(SRV_PORT);
    sin->sin_addr.s_addr=htonl(SRV_ADDR);

    if (t_bind(pollfds[0].fd, bind, bind) < 0) {
       t_error("t_bind() failed");
       exit(3);
    }

    pollfds[0].events = POLLIN;

    for(;;) {
       if (poll(pollfds, NUM_FDS, -1) < 0) {
          perror("poll() failed");
          exit(5);
       }

       for (i = 0; i < NUM_FDS; i++) {

          switch (pollfds[i].revents) {

          default:
                  perror(
                  "Poll returns error event");
                  exit(6);
                  break;
          case 0:
                  continue;

          case POLLIN:
                  do_event(i, pollfds[i].fd);
                  service_conn_ind(i, pollfds[i].fd);
          }
       }
    }
 }

 do_event(slot, fd)
 {
    struct t_discon *discon;
    int i;
```

```
switch (t_look(fd)) {

default:
   fprintf(stderr,"t_look: unexpected event\n");
   exit(7);
   break;

case -1:
   t_error("t_look() failed");
   exit(9);
   break;

case 0:
   /* If POLLIN is returned, this should not happen */
   fprintf(stderr,"No event returned from t_look()\n");
   exit(10);

case T_LISTEN:
   /*
    * Find free member in call area
    */
   for (i = 0; i < MAX_CONN_IND; i++) {
      if (calls[slot][i] == NULL)
              break;
   }

   if ((calls[slot][i] = (struct t_call *)t_alloc(fd,
                                      T_CALL, T_ALL)) == NULL) {
      t_error("t_alloc() of t_call structure failed");
      exit(11);
   }

   if (t_listen(fd, calls[slot][i]) < 0) {
      t_error("t_listen() failed");
      exit(12);
   }
   break;

case T_DISCONNECT:
   discon = (struct t_discon *)t_alloc(fd, T_DIS, T_ALL);

   if (t_rcvdis(fd, discon) < 0) {
      t_error("t_rcvdis() failed");
      exit(13);
   }
```

```
            /*
             * Find and delete ind call in area
             */
            for (i = 0; i < MAX_CONN_IND; i++) {
               if (discon->sequence == calls[slot][i]->sequence) {
                  t_free(calls[slot][i], T_CALL);
                  calls[slot][i] = NULL;
               }
            }
            t_free(discon, T_DIS);
            break;
         }
      }

      service_conn_ind(slot, fd)
      {
         int i;

         for (i = 0; i < MAX_CONN_IND; i++) {
            if (calls[slot][i] == NULL)
               continue;

            if ((conn_fd = t_open("/dev/tcp", O_RDWR, NULL)) < 0) {
               t_error("t_open() failed");
               exit(14);
            }

            if (t_accept(fd, conn_fd, calls[slot][i]) < 0) {
               if (t_errno == TLOOK) {
                     t_close(conn_fd);
                     return;
               }
               t_error("t_accept() failed");
               exit(16);
            }
            t_free(calls[slot][i], T_CALL);
            calls[slot][i] = NULL;

            run_server(fd);
         }
      }
```

```
run_server(listen_fd)
int listen_fd;
{
   int nbytes;
   FILE *logfp; /* Pointer to log file */
   char buf[1024];
   switch (fork()) {

   case -1:
      perror("fork() failed");
      exit(20);
      break;

   default: /* Parent process */

      /* Close conn_fd and monitor again */
      if (t_close(conn_fd) < 0) {
         t_error("t_close() failed for conn_fd");
         exit(21);
      }
      return;

   case 0:  /* Child process */

      /* Close listen_fd and execute service */
      if (t_close(listen_fd) < 0) {
         t_error("t_close() failed for listen_fd");
         exit(22);
      }
      if ((logfp = fopen(FILENAME, "r")) == NULL) {
         perror("Log file cannot be opened");
         exit(23);
      }

      if (t_look(conn_fd) != 0) { /* Is there already a disconnect? */
         fprintf(stderr, "t_look: unexpected event\n");
         exit(25);
      }

      while ((nbytes = fread(buf, 1, 1024, logfp)) > 0)
         if (t_snd(conn_fd, buf, nbytes, 0) < 0) {
             t_error("t_snd() failed");
             exit(26);
         }
```

```
         if (t_sndrel(conn_fd) < 0) {
            t_error("t_sndrel() failed");
            exit(27);
         }
         while(t_look(conn_fd) == 0) {
            sleep(1);
         }

         if(t_look(conn_fd) == T_DISCONNECT) {
            fprintf(stderr, "Connection aborted\n");
            exit(12);
         }
         exit(0);
      }
   }
```

10 XTI trace

XTI trace provides you with the means for creating trace information for the separate XTI calls of a communication process.

You can control the XTI trace using the XTITRACE environment variable.
You can use the XTITRACE variable to

- enable the XTI trace and
- define which information is to be collected.

You can alternatively enable the XTI trace at program runtime with the XTI *t_optmgmt()* function. The *t_optmgmt()* function is described in section “t_optmgmt() - manage transport endpoint options” on page 260.

By default, the logged trace information is saved in the directory for temporary files. You can use the *xtitrace* program to evaluate these files and output the trace information. You can define the evaluation scope by specifying special options with the *xtitrace* call.

The following is described in the sections below:

- How you set the parameters of the XTITRACE environment variable to log the desired trace information.
- How you output the logged trace information with the *xtitrace* program.

10.1 Setting the XTITRACE environment variable parameters

The first XTI call in a process evaluates the XTITRACE environment variable and, if necessary, enables the XTI trace. After the trace is enabled, the temporary XTIF*pid* trace file (*pid* specifies the process number) is opened in the desired directory (-f *dir* option, see the following page), if it is not already open. The trace data is written into this file.

If the XTIF*pid* cannot accept any further data, the subsequent trace data is written into the XTIS*pid* file. This file has the same function as the XTIF*pid* file. Once the XTIS*pid* is full, XTIF*pid* is cleaned up and the new trace data is then written into it. The trace mechanism can switch back and forth between the XTIF*pid* and XTIS*pid* files a number of times if necessary. With each file change, the process should save the data in the temporary file that was just written, into a permanent file. This prevents the logged trace information from being overwritten and allows it to be output at a later time with the *xtitrace* program.

The rw------- (0600) access rights are granted for the XTIF*pid* and XTIS*pid* trace files and they can be viewed under the user ID of the process. Memory is assigned dynamically for buffering the trace files. This memory, and the XTIF*pid* and XTIS*pid* files, remain assigned for the duration of the process.

The options specified for XTITRACE control the trace mechanism:

– The *s* and *S* options define the scope of information to be logged.
– The *r* option controls the cyclic overwriting of the XTIF*pid* and XTIS*pid* files.
– The *f* option controls the memory for the XTIF*pid* and XTIS*pid* files.

You set the XTITRACE environment variable parameters with the following statements:

```
XTITRACE="-option [ -r wrap][ -f dir]";

export XTITRACE;
```

-*option*

option defines the trace type. A value must be entered for *option* when a trace is enabled.

You can enter the following two values for *option*:

– s
Logs the XTI call function names and their parameters and return values. If errors occur, the values of *t_errno* and *errno* are logged and the error position *errpos*.

– S
Logs all information which is also logged if *s* is specified. If parameters occur which are passed as pointers, the values of the objects addressed by the pointers are also logged.
S should be specified in preference to *s*.

-r *wrap*

You input a decimal number for *wrap*.
wrap defines that the trace file is changed after *wrap* * BUFSIZ bytes: after each *wrap* * BUFSIZ logged bytes, the trace mechanism switches over from the XTIF*pid* file to the XTIS*pid* file and viceversa. The data in the file that is switched to is thereby overwritten in each case. The BUFSIZ constant is defined in <stdio.h>.

Default value for *wrap*: 512

-f *dir*

You use *dir* to specify the directory into which the XTIF*pid* and XTIS*pid* trace files are written.

Default value for *dir*: The standard */usr/tmp* directory

10.2 Outputting trace information with the xtitrace program

The *xtitrace* program reads the trace information generated by the XTI trace from one or more files. *xtitrace* processes this trace information according to the options specified for *xtitrace* and outputs the result to the standard output.

The status is 0 after successful execution of *xtitrace*, otherwise not equal to 0.

Calling the xtitrace program

You call the *xtitrace* program as follows:

`xtitrace[ -option] file ...`

-option

You use *option* to define which information of the trace file(s) specified by *file* ... is to be output. You can specify one or more of the following values for *option* with each *xtitrace* call.

– c
Outputs the trace information of XTI calls for the following actions:

  – Installation/deinstallation of a communications application
  – Setup/shutdown of a connection

  The XTI functions affected are: *t_accept()*, *t_bind()*, *t_close()*, *t_connect()*, *t_listen()*, *t_open()*, *t_rcvconnect()*, *t_rcvdis()*, *t_rcvrel()*, *t_snddis()*, *t_sndrel()* and *t_unbind()*.

– d
Outputs the trace information of XTI calls for data exchange.
The XTI functions affected are: *t_rcv()*, *t_rcvudata()*, *t_rcvuderr()*, *t_snd()* and *t_sndudata()*.

– m
Outputs the trace information of XTI calls not covered by the *c* and *d* options.
The XTI functions affected are: *t_alloc()*, *t_error()*, *t_free()*, *t_getinfo()*, *t_getstate()*, *t_look()*, *t_optmgmt()* and *t_sync()*.

– v
Outputs the trace information of all XTI calls and the values of the parameters and options concerned. If parameters occur which are passed as pointers, the values of the objects addressed by the pointers are also output. However, the last requires that the data concerned was recorded during tracing (see section “Setting the XTITRACE environment variable parameters” on page 218). If the *S* option was set for XTITRACE, you should specify the value *v* for *option*. Specifying *v* has the same effect as specifying *cdmv*.

file ...
You use *file* to specify the name of a file which contains binary trace data. You can also specify several file names.

XTI trace output format

The *xtitrace* program always starts its output with a header. After this, *xtitrace* writes the trace information for the separate XTI calls. Depending on the parameters set in the XTITRACE environment variable and *xtitrace* program, *xtitrace* outputs either a single line or several lines in different formats.

Header format

The header contains the following information:

- XTI library version number
- trace start date and time
- specified values of the *xtitrace* output option
- name(s) of the trace file(s) whose contents are output by *xtitrace*

Example

```
XTI    TRACE (Vx.x)              Mon Aug 11 15:13:34 1997
OPTIONS 'cdmv' , TRACE FILE 'XTIF00963'
```

Format of the first output line for a logged XTI call

The trace information for an XTI call always starts with a line that has the following format:

- The line starts with a time stamp:

  minutes:*seconds*.*milliseconds*   (e.g. 24:16.324)

  The millisecond accuracy depends on the hardware used.

- After the time stamp comes the recorded XTI call (e.g. *t_bind()*). This is followed by a list enclosed in parentheses and containing the parameters and their values for the XTI call concerned (in the order required by XTI). The parameter values shown are either in decimal (%d), hexadecimal (0x%x) or symbolic (%s) form. A parameter shown in hexadecimal notation always starts with 0x.

  The following applies for showing the parameters and their values:

  - Values of pointers are shown in hexadecimal.
  - With parameters of type integer (e.g. *fd*), the corresponding value can be shown in hexadecimal, decimal or symbolic form. Parameters and their values are separated by a blank.

- With XTI functions whose execution depends on the state of the transport endpoint, the trace log informs on whether the call blocks (default) or not (specification: O_NDELAY or O_NONBLOCK).

Format of additional output lines for a logged XTI call

xtitrace only outputs the trace information described below if the two following conditions are satisfied:

– The XTITRACE variable parameters were set with the *S* option for creating the trace.
– The *xtitrace* program parameters were set with the *v* option.

For parameters passed as pointers, *xtitrace* outputs the names and values of the data objects addressed by these pointers. The values of the data objects (e.g. structure components) are output in hexadecimal. The name conventions of parameters and structure components correspond to the naming conventions used in chapter “XTI(POSIX) library functions” on page 223.

The trace information on structure components also contains a few special characters which have the following meaning:

> The component concerned must be assigned a value by the calling communications application before the logged XTI function is called.

< A value is returned in the component concerned by the logged XTI function if the XTI function executes correctly.

- The value of the component concerned is meaningless for the logged XTI function.

If “---” is output instead of a component value, the component concerned has no value assigned.

Format of the last output line for a logged XTI call

The return value of the XTI function concerned is always output in the last line for a logged XTI call. If errors occur, *t_errno*, possibly *errno* and information on the error position (*errpos*) are output.

Example of a detailed report of an XTI call

```
24:16.320 t_bind (fd  5, req 0x8054ac8, ret  0x0)
       req:        addr.maxlen(-)  addr.len(>)       addr.buf(>)
                   ___             16                0x8054d48
          0  00021800 00000000  00000000 00000000|               |
                    qlen (>) 5
        return: 0
```

11 XTI(POSIX) library functions

The XTI(POSIX) library functions are described in this chapter.

The first thing described is the format in which the separate XTI functions are described. The subsequent overview collects several XTI functions together into task-oriented groups. Finally, all XTI functions are described in alphabetic order.

11.1 Description format

The XTI functions are described in a uniform format which is structured as follows:

Function name - brief functional description

```
#include < ... >
Syntax of function
```

Description

Detailed description of the functionality and an explanation of the parameters.

Return value

List and description of all possible return values for the function.

Errors

List and description of the error codes which are stored in *t_errno* If an error occurs during the function call or processing.

Note

Explanation of terms or information about interaction with other functions or tips for use. This section may be missing.

See also

Cross references to the descriptions of other functions.

11.2 Overview of functions

In the following overview of the XTI library functions, several functions are collected together into task-oriented groups.

Connection setup and shutdown over transport endpoints

Function	Description	See
t_open()	Set up transport endpoint	page 256
t_close()	Close transport endpoint	page 237
t_bind()	Assign a transport endpoint an address	page 234
t_unbind()	Deactivate a transport endpoint	page 289
t_connect()	Initiate a connection over a transport endpoint (e.g. by a client)	page 238
t_rcvconnect()	Get the status of a previously sent connection request	page 268
t_listen()	Test a transport endpoint for pending connection requests (e.g. by a server)	page 252
t_accept()	Accept a connection over a transport endpoint (e.g. by a server)	page 229
t_rcvrel()	Confirm reception of a request for orderly connection shutdown	page 272
t_rcvdis()	Get the cause of a connection shutdown	page 270
t_sndrel()	Initiate orderly connection shutdown	page 282
t_snddis()	Refuse a connection request or initiate an immediate abort of an established connection	page 280

Transferring data between transport endpoints

Function	Description	See
t_rcv()	Receive data over a transport endpoint (connection-oriented)	page 266
t_rcvudata()	Receive datagrams over a transport endpoint (connectionless)	page 274
t_rcvuderr()	Receive error information about a sent datagram (connectionless)	page 276
t_snd()	Send data over a transport endpoint (connection-oriented)	page 278
t_sndudata()	Send datagrams over a transport endpoint (connectionless)	page 284

Getting information about transport endpoints

Function	Description	See
t_getinfo()	Get protocol-specific information	page 245
t_getstate()	Get the current state of the transport provider	page 250
t_getprotaddr()	Get protocol addresses	page 248
t_look()	Get the current event on the transport endpoint reported by the transport provider	page 254

Managing options of a transport endpoint

Function	Description	see
t_optmgmt()	Manage options of a transport endpoint	page 260

Using the transport library data structures

Function	Description	See
t_alloc()	Reserve memory dynamically for data structures declared in the <xti.h> transport library	page 232
t_free()	Release memory reserved for data structures declared in the <xti.h> transport library	page 243
t_sync()	Synchronize data structures of the <xti.h> transport library	page 287

Generating error messages

Function	Description	See
t_error()	Output error message to standard output	page 241
t_strerror()	Output error message text	page 286

11.3 Functions

The XTI(POSIX) library functions are described in alphabetic order in this section.

To be able to execute the XTI functions, the application must link in the X/Open-compliant <xti.h> header file. The <xti.h> file is copied into the */usr/include* directory when SOCKETS(POSIX) is installed (see also section “Header files” on page 9).

If an XTI function returns the TSYSERR error, the *errno* error variable is set. the values for *errno* are defined in <errno.h>.

t_accept() - accept connection

```
#include <xti.h>

int t_accept(int fd, int resfd, struct t_call *call);
```

Description

The transport user calls the *t_accept()* function to accept a connection over a transport endpoint, which another transport user requested with the *t_connect()* function.

The *fd* parameter designates the local transport endpoint on which a connection request arrived. The *resfd* parameter specifies the local transport endpoint over which the connection is to be set up.

Two cases must be discriminated with the *resfd* transport endpoint on which the connection is to be accepted:

– *resfd* == *fd*
  No further connection requests may be pending on *fd* in this case, i.e. the transport user must have already used *t_accept()* or *t_snddis()* to process all connection requests previously received on *fd*. Otherwise, *t_accept()* terminates with an error and sets *t_errno* to TINDOUT.

– *resfd* != *fd*
  In this case, *resfd* must be in the T_UNBND or T_IDLE state when *t_accept()* is called (see section “t_getstate() - get current state” on page 250).

The user calls the *call* parameter to pass information that the transport provider needs for setting the connection up. *call* is a pointer to an object of type *struct t_call*.

The *t_call* structure is declared in <xti.h> as follows:

```
struct t_call {
   struct netbuf addr;
   struct netbuf opt;
   struct netbuf udata;
   int sequence;
};
```

call->addr contains the protocol address of the transport user who sent the connection request.

call->opt shows all the options of the connection concerned. The values and syntax of these options are protocol-specific.
Sending user data (*call->udata* parameter) is not supported.
call->sequence contains the value previously returned by *t_listen()*, which uniquely identifies the connection request pending on transport endpoint *fd*.

If further events are pending on the transport endpoint passed by *fd* (connection request or connection shutdown request), *t_accept()* terminates with an error and sets *t_errno* to TLOOK.

Return value

0:
: If successful.

-1:
: If an error occurs. *t_ errno* is set to indicate the error.

Errors

TBADF
: The specified descriptor does not reference a transport endpoint.

TOUTSTATE
: *t_accept()* was called in the wrong position within a sequence of XTI function calls for transport endpoint *fd* or the transport endpoint passed by *resfd* is not in either the T_IDLE or T_UNBND state.

TACCES
: The user has no allowance to accept a connection on the replying transport endpoint or use the specified options.

TBADDATA
: Sending user data is not supported.

TBADOPT
: The specified options had the wrong format or contained invalid information.

TBADSEQ
: An invalid sequence number was specified.

TINDOUT
: The function was called with *fd* == *resfd* and further connection requests are pending for the transport endpoint passed by *fd*. These previously received connection requests must first be processed with *t_accept()* or *t_snddis()*.

TLOOK
An asynchronous event arrived on the transport endpoint passed by *fd* and this must be processed immediately.

TNOTSUPPORT
The function is not supported by the underlying transport service.

TRESQLEN
The transport endpoint passed by *resfd* (with *resfd* != *fd*) is assigned a protocol address for which *qlen* > 0 applies.

TSYSERR
A system error occurred during execution of this function.

See also

t_connect(), t_getstate(), t_listen(), t_open(), t_rcvconnect()

t_alloc() - reserve memory for library structure

```
#include <xti.h>

char *t_alloc(int fd, int struct_type, int fields);
```

Description

The transport user calls the *t_alloc()* function to reserve memory dynamically for various types of structures. *t_alloc()* returns a pointer to the reserved structure object. Every structure object created with *t_alloc()* can be passed as a current parameter when specific XTI functions are called.

The user must specify the transport endpoint over which the structure object created with *t_alloc()* is passed when an XTI function is called (e.g. *t_bind()*), as the current parameter for *fd*. This allows *t_alloc()* to access the relevant size information. The size of the buffer that is created results from the same information that the user receives with *t_open()* and *t_getinfo()* for the transport endpoint concerned.

The *struct_type* parameter specifies the structure type. *t_alloc()* then reserves memory for the structure and for buffers to which this structure refers.
The user can specify the following values for *struct_type* when calling *t_alloc()*:

- T_BIND (for *struct t_bind*)
- T_CALL (for *struct t_call*)
- T_OPTMGMT (for *struct t_optmgmt*)
- T_DIS (for *struct t_discon*)
- T_UNITDATA (for *struct t_unitdata*)
- T_UDERROR (for *struct t_uderr*)
- T_INFO (for *struct t_info*)

Apart from *t_info*, all the above structures contain at least one component of type *struct netbuf*.

The *netbuf* structure is declared in <xti.h> as follows:

```
struct netbuf {
   unsigned int maxlen;
   unsigned int len;
   char *buf;
};
```

The user sets the *fields* parameter to specify whether memory is also to be reserved for the buffer for each *netbuf* structure in the structure specified by *struct_type*. *fields* is formed by inclusive ORing of the bits in any combination of the values described below:

- T_ADDR: *addr* component of the *t_bind*, *t_call*, *t_unitdata* or *t_uderr* structures
- T_OPT: *opt* component of the *t_optmgmt*, *t_call*, *t_unitdata* or *t_uderr* structures
- T_UDATA: *udata* component of the *t_call*, *t_discon* or *t_unitdata* structures
- T_ALL: all relevant components of the structure specified by *struct_type*

t_alloc() reserves memory for the buffer assigned to each *netbuf* structure specified by the *fields* parameter. *t_alloc()* also correspondingly initializes the *buf* pointer and the value of *maxlen* in the separate *netbuf* structures.

If the value of *maxlen* in any of the *netbuf* structures specified by *fields* has the value -1 or -2 (see *t_open()* or *t_getinfo()*), *t_alloc()* cannot determine the size of the buffer and terminates with an error. *t_errno* is set to TSYSERR and *errno* to EINVAL. For each *netbuf* structure not specified in *fields*, *buf* is set to NULL and *maxlen* to 0.

Return value

If execution was successful, *t_alloc()* returns a pointer to the newly created structure. In an error occurs, the null pointer is returned and *t_ errno* is set to indicate the error.

Errors

TBADF
: The specified file descriptor does not reference a transport endpoint.

TSYSERR
: A system error occurred during execution of this function.

See also

t_free(), t_getinfo(), t_open()

t_bind() - assign a transport endpoint an address

```
#include <xti.h>

int t_bind(int fd, struct t_bind *req, struct t_bind *ret);
```

Description

The user calls the *t_bind()* function to assign the transport endpoint specified by the *fd* parameter a protocol address and activates the transport endpoint.

After successful execution of *t_bind()*, the user has the following options:

– The user can call *t_listen()* in connection-oriented mode to check the transport endpoint specified by *fd* for pending connection requests and then, if necessary, use *t_accept()* to accept connections on *fd*. The user can also send connection requests to other transport endpoints over transport endpoint *fd* with *t_connect()*.

– In connectionless mode, the user can send or receive datagrams over the transport endpoint specified by *fd*.

The *req* and *ret* parameters each point to an object of type *struct t_bind*.

The *t_bind* structure is declared in <xti.h> as follows:

```
struct t_bind {
   struct netbuf addr;
   unsigned qlen;
};
```

The user specifies the protocol address to be assigned to the transport endpoint in *req->addr*. The user specifies the length of this address in bytes in *req->addr.len*. *req->addr.buf* points to the address buffer. *req->addr.maxlen* is meaningless.

The transport user passes a pointer to a buffer in *ret->addr.buf* and specifies the maximum length of this buffer in *ret->addr.maxlen*. After successful execution, *t_bind()* returns the address assigned to transport endpoint *fd* in *ret-> addr.buf*. *t_bind()* returns the actual length of this address in *ret->addr.len*.

t_bind() returns the TBUFOVFLW error code if the length specified in *ret->addr.maxlen* is too small for storing the address returned by *t_bind()*.
However, the state of the transport endpoint changes to T_IDLE.

req->qlen and *ret->qlen* are only significant if *fd* is run in connection-oriented mode, in which case they define the maximum number of pending connection requests that the transport provider supports for transport endpoint *fd*. A pending connection request is a connection request which was passed to the endpoint of the user by the transport system and has to date neither been accepted (*t_accept()*) nor refused (*t_snddis()*) by this user.
The number of connection requests for transport endpoint *fd* supported by the transport provider is calculated as follows:

– Before calling *t_bind()*, the user specifies in *req->qlen* the number of pending connection requests that the transport provider is to support on transport endpoint *fd*. *req->qlen* > 0 is only meaningful with a transport endpoint that the user later monitors passively for pending connection requests with *t_listen()*.

– *req->qlen* is evaluated by the transport provider. If the transport provider cannot support the number of pending connection requests specified in *req->qlen*, he reduces the value passed in *req->qlen* appropriately. However, the transport provider never reduces a *req->qlen* value that is > 0 to 0. The transport provider can currently support a maximum of 8 pending connection requests.

– *t_bind()* returns the number of pending connection requests that the transport provider actually supports for transport endpoint *fd* in *ret->qlen*.

If the user does not want to specify the address to be bound (assigned) to transport endpoint *fd*, he passes the null pointer as the current parameter for *req*. In this case, the transport provider selects the address to be bound, whereby he implicitly assumes a value of 0 for *req->qlen*.
The user can also pass the null pointer as the current parameter for *ret*, if he is indifferent to the value of *qlen* and the address bound to *fd* with *t_bind()* by the transport provider.
It is permissible to pass the null pointer for both *req* and *ret* in the same *t_bind()* call. The transport provider then selects the address which is bound to *fd*. However, *t_bind()* does not return this information to the user.

Return value

0:
If successful.

-1:
If an error occurs. *t_ errno* is set to indicate the error.

Errors

TACCES
The user has no allowance to use the specified address.

TADDRBUSY
The specified protocol address is already in use.

TBADADDR
The specified protocol address has the wrong format or contains invalid information.

TBADF
The specified file descriptor does not reference a transport endpoint.

TBUFOVFLW
The allowed number of bytes for a result parameter is too small to store the value of the parameter. The state of the transport provider is changed to T_IDLE and the information to be returned in **ret* is deleted.

TNOADDR
The transport provider could not reserve an address (see also section “Dependencies of the BS2000/OSD BCAM transport system” on page 302).

TOUTSTATE
The function was called in the wrong position within a sequence of XTI function calls for transport endpoint *fd*.

TSYSERR
A system error occurred during execution of this function.

See also

t_open(), t_optmgmt(), t_unbind()

t_close() - close transport endpoint

```
#include <xti.h>

int t_close(int fd);
```

Description

The user calls the *t_close()* function to inform the transport provider that he no longer needs the transport endpoint specified by *fd*. *t_close()* releases all local library resources reserved for *fd*.

t_close() should be called in the T_UNBND state (see section “t_getstate() - get current state” on page 250). As *t_close()* does not check any state information, it can also be called in all other states to close a transport endpoint.

If there are no further descriptors for transport endpoint *fd* in the calling process or any other process, the transport endpoint is shut down completely, i.e. the system resources are released. Established connections are aborted and any data not already sent or not fetched by the receiver is lost.

Return value

0:
: If successful.

-1:
: If an error occurs. *t_ errno* is set to indicate the error.

Errors

TBADF
: The specified file descriptor does not reference a transport endpoint.

See also

t_getstate(), t_open(), t_unbind()

t_connect() - request connection

```
#include <xti.h>

int t_connect(int fd, struct t_call *sndcall, struct t_call *rcvcall);
```

Description

The user calls the *t_connect()* function to send a connection request over local transport endpoint *fd* to another transport user who is specified by the protocol address passed with the *sndcall* parameter.

the *sndcall* and *rcvcall* parameters each point to an object of type *struct t_call*.

The *t_call* structure is declared in <xti.h> as follows:

```
struct t_call {
   struct netbuf addr;
   struct netbuf opt;
   struct netbuf udata;
   int sequence;
};
```

the caller of *t_connect()* passes information in *sndcall* that the transport provider needs to send a connection request:

– *sndcall->addr* contains the protocol address of the transport endpoint to which the connection request is to be sent.

– *sndcall->opt* contains protocol-specific information which the transport provider needs. However, *sndcall->opt* does **not** specify the structure of the options as the transport provider himself defines the structure of all options passed to him. These options are specific to the underlying protocol of the transport provider.
If the user passes the value 0 in *sndcall->opt.len*, the transport provider selects default options and the user does not have to negotiate them with the transport provider.

Since sending user data is not supported, *sndcall->udata* is meaningless for *t_connect()*. *sndcall->sequence* is also meaningless for *t_connect()*.

After successful execution, *t_connect()* returns information in *rcvcall* about the connection that was just set up.

- *rcvcall->addr* contains the protocol address of the transport endpoint which accepted the connection request with *t_accept()*.
  Before calling *t_connect()* the user must make the maximum length of the result buffer (*rcval->addr.buf*) known in *rcval->addr.maxlen*.
- *rcval->opt* contains protocol-specific information concerning the newly set up connection.
  Before calling *t_connect()* the user must make the maximum length of the result buffer (*rcval->opt.buf*) known in *rcval->opt.maxlen*.

Since receiving user data is not supported, *rcvcall->udata* is meaningless for *t_connect()*. *rcvcall->sequence* is also meaningless for *t_connect()*.

By default, *t_connect()* works in synchronous mode and waits (blocks) until a reply arrives from the destination user, i.e. transport user, to which the connection request was sent. *t_connect()* only relinquishes control back to the calling transport user after receiving the reply. Successful execution of *t_connect()* (return value 0) indicates that the requested connection has been set up.

However, *t_connect()* is executed in asynchronous mode if *t_open()* or the POSIX *fcntl()* function was used previously to set O_NDELAY or O_NONBLOCK for the transport endpoint specified by *fd*. In asynchronous mode, *t_connect()* does not wait for the reply from the destination user, but relinquishes control back to the calling user immediately after getting the status of the connection request. If the requested connection has not been set up yet, *t_connect()* returns the value -1 and sets *t_errno* to TNODATA.
In other words, *t_connect()* initiates the connection setup in asynchronous mode simply by sending a connection request to the destination user. The local user can get the status of the requested connection with the *t_rcvconnect()* function.

Return value

0:
: If successful.

-1:
: If an error occurs. *t_ errno* is set to indicate the error.

Errors

TACCES
The user has no allowance to use the specified address or options.

TBADADDR
The specified protocol address has the wrong format or contains invalid information.

TBADDATA
Sending user data is not supported.

TBADF
The specified file descriptor does not reference a transport endpoint.

TBADOPT
The specified protocol options had the wrong format or contained invalid information.

TBUFOVFLW
The number of bytes that were reserved for a result parameter are not enough to store the parameter value. If synchronous mode is being used, the state of the transport provider from the user viewpoint is set to T_DATAXFER and the information for the connection request, which should be returned in **rcvcall*, is removed.

TLOOK
An asynchronous event occurred on the transport endpoint passed in *fd* and this must be processed immediately.

TNODATA
O_NDELAY or O_NONBLOCK was set so that the function could initiate connection setup procedure successfully but does not wait for a reply from the remote user.

TNOTSUPPORT
This function is not supported by the underlying transport provider.

TOUTSTATE
The function was called in the wrong position within a sequence of XTI function calls for transport endpoint *fd*.

TSYSERR
A system error occurred during execution of this function.

See also

t_accept(), t_getinfo(), t_listen(), t_open(), t_optmgmt(), t_rcvconnect(), fcntl()

t_error() - output error message to the standard output

```
#include <xti.h>

int t_error(char *errmsg);
extern int t_errno;
extern char *t_errlist[];
extern int t_nerr;
```

Description

The user calls the *t_error()* function to write a self-formulated message describing the last error which occurred with an XTI function call, to the standard error output. his message, which describes the error in context, is passed in the *errmsg* parameter.

t_errlist is a vector of messages, which are each represented as a character string and allow user messages to be formatted. *t_errno* can be used as an index for this vector to receive a specific error message in string format (without end-of-line termination). *t_nerr* is the maximum index value for the *t_errlist* vector.
t_errno is set if an error occurs, however, it is not deleted for subsequent, successful calls.

The *t_error()* output comprises the error message passed by the user, followed by a colon (:) and the standard error output of the XTI function for the current value in *t_errno*. If *t_errno* has the value TSYSERR, *t_error()* also outputs the default error message for the current value in *errno*.

Return value

Always 0

Errors

No error codes are defined for *t_error()*.

Example

If the *t_connect()* function on transport endpoint *fd2* terminates with an error because an invalid address was specified, the error can follow the call below:

```
t_error("t_connect failed");
```

The following message is output:

```
t_connect failed: incorrect addr format
```

"t_connect failed" tells the user which function failed. "incorrect addr format" indicates the actual error which occurred.

t_free() - release library structure memory

```
#include <xti.h>

int t_free(char *ptr, int struct_type);
```

Description

The user calls the *t_free()* function to release memory which was previously assigned with the *t_alloc()* function. *t_free()* releases the memory for the object of type *struct_type* to which pointer *ptr* points.

struct_type specifies one of the six structure types described for *t_alloc()*:

- T_BIND (for *struct t_bind*)
- T_CALL (for *struct t_call*)
- T_OPTMGMT (for *struct t_optmgmt*)
- T_DIS (for *struct t_discon*)
- T_UNITDATA (for *struct t_unitdata*)
- T_UDERROR (for *struct t_uderr*)
- T_INFO (for *struct t_info*)

The *t_free()* function checks the *ptr->addr*, *ptr->opt* and *ptr->udata* components of type *struct netbuf* in the *struct_type* **ptr* object and releases the buffer to which the *buf* component of the separate *netbuf* structures point. If a *buf* pointer is the null pointer, *t_free()* does not try to release the memory involved. As soon as all *buf* buffers are released, *t_free()* releases the structure to which *ptr* points.

t_free() produces undefined results if *ptr* or any *buf* pointer points to a memory area which was not previously reserved with *t_alloc()*.

Return value

0:
    If successful.

-1:
    If an error occurs. *t_ errno* is set to indicate the error.

Errors

TSYSERR
A system error occurred during execution of this function.

See also

t_alloc()

t_getinfo() - get protocol-specific information

```
#include <xti.h>

int t_getinfo(int fd, struct t_info *info);
```

Description

The *t_getinfo()* function supplies the user with information about the current characteristics of the underlying transport protocol bound to transport endpoint (file descriptor) *fd*. *t_getinfo()* returns the same information in the *t_info* structure, to which the *info* parameter points, that was returned by *t_open()* when transport endpoint *fd* was set up. This allows the user to access the information supplied by *t_open()* at any time with *t_getinfo()*.

The *t_info* structure, to which the *info* parameter points, is declared in <xti.h> as follows:

```
struct t_info {
long addr;     /* Maximum length of the transport protocol address */
long options;  /* Maximum number of bytes of the protocol-specific options */
long tsdu;     /* Maximum size of a data packet (TSDU) */
long etsdu;    /* Maximum size of a packet for expedited data (ETSDU) */
long connect;  /* Maximum amount of data allowed for connection setup
                  function */
long discon;   /* Maximum amount of data allowed for the t_snddis() and
                  t_rcvdis() functions */
long servtype; /* Service type offered by the transport provider */
long flags;    /* Other transport provider information /*
};
```

The values of the *t_info* components have the following meaning:

addr

A value ≥ 0 defines the maximum length of a transport protocol address. The value -1 indicates that the address length is unlimited. The value -2 indicates that the transport provider does not support user accesses to the transport protocol address.

options

A value ≥ 0 defines the maximum length in bytes that the transport provider supports for protocol-specific options. The value -1 indicates that the length of the options is unlimited. The value -2 indicates that the transport provider does not support options that can be influenced by the user.

tsdu
A value > 0 defines the maximum length of a transport service data unit (TSDU). The value 0 indicates that the transport provider does not support the TSDU concept although he does offer sending a data stream over the connection without maintaining logical block limits. The value -1 indicates that the length of a TSDU is unlimited. The value -2 indicates that the transport provider does not support transferring normal data.

etsdu
A value > 0 defines the maximum length of an expedited transport service data unit (ETSDU). The value 0 indicates that the transport provider does not support the ETSDU concept although he does offer sending a data stream over the connection without maintaining logical block limits. The value -1 indicates that the length of an ETSDU is unlimited. The value -2 indicates that the transport provider does not support transferring expedited data.

connect
A value ≥ 0 defines the maximum amount of data that can be sent with connection setup functions. The value -1 indicates that the amount of data that can be sent during connection setup is unlimited. The value -2 indicates that the transport provider does not support sending data with connection setup functions.

discon
A value ≥ 0 defines the maximum amount of data that can be sent with the *t_snddis()* and *t_rcvdis()* functions. The value -1 indicates that the amount of data that can be sent with connection shutdown functions is unlimited. The value -2 indicates that the transport provider does not support sending data with connection shutdown functions.

servtype
This component specifies the service type supported by the transport provider (see following page).

flags
This field specifies other transport provider information (no information is currently supplied).

If the transport service user wishes to be independent of protocols, he can use the above values to determine the size of buffers required for storing the separate pieces of information. The user can also alternatively call the *t_alloc()* function to reserve memory for these buffers. An error occurs if a user exceeds the permissible limits with an XTI function call.
The values stored in the separate *t_info* components can be changed as a result of option negotiation (with *t_optmgmt()*). The user can get information on the current characteristics with the *t_getinfo()* function.

After *t_getinfo()* is executed, the *info->servtype* component contains one of the following values:

T_COTS_ORD
: The transport provider supports a connection-oriented service with an optional orderly connection shutdown. *t_getinfo()* returns the value -2 for *etsdu*, *connect* and *discon* for this service type.

T_CLTS
: The transport provider supports a connectionless service. *t_getinfo()* returns the value -2 for *etsdu*, *connect* and *discon* for this service type.

Return value

0:
: If successful.

-1:
: If an error occurs. *t_ errno* is set to indicate the error.

Errors

TBADF
: The specified file descriptor does not reference a transport endpoint.

TSYSERR
: A system error occurred during execution of this function.

See also

t_open()

t_getprotaddr() - get protocol addresses

```
#include <xti.h>

int t_getprotaddr(int fd, struct t_bind *boundaddr, struct t_bind *peeraddr);
```

Description

The *getprotaddr()* returns the local and remote protocol addresses currently assigned to transport endpoint *fd*. The *boundaddr* and *peeraddr* parameters point to objects of type *struct t_bind*.

The *t_bind* structure is declared in <xti.h> as follows:

```
struct t_bind {
   struct netbuf addr;
   unsigned qlen;
};
```

Before calling *t_getprotaddr()*, the user specifies the maximum size of the address buffer in *boundaddr->maxlen* and *peeraddr->maxlen*. The user also specifies with *boundaddr->addr.buf* and *peeraddr->addr.buf* pointers to buffers into which *t_getprotaddr()* is to return the address concerned.

After *t_getprotaddr()* is executed, *boundaddr->addr.buf* points to the address assigned to transport endpoint *fd* (if available). *boundaddr->addr.len* contains the length of this address. If transport endpoint *fd* is in the T_UNBND state, *t_getprotaddr()* returns the value 0 in the *boundaddr->addr.len* component.
After *t_getprotaddr()* is executed, *peeraddr->addr.buf* points to the address of the communications partner of *fd* (if available).
peeraddr->addr.len contains the length of this address. If transport endpoint *fd* is not in the T_DATAXFER state, *t_getprotaddr()* returns the value 0 in the *peeraddr->addr.len* component.

Return value

0:
  If successful.

-1:
  If an error occurs. *t_ errno* is set to indicate the error.

Errors

TBADF
The specified file descriptor does not reference a transport endpoint.

TBUFOVFLW
The number of bytes reserved for a result parameter (with *maxlen*) is greater than 0 but not large enough to store the value of the parameter concerned.

TPROTO
This error indicates that a communications problem was detected between XTI and the transport system, for which no other suitable error description is available.

TSYSERR
A system error occurred during execution of this function.

See also

t_bind()

t_getstate() - get current state

```
#include <xti.h>

int t_getstate(int fd);
```

Description

The *t_getstate()* function returns the current state of the transport endpoint.

Return value

If successful, the current state of the transport endpoint is returned.
If an error occurs, *t_getstate()* returns the value -1.

The current state of the transport endpoint can assume the following values:

T_UNBND
: The transport endpoint is not bound to the transport service.

T_IDLE
: The transport endpoint is bound to the transport system.

T_OUTCON
: A sent connection request has not been processed yet.

T_INCON
: An incoming connection request has not been processed yet.

T_DATAXFER
: Data transfer phase

T_OUTREL
: A request for orderly connection shutdown was sent (wait for indication of an orderly connection shutdown)

T_INREL
: Wait for a request for orderly connection shutdown.

If the transport provider is in a state transition at exactly the time of the *t_getstate()* call, *t_getstate()* terminates with an error.

Errors

TBADF
The specified file descriptor does not reference a transport endpoint.

TSTATECHNG
The transport provider is currently changing state.

TSYSERR
A system error occurred during execution of this function.

See also

t_open()

t_listen() - wait for connection requests

```
#include <xti.h>

int t_listen(int fd, struct t_call *call);
```

Description

The user calls the *t_listen()* function to monitor transport endpoint *fd* passively for connection requests which other transport endpoints send to *fd* with *t_connect()*. After execution of *t_listen()*, the *call* parameter points an object of type *struct t_call* which contains information about incoming connection requests.

The *t_call* structure is declared in <xti.h> as follows:

```
struct t_call {
   struct netbuf addr;
   struct netbuf opt;
   struct netbuf udata;
   int sequence;
};
```

t_listen() returns the protocol address of the transport service user who sent the connection request, in *call->addr.buf*. Before calling *t_listen()*, the user must specify the maximum size of the *call->addr.buf* result buffer in *call->addr.maxlen*.
Returning protocol-specific parameters in *call->opt* and user data in *call->udata* are not supported.
After execution of *t_listen()*, the value of *call->sequence* uniquely identifies the connection request which arrived, allowing the user to monitor several connection requests before replying to one of them.

By default, *t_listen()* works in synchronous mode, waits (blocks) if no connection requests are available and only returns control to the user after a connection request arrives. However, if the user previously set O_NDELAY or O_NONBLOCK with *t_open()* or the POSIX *fcntl()* function, *t_listen()* works in asynchronous mode. *t_listen()* then only polls for pending connection requests (*poll()*) and does not wait. If no connection requests are available, *t_listen()* returns the value -1 and sets *t_errno* to TNODATA.

Return value

0:
If successful.

-1:
If an error occurs. *t_ errno* is set to indicate the error.

Errors

TBADF
The specified file descriptor does not reference a transport endpoint.

TBADQLEN
The *qlen* value of the transport endpoint to which *fd* refers is 0.

TBUFOVFLW
The number of bytes reserved (with *maxlen*) for a result parameter is not enough to store the value of the parameter. The state of the transport provider changes to T_INCON from the viewpoint of the user. The information about the connection request which is to be returned in **call*, is deleted.

TLOOK
An asynchronous event occurred on the transport endpoint passed in *fd* and this must be processed immediately.

TNODATA
O_NDELAY or O_NONBLOCK is set but there are no connection requests in the queue.

TNOTSUPPORT
This function is not supported by the underlying transport provider.

TPROTO
The connection to the BCAM transport system has been shut down.

TSYSERR
A system error occurred during execution of this function.

See also

t_accept(), t_bind(), t_connect(), t_open(), t_rcvconnect(), fcntl()

t_look() - get current event

```
#include <xti.h>

int t_look(int fd);
```

Description

The user calls the *t_look()* function to get the current event on the transport endpoint specified by the *fd* parameter.
t_look() allows the transport provider to report an asynchronous event to the user if the user executes functions in synchronous mode. Some events must be reported immediately to the user and are indicated by a special error code (TLOOK) when the current or next function is executed.
The user can call *t_look()* to periodically poll a transport endpoint for asynchronous events (*poll()*).

Return value

If execution is successful, *t_look()* returns a value which indicates the event that occurred. *t_look()* returns the value 0 if no events occurred.
If an error occurs, -1 is returned and *t_errno* is set to indicate the error.

The following events can be returned by *t_look()*:

T_LISTEN
: Indication of a connection was received.

T_CONNECT
: Confirmation of a connection was received.

T_DATA
: Data was received.

T_DISCONNECT
: Indication of a connection shutdown was received.

T_UDERR
: Indication of a datagram error was received.

T_ORDREL
: Indication of an orderly connection shutdown was received.

Errors

TBADF
: The specified file descriptor does not reference a transport endpoint.

TSYSERR
: A system error occurred during execution of this function.

See also

t_open()

t_open() - set up a transport endpoint

```
#include <xti.h>
#include <fcntl.h>

int t_open(char *path, int oflag, struct t_info *info);
```

Description

The user calls the *t_open()* function to set up a transport endpoint by opening a file in a UNIX system which identifies a particular transport provider (i.e. the transport protocol). The *t_open()* call is the first step in initializing a transport endpoint.
t_open() returns a file descriptor to a transport endpoint of this type.

The following are supported, based on the TCP/IP protocol:

– */dev/tcp* for opening a connection-oriented transport endpoint
– */dev/udp* for opening a connectionless transport endpoint

The user passes a pointer to the path name of the file to be opened, with the *path* parameter. *oflag* can be formed by inclusive bit ORing of O_NDELAY or O_NONBLOCK with O_RDWR. These options are declared in the <fcntl.h> header file.

The transport endpoint set up with *t_open()* is identified in subsequent XTI function calls by the file descriptor returned by *t_open()*.

The *info* parameter points to an object of type *struct t_info* in which *t_open()* returns the characteristics of the underlying transport protocol.
t_open() does not return any protocol information if the null pointer is passed as the current parameter for *info* when *t_open()* is called.

The *t_info* structure is declared in <xti.h> as follows:

```
struct t_info {
long addr;     /* Maximum length of the transport protocol address */
long options;  /* Maximum number of bytes of the protocol specification
                  options */
long tsdu;     /* Maximum size of a data packet (TSDU) */
long etsdu;    /* Maximum size of an expedited data packet (ETSDU) */
long connect;  /* Maximum allowed amount of data for connection setup
                  functions */
long discon;   /* Maximum allowed amount of data for the t_snddis() and
                  t_rcvdis() functions */
long servtype; /* Service type offered by the transport provider */
long flags;    /* Other transport provider information /*
};
```

The values of the *t_info* components have the following meaning:

addr
A value ≥ 0 defines the maximum length of a transport protocol address. The value -1 indicates that the address length is unlimited. The value -2 indicates that the transport provider does not support user accesses to the transport protocol address.

options
A value ≥ 0 defines the maximum length in bytes that the transport provider supports for protocol-specific options. The value -1 indicates that the length of the options is unlimited. The value -2 indicates that the transport provider does not support options that can be influenced by the user.

tsdu
A value > 0 defines the maximum length of a transport service data unit (TSDU). The value 0 indicates that the transport provider does not support the TSDU concept although he does offer sending a data stream over the connection without maintaining logical block limits. The value -1 indicates that the length of a TSDU is unlimited. The value -2 indicates that the transport provider does not support transferring normal data.

etsdu
A value > 0 defines the maximum length of an expedited transport service data unit (ETSDU). The value 0 indicates that the transport provider does not support the ETSDU concept although he does offer sending a data stream over the connection without maintaining logical block limits. The value -1 indicates that the length of an ETSDU is unlimited. The value -2 indicates that the transport provider does not support transferring expedited data.

connect
A value ≥ 0 defines the maximum amount of data that can be sent with connection setup functions. The value -1 indicates that the amount of data that can be sent during connection setup is unlimited. The value -2 indicates that the transport provider does not support sending data with connection setup functions.

discon
A value ≥ 0 defines the maximum amount of data that can be sent with the *t_snddis()* and *t_rcvdis()* functions. The value -1 indicates that the amount of data that can be sent with connection shutdown functions is unlimited. The value -2 indicates that the transport provider does not support sending data with connection shutdown functions.

servtype
This component specifies the service type supported by the transport provider (see below).

flags
This field specifies other transport provider information (no information is currently supplied).

If the transport service user wishes to be independent of protocols, he can use the above values to determine the size of buffers required for storing the separate pieces of information. The user can also alternatively call the *t_alloc()* function to reserve memory for these buffers. An error occurs if a user exceeds the permissible limits with an XTI function call.

The *info->servtype* component contains one of the following values after *t_open()* is executed:

T_COTS_ORD
The transport provider supports a connection-oriented service with an optional orderly connection shutdown. *t_open()* returns the value -2 for *etsdu*, *connect* and *discon* for this service type.

T_CLTS
The transport provider supports a connectionless service. *t_open()* returns the value -2 for *etsdu*, *connect* and *discon* for this service type.

A transport endpoint can only support one of the above services at any one time.

Return value

If successful, *t_open()* returns a valid file descriptor.
If an error occurs, -1 is returned and *t_errno* is set to indicate the error.

Errors

TSYSERR
A system error occurred during execution of this function.

TBADFLAG
An invalid option was specified.

TBADNAME
The name specified in *path* is invalid.

TPROTO
A connection could not be set up to the transport system.

See also

open()

t_optmgmt() - manage transport endpoint options

```
#include <xti.h>

int t_optmgmt(int fd, struct t_optmgmt *req, struct t_optmgmt *ret);
```

Description

A transport service user can call the *t_optmgmt()* function to get, verify or negotiate protocol options with the transport provider.

The *fd* parameter specifies a transport endpoint. The *req* and *ret* parameters each point to an object of type *struct t_optmgmt*.

The *t_optmgmt* structure is declared in <xti.h> as follows:

```
struct t_optmgmt {
   struct netbuf opt;
   long flags;
};
```

The *opt* component specifies the protocol options. The *flags* component specifies the action to be executed with these options. The options are represented by a *netbuf* structure, similarly to the addresses with *t_bind()*.

The transport user employs the *req* parameter to request a specific transport provider with *t_optmgmt()* and to send options to the transport provider. The user specifies the length of the buffer (in bytes) in which the options are passed to the transport provider, in *req->opt.len*. *req->opt.buf* points to this buffer. *req->opt.maxlen* is meaningless.

Each option is stored in the option buffer *req->opt.buf* as a *t_opthdr* structure and must be arranged within the buffer on word boundaries. If the user specifies several options, they must all belong to the same protocol level.

The *t_opthdr* structure is declared in <xti.h> as follows:

```
struct t_opthdr {
   unsigned long len;    /* Defines the total length of the option and */
                         /* is calculated from the sum of the length   */
                         /* of the t_opthdr structure and the length   */
                         /* of a possible subsequent option value      */
   unsigned long level;  /* Specifies the protocol level               */
   unsigned long name;   /* Specifies the option                       */
   unsigned long status; /* Supplies information as to whether this    */
                         /* option could be set/reset                  */
};
```

The user can read from the option buffer and write to it with the OPT_NEXTHDR (*pbuf*, *buflen*, *poption*) macro which is defined in <xti.h>. The *pbuf* parameter, is a pointer to the start of the option buffer, *buflen* defines the length of the option buffer and *poption* is a pointer to the current option within the buffer. The OPT_NEXTHDR macro returns a pointer to the next option or the null pointer if the end of the buffer has been reached.

Before calling *t_optmgmt()*, the user must specify in *ret->opt.maxlen* the maximum length (in bytes) of the result buffer in which *t_optmgmt()* returns this information. *req->opt.buf* points to this buffer. After *t_optmgmt()* is executed, *ret->opt.len* contains the actual length of the returned options and flag values.

The user must specify one of the following actions in *req->flags*:

T_NEGOTIATE
: The user employs this action to negotiate the values of the options specified in *req->opt.buf* with the transport provider. The transport provider returns the negotiated values in the result buffer *ret->opt.buf*.

  The status field of the relevant *t_opthdr* structure shows the result of the operation for each option.
  The status field can assume the following values:

  - T_SUCCESS, if the option could be successfully changed.
  - T_PARTSUCCESS, if an option value lower than the specified one could be set.
  - T_FAILURE, if the change could not be carried out.
  - T_READONLY, if the option can only be read and not changed.
  - T_NOTSUPPORT, if the transport provider does not support the option.

t_optmgmt() returns the lowest common result of all option-specific results in *ret->flags*. T_NOTSUPPORT has the highest validity and the validity of the results decreases in the order T_NOTSUPPORT, T_READONLY, T_FAILURE, T_PARTSUCCESS, T_SUCCESS.

In each protocol level, the user can reset all options supported in the level concerned to its original values with the T_ALLOPT option. *t_optmgmt()* then returns all options with their values in the result buffer *ret->opt.buf*.

T_CHECK
The user can employ this action to check whether the transport provider supports the options specified in *req->opt.buf*.

If an option is specified without a value, *t_optmgmt()* only sets the status field in the result buffer *ret->opt.buf* for this option.
The status field contains one of the following values:

- T_SUCCESS, if the transport provider supports the option.
- T_NOTSUPPORT, if the transport provider does not support the option.
- T_READONLY, if the option is read-only.

If an option is specified with a value, *t_optmgmt()* returns the result in the manner described above under “T_NEGOTIATE”. *t_optmgmt()* then returns the lowest common result of all option-specific results in *ret->flags*.

T_CURRENT
The user can employ this action to get the values of all options specified in *req->opt.buf*. After *t_optmgmt()* is executed, *ret->opt.buf* contains the options and their current values. *ret->opt.flags* indicates the result:

- T_SUCCESS, if the transport provider supports the option.
- T_NOTSUPPORT, if the transport provider does not support the option.
- T_READONLY, if the option is read-only.

On each protocol level, the user can employ the T_ALLOPT option to direct *t_optmgmt()* to return all supported options with their values.

T_DEFAULT

This action allows the user to have the default values of the options specified in **req* returned in **ret*.
The status field of an option in *ret->opt.buf* then has the following value:

- T_SUCCESS, if the transport provider supports the option.
- T_NOTSUPPORT, if the transport provider does not support the option.
- T_READONLY, if the option is read-only.

t_optmgmt() then returns the lowest common result of all option-specific results in *ret->flags*.

On each protocol level, the user can employ the T_ALLOPT option to return all options supported in the level concerned with their original values.

Protocol levels and options

The options are distributed over different protocol levels.

Table 15 provides an overview of protocol levels and options:

Protocol level	Option name	Type of option value	Option values
XTI_GENERIC	XTI_DEBUG	unsigned long	0 or XTI_GENERIC
INET_TCP	TCP_KEEPALIVE	struct t_kpalive	(see text)
	TCP_NODELAY	unsigned long	T_YES or T_NO
INET_IP	IP_BROADCAST	unsigned int	T_YES or T_NO

Table 15: Protocol levels and options

XTI_GENERIC protocol level options

XTI_DEBUG

The user can define whether diagnostic information is to be generated with this option.

- XTI_GENERIC specified as an option value: diagnostic information is generated.
- No option value specified: no diagnostic information is generated.

See also chapter “XTI trace” on page 217 for details of diagnostic information.

INET_TCP protocol level options

TCP_KEEPALIVE

Setting this option activates a mechanism which periodically tests that a connection is still established. The option value is stored in an object of type *struct t_kpalive*.

The *t_kpalive* structure is declared in <xti.h> as follows:

```
struct t_kpalive {
   long kp_onoff;     /* Enable/disable option */
   long kp_timeout;   /* Keep-alive timeout in minutes */
};
```

The “sign-of-life monitoring” for connections can be enabled or disabled with the *kp_onoff* parameter, which can assume the value T_YES or T_NO.
The *kp_timeout* parameter is meaningless as the transport system defines the time intervals for connection monitoring itself.

TCP_NODELAY

This option allows the user to influence the time response when sending data.
Data is sent immediately by default. However, if delays occur when sending separate pieces of data, the transport system collects the small amounts of data and sends them together at a later time. This reduces the load on the network. If the TCP_NODELAY option is set (option value T_YES) this mechanism is ineffective, i.e. the data is sent immediately.

INET_IP protocol level options

IP_BROADCAST

The user controls the sending of broadcast messages with this option.
Since broadcast messages can always be sent in BS2000/OSD, the IP_BROADCAST option has not functional significance. However, it must be noted that the reception of broadcast messages may be unauthorized.

Return value

0:

If successful.

-1:

If an error occurs. *t_ errno* is set to indicate the error.

Errors

TACCES
: The user has no rights to negotiate the specified options.

TBADF
: The specified file descriptor does not reference a transport endpoint.

TBADFLAG
: An invalid flag was specified.

TBADOPT
: The specified protocol options either had the wrong format or contained invalid information.

TBUFOVFLW
: The number of bytes reserved for a result parameter with *maxlen* are not enough to store the parameter value. The information to be returned in **ret* is deleted.

TOUTSTATE
: The function was called in the wrong position within a sequence of XTI function calls for transport endpoint *fd*.

TSYSERR
: A system error occurred during execution of this function.

See also

t_getinfo(), t_open()

t_rcv() - receive data over a connection

```
#include <xti.h>

int t_rcv(int fd, char *buf, unsigned nbytes, int *flags);
```

Description

The transport user can receive data over an established connection with the *t_rcv()* function. The *fd* parameter identifies the local transport endpoint over which the data is received. *buf* points to a receive buffer in which *t_rcv()* stores the incoming user data. The user specifies the size of this receive buffer with *nbytes*.
The *flags* parameter is not supported.

By default, *t_rcv()* works in synchronous mode, i.e. *t_rcv()* waits for further data to arrive and blocks if no data is currently available.
However, if O_NDELAY or O_NONBLOCK was previously set with *t_open()* or the POSIX *fcntl()* function for the transport endpoint specified by *fd, t_rcv()* works in asynchronous mode and terminates with an error if no data is available. *t_rcv()* then returns the value -1 and sets *t_errno* to TNODATA.

Return value

After successful execution, *t_rcv()* returns the number of received bytes.
If an error occurs, -1 is returned and *t_errno* is set to indicate the error.

Errors

TBADF
: The specified file descriptor does not reference a transport endpoint.

TLOOK
: An asynchronous event occurred on the transport endpoint passed in *fd* and this must be processed immediately.

TNODATA
: O_NDELAY or O_NONBLOCK was set but no data is currently available from the transport provider.

TNOTSUPPORT
: This function is not supported by the underlying transport provider.

TOUTSTATE
The function was called in the wrong position within a sequence of XTI function calls for transport endpoint *fd*.

TSYSERR
A system error occurred during execution of this function.

See also

t_open(), t_snd(), fcntl()

t_rcvconnect() - get the status of a connection request

```
#include <xti.h>

int t_rcvconnect(int fd, struct t_call *call);
```

Description

The transport user can call the *t_rcvconnect()* function to determine the status of a connection that was previously requested with *t_connect()* in asynchronous mode. In asynchronous mode, *t_rcvconnect()* is used in conjunction with *t_connect()* to set up a connection. The connection is set up after successful execution of *t_rcvconnect()*.

The *fd* parameter specifies the local transport endpoint on which the connection, which was previously requested with *t_connect()*, is to be set up. The *call* parameter points to an object of type *struct t_call* in which *t_rcvconnect()* returns information about the connection which was previously requested with *t_connect()*.

The *t_call* structure is declared in <xti.h> as follows:

```
struct t_call {
   struct netbuf addr;
   struct netbuf opt;
   struct netbuf udata;
   int sequence;
};
```

t_rcvconnect() returns the protocol address of the replying transport endpoint in *call->addr*. Returning protocol-specific information or user data in *call->udata* is not supported by the transport provider. *call->sequence* is meaningless for the *t_rcvconnect()* function.
Before calling *t_rcvconnect()*, the user must supply the *maxlen* component in the separate *netbuf* structures of **call* with the appropriate maximum buffer sizes.
The null pointer can also be passed as the current parameter for *call*. In this case, *t_rcvconnect()* does not return any information to the user.

By default, *t_rcvconnect()* works in synchronous mode, waits for confirmation of a connection previously requested with *t_connect()* and only returns control to the calling transport user after receiving the confirmation. After *t_rcvconnect()* is executed, *call->addr* contains the valid information about the connection that was just set up.

However, if the user previously set O_NDELAY or O_NONBLOCK with *t_open()* or the POSIX *fcntl()* function, *t_rcvconnect()* works in asynchronous mode. *t_rcvconnect()* then does not wait for connection confirmation, but returns control immediately to the calling user after getting the status of the connection request. If the requested connection is not set up yet, *t_rcvconnect()* returns the value -1 and sets *t_errno* to TNODATA. In this case, the user must call *t_rcvconnect()* again at a later time to complete the connection setup phase and receive the relevant information in *call->addr*.

Return value

0:
: If successful.

-1:
: If an error occurs. *t_ errno* is set to indicate the error.

Errors

TBADF
: The specified file descriptor does not reference a transport endpoint.

TBUFOVFLW
: The number of bytes reserved for a result parameter is not enough to store the value of the parameter. The state of the transport provider is set to T_DATAXFER from the viewpoint of the user and the information for the connection request that should be returned in **call* is removed.

TLOOK
: An asynchronous event occurred on the transport endpoint passed in *fd* and this must be processed immediately.

TNODATA
: O_NDELAY or O_NONBLOCK was set but no connection confirmation has arrived yet.

TNOTSUPPORT
: This function is not supported by the underlying transport provider.

TOUTSTATE
: The function was called in the wrong position within a sequence of XTI function calls for transport endpoint *fd*.

TSYSERR
: A system error occurred during execution of this function.

See also

t_accept(), t_bind(), t_connect(), t_listen(), t_open(), fcntl()

t_rcvdis() - get the cause of a connection shutdown

```
#include <xti.h>

int t_rcvdis(int fd, struct t_discon *discon);
```

Description

The user can get the cause of a connection shutdown with the *t_rcvdis()* function. The *fd* parameter specifies the local transport endpoint of the connection which was shut down. The *discon* parameter points to an object of type *struct t_discon*.

The *t_discon* structure is declared in <xti.h> as follows:

```
struct t_discon {
   struct netbuf udata;
   int reason;
   int sequence;
};
```

After execution of *t_rcvdis()*, *discon->reason* contains a protocol-dependent code which specifies the cause of the connection shutdown. This code corresponds to one of the possible values for the *errno* error variable (defined in <errno.h>). The following codes are currently possible:

ECONNREFUSED
: The connection request was refused by the partner.

ECONNRESET
: The connection was aborted by the partner.

ENETDOWN
: The connection was aborted by the transport system. In this case, the user should close the transport endpoint with *t_close()*.

ETIMEDOUT
: The connection could not be set up within a specific time.

The value returned in *discon->sequence* identifies a pending connection request which is associated with the connection setup. *discon->sequence* is only meaningful if the transport user that called the *t_rcvdis()* function previously called *t_listen()* one or more times to monitor socket *fd* for pending connection requests and is now processing these connection requests. When a connection shutdown request arrives, the user can check the value of *discon->sequence* to determine which of the pending connection requests is concerned. Returning user data in *discon->udata* is not supported by the transport provider.

If the transport user is not interested in the returned values of *discon->reason* and *discon->sequence*, he can specify the null pointer as the current parameter for *discon* with the *t_rcvdis()* call.

Return value

0:
: If successful.

-1:
: If an error occurs. *t_ errno* is set to indicate the error.

Errors

TBADF
: The specified file descriptor does not reference a transport endpoint.

TNODIS
: There is currently no connection shutdown request available on the specified transport endpoint.

TNOTSUPPORT
: This function is not supported by the underlying transport provider.

TOUTSTATE
: The function was called in the wrong position within a sequence of XTI function calls for transport endpoint *fd*.

TSYSERR
: A system error occurred during execution of this function.

See also

t_connect(), t_listen(), t_open(), t_snddis()

t_rcvrel() - confirm a connection shutdown request

```
#include <xti.h>

int t_rcvrel(int fd);
```

Description

The user can confirm reception of a request for orderly connection shutdown with the *t_rcvrel()* function. The *fd* parameter specifies the local transport endpoint belonging to the connection.

After receiving the request, the user should never cause permanent blocking. However, the user can send further data over the connection as long as he has not called the *t_sndrel()* function.

Return value

0:
: If successful.

-1:
: If an error occurs. *t_ errno* is set to indicate the error.

Errors

TBADF
: The specified file descriptor does not reference a transport endpoint.

TLOOK
: An asynchronous event occurred on the transport endpoint passed in *fd* and this must be processed immediately.

TNOREL
: There is currently no indication for an orderly connection shutdown on the specified transport endpoint.

TNOTSUPPORT
: This function is not supported by the underlying transport provider.

TOUTSTATE
: The function was called in the wrong position within a sequence of XTI function calls for transport endpoint *fd*.

TSYSERR
A system error occurred during execution of this function.

See also

t_open(), t_sndrel()

t_rcvudata() - receive datagrams

```
#include <xti.h>

int t_rcvudata(int fd, struct t_unitdata *unitdata, int *flags);
```

Description

The user can receive a datagram from another user in connectionless mode with the *t_rcvudata()* function.
The *fd* parameter specifies the local transport endpoint over which the datagram is received. After *t_rcvudata()* is executed, *flags* informs the user whether the datagram was received in full. *unitdata* is a pointer to an object of type *struct t_unitdata* in which *t_rcvudata()* returns information about the received datagram.

The *t_unitdata* structure is declared in <xti.h> as follows:

```
struct t_unitdata {
   struct netbuf addr;
   struct netbuf opt;
   struct netbuf udata;
};
```

Before calling *t_rcvudata()*, the user must supply the *maxlen* component in the separate *netbuf* structures of **unitdata* with the maximum values of each buffer size.

After *t_rcvudata()* is executed, *unitdata->addr* contains the protocol address of the sender, *unitdata->opt* contains protocol-specific options for the received datagram and *unitdata->udata* contains the received user data.

By default, *t_rcvudata()* works in synchronous mode, i.e. *t_rcvudata()* waits for a datagram to arrive and blocks if no datagrams are currently available.
However, if O_NDELAY or O_NONBLOCK was previously set with *t_open()* or the POSIX *fcntl()* function for the transport endpoint specified by *fd, t_rcvudata()* works in asynchronous mode and terminates with an error if no datagrams are available. *t_rcvudata()* then returns the value -1 and sets *t_errno* to TNODATA.

If the buffer in *unitdata->udata* is too small to store the datagram, *t_rcvudata()* stores as much of the datagram as possible in the buffer and sets the T_MORE flag. The T_MORE flag indicates that an additional *t_rcvudata()* call is needed to receive the remaining part of the datagram. Until the datagram is completely received, subsequent *t_rcvudata()* calls return the value 0 for the lengths of the protocol address and options.

Return value

0:
: If successful.

-1:
: If an error occurs. *t_ errno* is set to indicate the error.

Errors

TBADF
: The specified file descriptor does not reference a transport endpoint.

TBUFOVFLW
: The number of bytes that were reserved for the protocol address to be returned or the options, is too small to store this information. The information that should be returned in **unitdata* is deleted.

TLOOK
: An asynchronous event occurred on the transport endpoint passed in *fd* and this must be processed immediately.

TNODATA
: O_NDELAY or O_NONBLOCK was set but there are currently no datagrams available from the transport provider.

TNOTSUPPORT
: This function is not supported by the underlying transport provider.

TOUTSTATE
: The function was called in the wrong position within a sequence of XTI function calls for transport endpoint *fd*.

TSYSERR
: A system error occurred during execution of this function.
In the present case, the *errno* error variable contains more detailed information:

EFAULT	The area specified in *unitdata->addr*, *unitdata->opt* or *unitdata->udata* is outside the process address range.
ETIME	The datagram was deleted because a transport system-dependent time limit was exceeded (see also "Relevant settings with the BCAM transport system" on page 303).
EINTR	The call was interrupted by a signal.

See also

t_rcvuderr(), t_sndudata()

t_rcvuderr() - get error information about a sent datagram

```
#include <xti.h>

int t_rcvuderr(int fd, struct t_uderr *uderr);
```

Description

The user can get information about an error which occurred with a previously sent or received datagram in connectionless mode, with the *t_rcvuderr()* function. *t_rcvuderr()* should only be called after an error is indicated.

The *fd* parameter specifies the local transport endpoint over which the error message was received. The *uderr* parameter is a pointer to an object of type *struct t_uderr*.

The *t_uderr* structure is declared in <xti.h> as follows:

```
struct t_uderr {
   struct netbuf addr;
   struct netbuf opt;
   long error;
};
```

Before calling *t_rcvuderr()*, the user must supply the *maxlen* component in *uderr->addr* with the value of each maximum buffer size.
Returning protocol-specific options in *uderr->opt* is not supported by the transport provider.

t_uderr() returns a protocol-specific error code in *uderr->error*. This error code corresponds to one of the possible values for the *errno* error variable (defined in <errno.h>). The following codes are currently possible:

EADDRNOTAVAIL
: The partner to which the datagram was last to be sent with *t_sndudata()* is not reachable.

ENETDOWN
: The transport endpoint was cut off by the transport system. In this case, the user should close the transport endpoint with *t_close()*.

If the user does not want to determine the faulty datagram, he can pass the null pointer as the current parameter for *uderr* with the *t_rcvuderr()* call.

Return value

0:
If successful.

-1:
If an error occurs. *t_ errno* is set to indicate the error.

Errors

TBADF
The specified file descriptor does not reference a transport endpoint.

TBUFOVFLW
The number of bytes reserved for the protocol address to be returned or the options is too small to store this information. The information that is to be returned in **uderr* is not considered.

TNOTSUPPORT
This function is not supported by the underlying transport provider.

TNOUDERR
There is currently no error message for a datagram on the specified transport endpoint.

TSYSERR
A system error occurred during execution of this function.

See also

t_rcvdata(), t_sndudata()

t_snd() - send data over a connection

```
#include <xti.h>

int t_snd(int fd, char *buf, unsigned nbytes, int flags);
```

Description

The user sends data with the *t_snd()* function.
The *fd* parameter specifies the local transport endpoint over which the data is to be sent. *buf* is a pointer to the user data to be sent. The user specifies the length of the user data (in bytes) to be sent, with *nbytes*. *flags* is not supported by the transport provider and the value 0 must therefore be passed for *flags* with the *t_snd()* call.

By default, *t_snd()* works in synchronous mode and waits (blocks) if flow control limits prevent all data being taken over by the transport provider at the time of the *t_snd()* call. However, if O_NDELAY or O_NONBLOCK was previously set with *t_open()* or the POSIX *fcntl()* function for the transport endpoint specified with *fd*, *t_snd()* is executed in asynchronous mode and terminates with an error if flow control limits exist.

After successful execution, the return value of *t_snd()* defines the number of data bytes accepted by the transport provider. This number normally corresponds to the value passed in the *nbytes* parameter. However, in asynchronous mode it is possible that only part of the data to be sent is accepted by the transport provider. In this case, *t_snd()* returns a value less than *nbytes*.

Return value

After successful execution, *t_snd()* returns the number of bytes accepted by the transport provider.
If an error occurs, *t_errno* is set to -1 to indicate the error.

Errors

TBADDATA
The *nbytes* parameter has the value 0, but sending null bytes is not supported by the underlying transport provider.

TBADF
The specified file descriptor does not reference a transport endpoint.

TFLOW
O_NDELAY or O_NONBLOCK was set but the flow control has not allowed the transport provider to accept data at this time.

TLOOK
An asynchronous event occurred on the transport endpoint passed in *fd* and this must be processed immediately.

TNOTSUPPORT
This function is not supported by the underlying transport provider.

TOUTSTATE
The function was called in the wrong position within a sequence of XTI function calls for transport endpoint *fd*.

TSYSERR
A system error occurred during execution of this function.

See also

t_open(), t_rcv(), fcntl()

t_snddis() - refuse or abort a connection

```
#include <xti.h>

int t_snddis(int fd, struct t_call *call);
```

Description

The user can execute the following actions with the *t_snddis()* function:

- refuse a connection request
- initiate an abortive release of an established connection

The *fd* parameter specifies the local transport endpoint of the connection to be shut down or requested. The *call* parameter points to an object of type *struct t_call*.

The *t_call* structure is declared in <xti.h> as follows:

```
struct t_call {
   struct netbuf addr;
   struct netbuf opt;
   struct netbuf udata;
   int sequence;
};
```

The *call* parameter is used differently, depending on whether *t_snddis()* is to be used to refuse a connection request or set a connection up.

- If a connection request is to be refused, the null pointer must not be passed for *call* with the *t_snddis()* call. The user must specify a value in *call->sequence* that identifies the refused connection request to the transport provider. The contents of *call->addr*, *call->opt* and *call->udata* are ignored by *t_snddis()*.
- The null pointer can be passed for *call* if a connection is to be shut down.

Return value

0:
: If successful.

-1:
: If an error occurs. *t_ errno* is set to indicate the error.

Errors

TBADF
: The specified file descriptor does not reference a transport endpoint.

TBADSEQ
: An invalid sequential number was specified or the null pointer was specified for *call* when refusing connection request. The outgoing queue of the transport provider is deleted, which can cause loss of data.

TLOOK
: An asynchronous event occurred on the transport endpoint passed in *fd* and this must be processed immediately.

TNOTSUPPORT
: This function is not supported by the underlying transport provider.

TOUTSTATE
: The function was called in the wrong position within a sequence of XTI function calls for transport endpoint *fd*. The outgoing queue of the transport provider may be deleted, which can cause loss of data.

TSYSERR
: A system error occurred during execution of this function.

See also

t_connect(), t_getinfo(), t_listen(), t_open()

t_sndrel() - initiate an orderly connection shutdown

```
#include <xti.h>

int t_sndrel(int fd);
```

Description

The user initiates the orderly shutdown of a transport connection with the *t_sndrel()* function. *t_sndrel()* also informs the transport provider that the user will send no further data. The *fd* parameter specifies the local transport endpoint of the connection to be shut down.

After *t_sndrel()* is executed, the user must not send any further data over the connection. However, the user can receive further data over the connection as long as he has not received a request for orderly connection shutdown.

Return value

0:
: If successful.

-1:
: If an error occurs. *t_ errno* is set to indicate the error.

Errors

TBADF
: The specified file descriptor does not reference a transport endpoint.

TFLOW
: O_NDELAY or O_NONBLOCK was set but the flow control has not allowed the transport provider to accept the function at this time.

TLOOK
: An asynchronous event occurred on the transport endpoint passed in *fd* and this must be processed immediately.

TNOTSUPPORT
: This function is not supported by the underlying transport provider.

TOUTSTATE
: The function was called in the wrong position within a sequence of XTI function calls for transport endpoint *fd*.

TSYSERR
A system error occurred during execution of this function.

See also

t_open(), t_rcvrel()

t_sndudata() - send datagrams

```
#include <xti.h>

int t_sndudata(int fd, struct t_unitdata *unitdata);
```

Description

The user sends a datagram to another transport user in connectionless mode with the *t_sndudata()* function.
The *fd* parameter specifies the local transport endpoint over which the datagram is sent.
The *unitdata* parameter is a pointer to an object of type *struct t_unitdata*.

The *t_unitdata* structure is declared in <xti.h> as follows:

```
struct t_unitdata {
   struct netbuf addr;
   struct netbuf opt;
   struct netbuf udata;
};
```

Before calling *t_rcvudata()*, the user specifies the destination protocol address in *unitdata->addr* and the data to be transferred in *unitdata->udata*. Setting protocol-specific options in *unitdata->opt* is not supported by the transport provider.
If the user specified the value 0 in *unitdata->addr.len* and the transport provider does not support sending null bytes, *t_sndudata()* returns the value -1 and sets *t_errno* to TBADDATA.

By default, *t_sndudata()* works in synchronous mode and waits (blocks) if flow control limits prevent the transport provider from accepting the datagram at the time of the *t_sndudata()* call.
However, if O_NDELAY or O_NONBLOCK was previously set with *t_open()* or the POSIX *fcntl()* function for the transport endpoint specified with *fd*, *t_sndudata()* works in asynchronous mode and terminates with an error if the transport provider does not accept the datagram immediately.

If *t_sndudata()* was called in an invalid state or the datagram length specified in *unitdata->udata.len* is greater than the TSDU length, the transport provider generates an EPROTO protocol error (see the TSYSERR error). If the EPROTO error was generated because of an invalid state, it is only reported when transport endpoint *fd* is referenced. The length of the TSDU (transport service data unit) is returned by the *t_open()* and *t_getinfo()* functions.

Return value

0:
: If successful.

-1:
: If an error occurs. *t_ errno* is set to indicate the error.

Errors

TBADADDR
: The specified protocol address had the wrong format or contained invalid information.

TBADDATA
: The *nbytes* parameter has the value 0, but sending null bytes is not supported by the underlying transport provider, or the message was too long to be sent in one piece.

TBADF
: The specified file descriptor does not reference a transport endpoint.

TFLOW
: O_NDELAY or O_NONBLOCK was set but the flow control has not allowed the transport provider to accept data at this time.

TLOOK
: An asynchronous event occurred on the transport endpoint passed in *fd* and this must be processed immediately.

TNOTSUPPORT
: This function is not supported by the underlying transport provider.

TSYSERR
: A system error occurred during execution of this function.
In the present case, the *errno* error variable contains more detailed information:

EFAULT — The area specified in *unitdata->addr*, *unitdata->opt* or *unitdata->udata* is outside the process address range.

ENOBUFS — Not enough system resources are currently available to execute the send job.

EINTR — The call was interrupted by a signal.

See also

t_rcvudata(), t_rcvuderr(), fcntl()

t_strerror() - output error message

```
#include <xti.h>

char *t_strerror(int errnum);
```

Description

The user can generate the message text for an XTI error number or the relevant *t_errno* error code with the *t_strerror()* function.

t_strerror() maps the XTI error number specified by the *errnum* parameter to the relevant message string and returns a pointer to this character string. The message string is not changed by the program but can be overwritten by subsequent *t_strerror()* calls. The message string is not terminated with a newline character.

Return value

The *t_strerror()* function returns a pointer to the generated character string.

See also

t_error()

t_sync() - synchronize transport library

```
#include <xti.h>

int t_sync(int fd);
```

Description

The user can synchronize the data structures for the transport endpoint specified by *fd*, which are managed by the transport library, with information of the underlying transport provider using the *t_sync()* function. *t_sync()* also allows two cooperating processes to synchronize their interaction with the transport provider.
For example, if a process creates a new process and calls *exec()*, the new process must call the *t_sync()* function to:

– build up the private library data structure which is bound to a transport endpoint and

– synchronize the data structure with relevant transport provider information.

It must be noted that the transport provider sees all users of a transport endpoint as a single user. Therefore, if several user processes use the same transport endpoint, they should coordinate their tasks such that the transport provider does not end up in a faulty state. To do this, the separate user processes can call *t_sync()* to get the current state of the transport provider before initiating further actions.

Coordination with *t_sync()* is only allowed between cooperating processes as it is possible that a process or incoming event can change the state of the transport provider after *t_sync()* was executed.

Return value

After successful execution, *t_sync()* returns the state of the transport provider. If an error occurs, -1 is returned and *t_errno* is set to indicate the error.

The following transport provider states are possible as *t_sync()* return values:

T_UNBND
: The transport endpoint is not bound to the transport service.

T_IDLE
: The transport endpoint is bound to the transport service.

T_OUTCON
: A sent connection request has not been processed yet.

T_INCON
: A connection request which arrived has not been processed yet.

T_DATAXFER
: Data transfer phase.

T_OUTREL
: A request for orderly connection shutdown was sent (wait for indication of an orderly connection shutdown).

T_INREL
: Wait for a request for orderly connection shutdown.

Errors

TBADF
: The specified file descriptor does not reference a transport endpoint.

TSTATECHNG
: The transport provider is changing state.

TSYSERR
: A system error occurred during execution of this function.

See also

dup(), exec(), fork(), open()

t_unbind() - deactivate transport endpoint

```
#include <xti.h>

int t_unbind(int fd);
```

Description

The transport user can call the *t_unbind()* function to deactivate a transport endpoint, which was previously assigned an address with the *t_bind()* function. The *fd* parameter specifies the transport endpoint which is to be deactivated.
After *t_unbind()* has been successfully executed, the transport provider accepts no further data or events addressed to transport endpoint *fd*.

Return value

0:
: If successful.

-1:
: If an error occurs. *t_ errno* is set to indicate the error.

Errors

TBADF
: The specified file descriptor does not reference a transport endpoint.

TLOOK
: An asynchronous event occurred on the transport endpoint passed in *fd* and this must be processed immediately.

TNOTSUPPORT
: This function is not supported by the underlying transport provider.

TOUTSTATE
: The function was called in the wrong position within a sequence of XTI function calls for transport endpoint *fd*.

TSYSERR
: A system error occurred during execution of this function.

See also

t_bind()

12 Compiling and linking a communications application

This chapter describes:

- Compiling and linking a SOCKETS(POSIX) or XTI(POSIX) application program with the POSIX shell commands.
- Compiling and linking a SOCKETS(POSIX) application program in BS2000/OSD using two example BS2000/OSD procedures.

12.1 Compiling and linking with the POSIX shell

If the source file is stored in the UFS, you can compile your application program with the following POSIX shell command:

```
c89 -c program.c
```

If necessary, you can use the -O switch to optimize the program code and -g for debugging.

The following command links the compiled program:

```
c89 -o program program.o -lxnet
```

The following command compiles and links the program in one step:

```
c89 -o program program.c -lxnet
```

You must additionally install the POSIX_HEADER installation package to be able to compile your SOCKETS(POSIX) or XTI(POSIX) application program with the POSIX shell. The package installation is described in the POSIX manual “Basics for Users and System Administrators”.

The functions of SOCKETS(POSIX) support the ability of the C compiler to generate programs containing ASCII literals, see also the compiler manual “C/C++ V3.1A (BS2000/OSD)”.

12.2 Compiling and linking in BS2000/OSD

The example procedure shown below illustrates how a SOCKETS(POSIX) or XTI(POSIX) application can be compiled and linked in BS2000/OSD.

Example

```
/BEGIN-PROCEDURE LOGGING=ALL,PARAMETERS=YES(PROCEDURE-PARAMETERS=( -
/ &ELEMENT = ELEMENTNAME                                           -
/ ,&SRCLIB = TEST.SRC.LIB                                          -
/ ,&MODLIB = TEST.MOD.LIB                                          -
/ ,&PRGLIB = TEST.PRG.LIB                                          -
/ ,&PROMPT = NO                                                    -
/ ,&STDINCLIB = $TSOS.SYSLNK.CRTE                                  -
/ ,&STDINCLIB1 = $TSOS.SYSLNK.CRTE.CPP                             -
/ ,&STDINCLIB2 = $TSOS.SYSLIB.POSIX-HEADER                         -
/ ,&INCLIB = $TSOS.SYSLIB.POSIX-SOCKETS.050                        -
/ ,&SCHAL = $TSOS.SYSLNK.CRTE.POSIX                                -
/ ),ESCAPE-CHARACTER=C'&')                                         -
/ ,INTERRUPTION-ALLOWED=YES
/ASSIGN-SYSDTA *SYSCMD
/ASSIGN-SYSLST LST.C.&ELEMENT
/REMARK * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
/REMARK ** STARTING THE COMPILER **
/REMARK * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
/START-CPLUS-COMPILER
//MODIFY-SOURCE-PROPERTIES                                         -
//  LANGUAGE=*C(MODE=*ANSI),                                       -
//  DEFINE='_OSD_POSIX'
//MODIFY-INCLUDE-LIBRARIES                                         -
//  USER-INCLUDE-LIBRARY=*SOURCE-LIBRARY,                          -
//  STD-INCLUDE-LIBRARY=(                                          -
//    *STANDARD-LIBRARY,                                           -
//    &STDINCLIB2,                                                 -
//    &INCLIB)
//MODIFY-RUNTIME-PROPERTIES                                        -
//  PARAMETER-PROMPTING=&PROMPT
//MODIFY-LISTING-PROPERTIES                                        -
//  OPTIONS=*YES,                                                  -
//  SOURCE=*YES,                                                   -
//  SUMMARY=*YES,                                                  -
//  INCLUDE-INFORMATION=*ALL,                                      -
//  OUTPUT=*LIBRARY-ELEMENT(                                       -
//    LIBRARY=&MODLIB,                                             -
//    ELEMENT=&ELEMENT)
//COMPILE                                                          -
//  SOURCE=*LIBRARY-ELEMENT(                                       -
```

```
//     LIBRARY=&SRCLIB,                                              -
//     ELEMENT=&ELEMENT..C),                                         -
//   MODULE-OUTPUT=*LIBRARY-ELEMENT(                                 -
//     LIBRARY=&MODLIB,                                              -
//     ELEMENT=&ELEMENT)
//END
/REMARK * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
/REMARK
/REMARK IN THE FOLLOWING THE COMPILED PROGRAM IS LINKED TO THE REQUIRED
/REMARK LIBRARIES. THE LIBRARIES MUST BE SPECIFIED IN THE ORDER SHOWN IN
/REMARK THIS PROCEDURE.
/REMARK
/REMARK * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
/REMARK
/REMARK ** SET RESOLVES-LINKS ***
/REMARK
/SET-FILE-LINK LINK-NAME=BLSLIB01,FILE-NAME=&STDINCLIB
/SET-FILE-LINK LINK-NAME=BLSLIB02,FILE-NAME=&STDINCLIB1
/SET-FILE-LINK LINK-NAME=BLSLIB03,FILE-NAME=&INCLIB
/REMARK * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
/REMARK ** STARTING THE LINKER ***
/REMARK * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
/ START-BINDER
//START-LLM-CREATION INTERNAL-NAME=&ELEMENT,INCL-DEF=PAR(TEST-SUP=YES)
//INCLUDE-MODULES LIB=&MODLIB,ELEM=&ELEMENT,TYPE=L
//INCLUDE-MODULES LIB=&SCHAL,ELEMENT=*ALL,TYPE=(L,R),TEST-SUPPORT=YES
//RESOLVE-BY-AUTOLINK LIBRARY=*BLSLIB-LINK
//SAVE-LLM LIB=&PRGLIB,ELEM=&ELEMENT,OVER=YES,TEST-SU=YES
//END
/REMARK * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
/ASSIGN-SYSDTA *PRIMARY
/ASSIGN-SYSLST *PRIMARY
/END-PROCEDURE
```

13 Configuration and configuration files

When the POSIX subsystem is started, all steps required for configuring connection to the network are carried out automatically. The only thing visible to the user is the starting of the *inetd* daemon program by the *init* process.

This chapter describes:

- the *inetd* daemon program (Internet superserver)
- configuration files for hosts, networks, protocols and services
- dependencies of the SOCKETS(POSIX) and XTI(POSIX) applications on the BS2000/OSD BCAM transport system

13.1 inetd daemon program

inetd is one of the Internet daemons in UNIX systems. Since *inetd* plays a central role when starting the Internet services, it is also called the “Internet superserver”.

As also in UNIX systems, *inetd* is configured using the */etc/inet/inetd.conf* file. *inetd* is started when the system is booted and uses the *inetd.conf* file to determine which services are to be started via *inetd* if required. *inetd* then creates a socket for each service specified in the *inetd.conf* file and assigns a port number to each of these sockets.

inetd uses *select()* calls (see page 143) for the separate sockets to ensure that they are ready for reading. *inetd* then monitors the separate sockets with the *listen()* function for connection requests from the clients.
inetd proceeds as follows with each socket on which a connection request is pending:

1. *inetd* accepts the connection request with *accept()*.
2. *inetd* uses *fork()* and *dup()* to create two file descriptors for the socket, 0 (*stdin*) and 1 (*stdout*).
3. *inetd* starts the relevant services for the socket with *exec()*.

Using *inetd* therefore has the advantage that it is not necessary to start all server processes when the system is booted: a server only has to be started when a client has requests for it.

inetd also simplifies the tasks of a server as *inetd* takes care of most of the communications process during connection setup. The server can assume that the communications endpoint assigned to it has file descriptors 0, 1 and 2 and is already connected to the client. This allows the server to immediately execute functions such as *read*(), *write()*, *send()* or *rcv()*, i.e. the server program code can be kept very simple.

An application programmer who develops servers started via *inetd* can get the address of the communications partner, i.e. the address of the client socket, with the *getpeername()* function (see page 100).

13.2 Configuration files

The following files are described in this section:

- inetd.conf
- protocols
- services
- networks
- hosts

If you have modified these files, you must direct the *inetd* daemon program to reread the files with the following POSIX command:

```
kill -1 process_number_of_inetd
```

13.2.1 inetd.conf - available servers

The file contains entries for the servers which the *inetd* daemon program calls when a request arrives over the socket interface. By default, only the *echo* and *time* services of *inetd* and the R command programs (*rlogin* ...) are activated.

Each entry for a server consists of a line in the following format:

```
service_name socket_type protocol wait_state ID server_program
server_arguments
```

service_name
: Name of the service, as entered in */etc/inet/services*

socket_type
: Type of the socket. Datagram or stream socket

protocol
: Name of the protocol, as entered in */etc/inet/protocols. tcp6* and *udp6* can also be used instead of *tcp* and *udp*, provided the server involved supports IPv6.

wait_state
: Defines whether the server releases the socket immediately (*nowait*) or only after a certain time (*wait*).

ID
: User ID under which the server is to run

server_program
: Path name of the server program

server_arguments
: Possible parameters for the server call

Example

```
#
# Shell and login are BSD protocols.
#

shell   stream   tcp   nowait   sysroot   /usr/sbin/in.rshd      in.rshd
login   stream   tcp   nowait   sysroot   /usr/sbin/in.rlogind   in.rlogind


#
# Echo, discard,daytime, and chargen are used primarily for testing.
##

echo      stream   tcp   nowait   sysroot   internal
echo      dgrm     udp   wait     sysroot   internal
discard   stream   tcp   nowait   sysroot   internal
discard   dgrm     udp   wait     sysroot   internal
daytime   stream   tcp   nowait   sysroot   internal
daytime   dgrm     udp   wait     sysroot   internal
chargen   stream   tcp   nowait   sysroot   internal
chargen   dgrm     udp   wait     sysroot   internal
```

Additionally installed server applications can be entered into this file by systems support.

13.2.2 protocols - available protocols

The file contains information about the possible protocols. Each entry for a protocol consists of a line in the following format:

```
protocol_name  protocol_number aliases #comment
```

Example

```
ip        0     IP        # internet protocol, pseudo protocol number
icmp      1     ICMP      # internet control message protocol
ggp       3     GGP       # gateway-gateway protocol
tcp       6     TCP       # transmission control protocol
egp       8     EGP       # exterior gateway protocol
pup       12    PUP       # PARC universal packet protocol
udp       17    UDP       # user datagramm protocol
hmp       20    HMP       # host monitoring protocol
xns-idp   22    XNS-IDP   # Xerox NS IDP
rdp       27    RDP       # “reliable datagram” protocol
```

The file is to be considered as being static since the numbers are only assigned by the standards committees (OSI, IEEE and IANA).

13.2.3 services - available services

The file contains information about the services. Each entry for a service consists of a line in the following format:

```
service_name   port_number/protocol   aliases # comment
```

A short extract from the file is shown below:

```
tcpmux     1/tcp
echo       7/tcp
echo       7/udp
telnet     23/tcp
smtp       25/tcp     mail
snmp       161/udp                   # network management agent
login      513/udp                   # Xerox NS IDP
nfsd       2049/udp                  # NFS server daemon
xserver    6000/tcp                  # X-Window server display
```

This file is static to a large degree since most of the numbers are standardized. However, free numbers can be assigned in local networks.

13.2.4 networks - reachable networks

The file contains information about reachable networks. Each entry for a network consists of a line in the following format:

```
network_name network_number aliases
```

Aliases are alternative names for the network, which are only known on the local system.

Example

```
loopback        127
firm            132.45
```

The file contains one standard entry. *loopback* designates a network interface for local communications.

Additional reachable networks can be entered into the file by systems support. It must be noted that a network is only reachable if the system routing has information about the network.

13.2.5 hosts - reachable hosts

This file contains information about reachable (known) hosts. Each entry consists of a line in the following format:

```
host_address host_name aliases
```

The file contains the following standard entry

```
127.0.0.1     localhost  local
```

Entries by systems support are only required here if an application uses the *gethostent()* function (see page 91) to get the reachable host names and addresses. This information can be obtained via DNS or BCAM with the *gethostbyname(), gethostbyaddr(), getipnodebyname()* and *getipnodebyaddr()* functions (see page 91 and page 94).

13.3 Dependencies of the BS2000/OSD BCAM transport system

This section outlines the points you must note with SOCKETS(POSIX) and XTI(POSIX) applications with regard to the BS2000/OSD BCAM transport system. Please refer to the manual "openNet Server V3.0 (BS2000/OSD)" [6] for more detailed information on BCAM.

BCAM as the SOCKETS(POSIX) and XTI(POSIX) communications manager

BCAM supports several communications architectures as the basis of the data communications system for BS2000/OSD hosts. Socket and XTI applications can communicate via the TCP/IP and UDP/IP protocols of the Internet architecture. The communications system is managed with BCAM administration commands. The most important BCAM commands in this respect are BCIN (generate end systems dynamically) and BCSHOW (get state information, e.g. port assignments). You can also use the corresponding SDF commands instead of BCIN and BCSHOW. You will find additional BCAM commands that are relevant for socket and XTI applications under "Relevant settings with the BCAM transport system" on page 303.

Assigning a socket or transport endpoint a special Internet address

When SOCKETS(POSIX) is used on a system with more than one Internet port, linking of a socket to a special Internet port is supported. For this purpose it may be necessary to set the SO_REUSEADDR option with the *setsockopt()* function. However, up to and including BCAM V17 the interface actually used to send datagrams and connection requests cannot be selected in this way for sockets of the type SOCK_DGRAM or client sockets of the type SOCK_STREAM.

Dependency of IPv6 support in SOCKETS(POSIX) on BCAM and SOCKETS(BS2000)

Communications with SOCKETS(POSIX) in the AF_INET6 (IPv6) Internet address family requires BCAM version 16.0.

The new functions *getaddrinfo()*, *freeaddrinfo()*, *gai_strerror()*, *getipnodebyaddr()*, *getipnodebyname()*, *freehostinfo()*, *getnameinfo()*, *inet_ntop()* and *inet_pton()*, which were introduced with the IPv6 support in accordance with RFC 2553, can only be executed on systems with BCAM V16.0 (or higher).

These functions always return the error EAFNOSUPPORT if they are executed on a system with an earlier BCAM version.

The functions *getaddrinfo()*, *getipnodebyaddr()*, *getipnodebyname()* and *getnameinfo()* use the DNS Resolver service of the SOCKETS(BS2000) subsystem (also for IPv4 addresses). This requires at least SOCKETS(BS2000) V2.0.

Relevant settings with the BCAM transport system

The BCAM commands which have effects on SOCKETS(POSIX) and XTI(POSIX) applications are shown below. The BCAM commands and their parameters are described in detail in the “openNet Server V3.0 (BS2000/OSD)” manual.

BCAM DCSTART and BCMOD commands (BCAM limits)

The MAXNPA, MAXNPT and MAXCNN operands limit the number of network applications and connections.

BCAM BCTIMES command (BCAM time settings)

The CONN operand limits the wait time for connection requests.
The DATAGRAM operand limits the linger period for connectionless transport service messages.
The LETT operand limits the linger period for connection-oriented transport service messages.

BCAM BCOPTION command (BCAM mode options)

The BROADCAST operand defines whether the host is allowed to receive broadcast messages. There are no restrictions on sending broadcast messages.

BCAM BCMOD and DCOPT commands (predefine/modify DCSTART parameters)

The FREEPORT# operand defines the first free port number that can be assigned dynamically by BCAM to an application. PRIVPORT# defines the first socket port number that can be assigned to non-privileged and privileged applications.
FREEPORT# must always be greater than or equal to PRIVPORT#.

Support for the Domain Name Service (DNS)

SOCKETS(POSIX) supports the DNS concept if the subsystem SOCKETS(BS2000) V2.0 and/or the DNS Resolver from the product *interNet* Services (formally TCP-IP-SV) have been configured and started, see the manual "interNet Services V3.0 (BS2000/OSD) Administrator Guide".
The DNS collects information on the hosts connected to a network and makes this information available to all hosts via the network.

If you are using BCAM as of V16.0, the following DNS functionality is available:

– for *gethostbyname()*, *gethostbyaddr()*
  DNS Resolver functionality in *inter*Net Services (TCP-IP-SV)
  DNS Resolver functionality in SOCKETS(BS2000)

– for *getipnodebyname()*, *getipnodebyaddr()*, *getaddrinfo()*, *getnameinfo()*
  DNS Resolver functionality in SOCKETS(BS2000)

14 Compatibility restrictions

Compatibility to UNIX applications

The SOCKETS(POSIX) interface implementation complies with the SINIX V5.41 implementation and has been enhanced by the support of IPv6. This ensures that SOCKETS(POSIX) applications are source-compatible to UNIX to a large degree. The following restrictions apply:

- The RAW socket interface is not supported.
- Out-of-band data is not supported.
- Wait points for all blocking operations on POSIX file descriptors lie in the POSIX subsystem and are therefore out of reach to the application programmer.

Compatibility to SOCKETS(BS2000) applications

SOCKETS(BS2000) applications are not compatible to applications developed with the SOCKETS(POSIX) functions.

XTI Compatibilities

The following restrictions apply for XTI:

- Sending expedited data is not supported.
- Only the transport services for TCP/IP and UDP/IP are supported.
- IPv6 is not supported.

Related publications

The manuals are available as online manuals, see *http://manuals.fujitsu-siemens.com*, or in printed form which must be paid and ordered separately at *http://FSC-manualshop.com*.

[1] **C Library Functions** (BS2000/OSD)
for POSIX Applications
Reference Manual

Target group
This manual addresses C and C++ programmers.
Contents
The manual documents the XPG4-conformant C programming interface which is supported by the POSIX subsystem in BS2000. This programming interface permits access to both the POSIX file system and BS2000 files. The programming interface also incorporates extensions which ensure compatibility with the existing C library described in a separate chapter.

[2] **POSIX (BS2000/OSD)**
Commands
User Guide

Target group
This manual addresses all users of the POSIX shell.
Contents
This manual is designed as a work of reference. It describes working with the POSIX shell and the commands of the POSIX shell in alphabetical order.

[3] **POSIX** (BS2000/OSD)
Basics for Users and System Administrators
User Guide

Target group
BS2000 system administrators, POSIX administrators, BS2000 users, users of UNIX workstations
Contents
– Introduction to and working with POSIX
– BS2000 software products in a POSIX environment
– Installing POSIX
– Controlling POSIX and administering file systems
– Administering POSIX users
– BS2000 commands for POSIX

[4] **Reliant UNIX V5.45**
Network Programming Interfaces
Programmer's Guide

Target group
The manual is aimed at application programmers who want to develop software for use in networks.
Contents
It describes how to write application programs that use the Reliant UNIX network functions (TLI, Sockets, RCP).

[5] **C/C++ V3.1A** (BS2000/OSD)
C/C++ Compiler
User Guide

Target group
C and C++ users in a BS2000 environment.
Contents
– Description of all activities in the creation of executable C and C++ programs: compilation, linking, loading, debugging
– Programming notes and detailed information on: optimization, program flow control, linking of functions and languages, C and C++ language scope of the compiler.

[6] *open*Net Server V3.0 (BS2000/OSD)
BCAM V17.0A
User Guide

Target group
The manual is intended for network planners, generators and administrators who define BCAM BS2000 systems.
Contents
The manual describes BCAM itself, how it is embedded in TRANSDATA and TCP/IP and ISO networks, plus generation and administrative activities. Generation examples illustrate the description. Additionally BCAM tools for generation and diagnosis are described. To conclude, the manual describes BCAM commands required for generation and operation. Information is also provided on KOGS macros required for statistical generation and a list of BCAM error messages is given.

[7] *inter*Net Services V3.0 (BS2000/OSD)
Administrator Guide

Target group
This manual is intended for network planners, generators and administrators who wish to use Internet Services in BS2000/OSD.
Contents
The manual describes the functionality of the Internet Services BOOTP/DHCP, TFTP, DNS, FTP, TELNET, LDAP and NTP in BS2000/OSD. It also covers the installation, administration, operation, and logging and diagnostic options of the individual components, TLS-Support in FTP and TELNET Servers, FTP exit and the TELNET exits as well as generating random numbers in BS2000/OSD and POSIX.

[8] *inter*Net Services V3.0 (BS2000/OSD)
User Guide

Target group
This manual is intended for users and network planners, generators and administrators who wish to use Internet Services in conjunction with BS2000/OSD.
Contents
The manual introduces the components of *inter*Net Services. It contains a detailed description of FTP, TELNET, the FTAC interface for FTP and TELNET, and also of the Mailreader. A further major topic of the manual is TLS/SSL support of FTP and TELNET. Network administrators require this manual as a supplement to the Administrator Guide.

[9] ***open*Net Server** (BS2000/OSD)
IPv6 Introduction and Conversion Guide, Stage 1
User Guide

Target group
This manual is intended for everyone responsible for deciding as to the introduction of IPv6 in BS2000/OSD, as well as anyone using the IPv6 functionality on BS2000/OSD mainframes or planning to install IPv6 in BS2000/OSD.
Contents
The manual explains the commercial and technical foundations of IPv6. In addition, it describes the transition from IPv4 to IPv6 with the aid of examples and outlines the current status of the implementation of IPv6 in BS2000/OSD. Detailed information on “IPv6 addressing” and “DNS utilization” can be found in the appendix.

Other publications

X/Open CAE Specification
Networking Services, Issue 4

Index

Contents

POSIX

SOCKETS/XTI for POSIX
User Guide

Target group

C and C++ programmers who develop communications applications on the basis of the POSIX interface using SOCKETS and/or XTI functions.

Contents

– Introduction to SOCKETS(POSIX)
– User functions of SOCKETS(POSIX)
– Introduction to XTI(POSIX)
– XTI trace
– Library functions of XTI(POSIX)
– Compiling and linking of communications applications
– Configuration and configuration files, BCAM dependencies
– Restrictions on compatibility

Edition: March 2005

File: posix_s.pdf

This manual was produced by
cognitas. Gesellschaft für Technik-Dokumentation mbH
www.cognitas.de

Fujitsu Siemens computers GmbH
User Documentation
81730 Munich
Germany

Fax: (++49) 700 / 372 00001

e-mail: manuals@fujitsu-siemens.com
http://manuals.fujitsu-siemens.com

Comments
Suggestions
Corrections

Submitted by

Comments on POSIX
SOCKETS/XTI for POSIX

Information on this document

On April 1, 2009, Fujitsu became the sole owner of Fujitsu Siemens Computers. This new subsidiary of Fujitsu has been renamed Fujitsu Technology Solutions.

This document from the document archive refers to a product version which was released a considerable time ago or which is no longer marketed.

Please note that all company references and copyrights in this document have been legally transferred to Fujitsu Technology Solutions.

Contact and support addresses will now be offered by Fujitsu Technology Solutions and have the format *…@ts.fujitsu.com*.

The Internet pages of Fujitsu Technology Solutions are available at *http://ts.fujitsu.com/*...
and the user documentation at *http://manuals.ts.fujitsu.com*.

Hinweise zum vorliegenden Dokument

Zum 1. April 2009 ist Fujitsu Siemens Computers in den alleinigen Besitz von Fujitsu übergegangen. Diese neue Tochtergesellschaft von Fujitsu trägt seitdem den Namen Fujitsu Technology Solutions.

Das vorliegende Dokument aus dem Dokumentenarchiv bezieht sich auf eine bereits vor längerer Zeit freigegebene oder nicht mehr im Vertrieb befindliche Produktversion.

Bitte beachten Sie, dass alle Firmenbezüge und Copyrights im vorliegenden Dokument rechtlich auf Fujitsu Technology Solutions übergegangen sind.

Kontakt- und Supportadressen werden nun von Fujitsu Technology Solutions angeboten und haben die Form *…@ts.fujitsu.com*.

Die Internetseiten von Fujitsu Technology Solutions finden Sie unter *http://de.ts.fujitsu.com/*..., und unter *http://manuals.ts.fujitsu.com* finden Sie die Benutzerdokumentation.