1 Einleitung

SOCKETS/XTI(POSIX) ist die Bezeichnung für die Socket- und XTI-Funktionen innerhalb der POSIX-Schnittstelle für BS2000/OSD. Diese Funktionen bieten die Entwicklungsumgebung für BS2000/OSD-Anwender, die Socket- oder XTI-Anwenderprogramme unter POSIX schreiben wollen.

1.1 Kurzbeschreibung des Produkts

Mit POSIX werden die Socket- und XTI-Funktionen gemäß der Spezifikation der X/Open-Group für das UNIX95-Branding angeboten. Die Socket- und XTI-Programmierung bietet eine Reihe von Möglichkeiten bei der Entwicklung von Kommunikationsanwendungen:

- Die Socket-Schnittstelle (SOCKETS) ist eine der Schnittstellen zur Netzwerkprogrammierung innerhalb des POSIX-Subsystems. Damit können Kommunikationsanwendungen auf Basis der TCP/IP-Protokolle entwickelt werden.
- X/Open Transport Interface (XTI) ist der von X/Open definierte Standard für eine Reihe von Programmierschnittstellen, die einer Anwendung den Zugang zur Netzwerkebene ermöglichen.
- RFC 2553 für IPV6-Sockets-Anwendungen

1.2 Zielgruppe des Handbuchs

Das vorliegende Handbuch wendet sich an Programmierer, die mit den Funktionen der SOCKETS- bzw. XTI-Schnittstelle Kommunikationsanwendungen auf der Basis der POSIX-Schnittstelle entwickeln.
Kenntnisse der C-Programmierung und der POSIX-Funktionen werden vorausgesetzt.

1.3 Wegweiser durch das Handbuch

Im vorliegenden Handbuch sind die verschiedenen Möglichkeiten der Socket- und XTI-Programmierung beschrieben und anhand einiger einfacher Beispiele erläutert. Die Beispielprogramme zeigen die Verwendung von SOCKETS- bzw. XTI-Funktionen sowohl für verbindungsorientierte Kommunikationsanwendungen über das TCP-Protokoll als auch für verbindungslose Kommunikationsanwendungen über das UDP-Protokoll.

Das Handbuch ist wie folgt aufgebaut:

- Die Kapitel 2 bis 5 geben eine Einführung in die Entwicklung von SOCKETS(POSIX)-Kommunikationsanwendungen. Anhand von Programmbeispielen werden grundlegende Themen wie Adress-Strukturen, Verbindungsaufbau, Datenübertragung und Client-/Server-Kommunikation behandelt.
- Im Kapitel 6 finden Sie einen alphabetischen Nachschlageteil mit den Benutzerfunktionen der SOCKETS(POSIX)-Schnittstelle.
- Die Kapitel 7 bis 9 geben eine Einführung in die Entwicklung von XTI(POSIX)-Kommunikationsanwendungen. Anhand von Programmbeispielen werden grundlegende Themen wie Verbindungsaufbau, Datenübertragung und Client-/Server-Kommunikation behandelt.
- Im Kapitel 10 ist der XTI-Trace beschrieben.
- Im Kapitel 11 finden Sie einen alphabetischen Nachschlageteil mit den Bibliotheksfunktionen der XTI(POSIX)-Schnittstelle.
- Im Kapitel 12 wird anhand von zwei Beispiel-Prozeduren gezeigt, wie Sie Ihre fertiggestellten Programme übersetzen und binden können.
- Im Kapitel 13 sind das Internet-Dämonprogramm *inetd* sowie die Konfigurationsdateien der Internet-Kommunikation beschrieben. Außerdem sind die Abhängigkeiten der SOCKET(POSIX)- bzw. XTI(POSIX)-Anwendungen vom BS2000/OSD-Transportsystem BCAM dargestellt.
- Im Kapitel 14 sind die Kompatibilitäts-Einschränkungen der SOCKETS(POSIX)- bzw. XTI(POSIX)-Schnittstelle gegenüber den folgenden Schnittstellen beschrieben:
  - Socket- bzw. XTI-Schnittstelle unter UNIX-Systemen
  - Socket-Schnittstelle im BS2000/OSD

1.4 Änderungen gegenüber der Ausgabe Februar 2001

Dieser Abschnitt gibt einen Überblick über die Änderungen im Handbuch SOCKETS/XTI für POSIX Ausgabe März 2005 gegenüber der Ausgabe Februar 2001.

- Erweiterung der Funktion *ioctl()* um neue Steuerfunktionen:

  SIOCGLIFNUM
  SIOCGLIFCONF
  SIOCGLIFADDR
  SIOCGLIFINDEX
  SIOCGLIFBRDADDR
  SIOCGLIFNETMASK
  SIOCGLIFFLAGS
  SIOCGIFNUM
  SIOCGIFINDEX
  SIOCGIFNETMASK

- Außerdem unterstützen die Funktionen von SOCKETS(POSIX) jetzt die Möglichkeit des C-Compilers, Programme zu erzeugen, die ASCII-Literale enthalten.

1.5 Typografische Gestaltungsmittel

Im vorliegenden Handbuch werden die folgenden typografischen Gestaltungsmittel verwendet:

für Hinweistexte

—————— ——————	Syntaxdefinitionen sind oben und unten durch waagrechte Linien begrenzt; Fortsetzungszeilen innerhalb von Syntaxdefinitionen sind eingerückt.
`dicktengleiche Schrift`	Programmtext in Beispielen; Syntaxdarstellungen.
kursive Schrift	Namen von Programmen, Funktionen, Funktionsparametern, Dateien, Strukturen und Strukturkomponenten im beschreibenden Text; Syntaxvariable (z.B. *dateiname*)
<spitze Klammern>	kennzeichnen Include-Dateien im beschreibenden Text.
[ ]	Optionale Angaben. Die eckigen Klammern sind Metazeichen, die innerhalb von Anweisungen nicht angegeben werden dürfen.
...	In Syntaxdefinitionen bedeuten die Punkte, dass die vorausgehende Angabe beliebig oft wiederholt werden kann. In Beispielen bedeuten die Punkte, dass die restlichen Teile für das Verständnis des Beispiels ohne Bedeutung sind. Die Punkte sind Metazeichen, die innerhalb von Anweisungen nicht angegeben werden dürfen.

Die Gestaltungsmittel für die Beschreibung der Benutzerfunktionen werden am Anfang der entsprechenden Kapitel vorgestellt.

Verweise innerhalb des Handbuchs geben die betreffende Seite im Handbuch und je nach Bedarf auch den Abschnitt bzw. das Kapitel an. Verweise auf Themen, die in einem anderen Handbuch beschrieben sind, enthalten den Kurztitel des betreffenden Handbuchs. Den vollständigen Titel finden Sie im Literaturverzeichnis.

1.6 Readme-Datei

Funktionelle Änderungen und Nachträge der aktuellen Produktversion zu diesem Handbuch entnehmen Sie bitte ggf. der produktspezifischen Readme-Datei. Sie finden die Readme-Datei auf Ihrem BS2000-Rechner unter dem Dateinamen SYSRME.*produkt*.*version*.*sprache*. Die Benutzerkennung, unter der sich die Readme-Datei befindet, erfragen Sie bitte bei Ihrer zuständigen Systembetreuung. Die Readme-Datei können Sie mit dem Kommando /SHOW-FILE oder mit einem Editor ansehen oder auf einem Standarddrucker mit folgendem Kommando ausdrucken:

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

2 Grundlagen von SOCKETS(POSIX)

In diesem Kapitel werden grundlegende Begriffe und Funktionen der Socket-Programmierung erläutert. Programmbeispiele zu den in diesem Kapitel behandelten Themen sind im Kapitel „Client-/Server-Modell bei SOCKETS(POSIX)“ auf Seite 61 zusammengefasst. Die einzelnen Funktionen der SOCKETS-Schnittstelle sind ausführlich beschrieben im Kapitel „Benutzerfunktionen von SOCKETS(POSIX)“ auf Seite 73.

2.1 POSIX-Netzanbindung über die SOCKETS-Schnittstelle

Die SOCKETS-Schnittstelle ist eine der Schnittstellen zur Netzwerkprogrammierung innerhalb des POSIX-Subsystems. Damit können Kommunikationsanwendungen auf der Basis der TCP/IP-Protokolle entwickelt werden. NEA- und OSI-Protokolle werden nicht unterstützt.

Die SOCKETS-Schnittstelle ist in einer eigenen Bibliothek definiert. Wenn diese Bibliothek in eine POSIX-Anwendung eingebunden ist, stellen die SOCKETS-Schnittstellen über das Subsystem POSIX und das Transportsystem BCAM die Verbindung zum Netzwerk her.

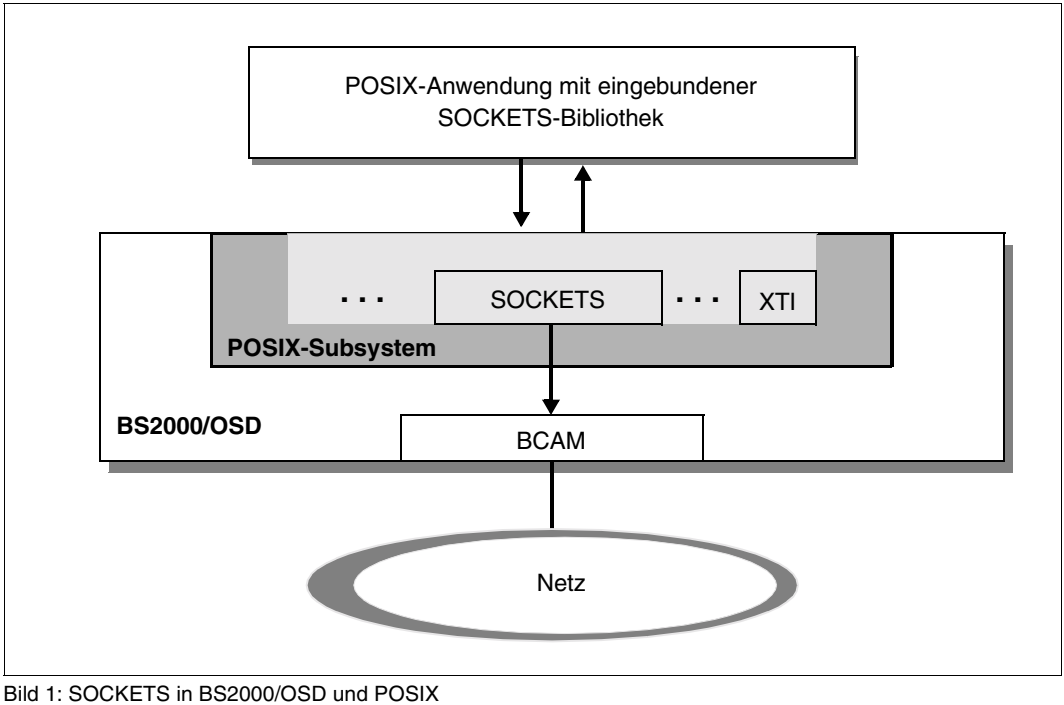

Bild 1: SOCKETS in BS2000/OSD und POSIX

Die Bibliotheken zur Netzanbindung in POSIX stellen das Bindeglied zwischen den POSIX-Objekten, wie z.B. Dateideskriptoren, und den BS2000-Mechanismen dar. Durch die Unterschiede der POSIX-Konzepte gegenüber dem BS2000 ergeben sich einige Einschränkungen im Gebrauch der Funktionen. Diese Einschränkungen sind näher beschrieben im Kapitel „Einschränkungen zur Kompatibilität“ auf Seite 319.

Die Funktionen zum Eröffnen einer Netzverbindung liefern einen Socket-Dateideskriptor zurück. Dieser kann in allen relevanten POSIX-Funktionen verwendet werden, die mit Dateideskriptoren arbeiten.

2.2 Include-Dateien

Bei der Installation von SOCKETS(POSIX) werden X/Open-konforme Include-Dateien in das Verzeichnis */usr/include* kopiert. In Kapitel „Benutzerfunktionen von SOCKETS(POSIX)“ auf Seite 73 und Kapitel „Bibliotheksfunktionen von XTI(POSIX)“ auf Seite 235 ist bei der Beschreibung jeder Socket- bzw. XTI-Funktion angegeben, welche Include-Datei(en) eine Anwendung für die Ausführung dieser Funktion einbinden muss. SOCKETS/XTI(POSIX) stellt folgende Include-Dateien zur Verfügung:

arpa/inet.h
- Definition von Hilfsfunktionen und Makros für die Manipulation von Internet-Adressen
- Definition der Struktur *in_addr* analog der Definition in <netinet/in.h>

sys/sockio.h
- Definitionen für die von der Socket-Funktion *ioctl()* aufgerufenen Socket-Steuerfunktionen

net/if.h
- Strukturen für Paketvermittlungs-Interface

netdb.h
- Strukturen und Funktionsdeklarationen für Hilfsfunktionen zur Adressumwandlung
- Definitionen der Flags zur Steuerung der Adressumwandlungsfunktionen
- Definitionen der Fehlermeldungen der Adressumwandlungsfunktionen

netinet/in.h
- Definition der Adress-Strukturen für die Internet-Domänen (AF_INET, AF_INET6)
- symbolische Konstanten für Protokolltypen
- Testmakros für die Domäne AF_INET6

sys/socket.h
- Definition der Socket-Adress-Struktur und anderer Strukturen für Socket-Systemfunktionen
- Deklaration der Socket-Systemaufrufe
- symbolische Konstanten für Socket-Optionen und Socket-Typen

sys/time.h
- *timval*-Struktur für *select()* und Subfunktion *linger*

sys/byteorder.h
- Makros zur Umwandlung der Byte-Reihenfolge

sys/un.h
- Adress-Struktur für Domäne von UNIX-Systemen (AF_UNIX)

sys/xti_inet.h
- Internet-spezifische Strukturen und Optionen des Transportanbieters

xti.h
- Deklaration der XTI-Funktionen
- Strukturen und Konstanten des Transportanbieters
- symbolische Konstanten für XTI-Fehlercodes
- Zustände und Optionen des Transportendpunkts

2.3 Socket-Typen

Ein Socket ist ein grundlegender Baustein für die Entwicklung von Kommunikationsanwendungen. Ein Socket bildet einen Kommunikationsendpunkt. Ihm kann ein Name zugeordnet werden, über den der Socket angesprochen und adressiert werden kann.

Jeder Socket gehört einem bestimmten Typ an und hat mindestens einen zugehörigen Prozess. Mehrere verwandte Prozesse können denselben Socket verwenden. Ein Prozess kann Verbindung zu mehreren Sockets haben.

Ein Socket gehört zu einer bestimmten Kommunikationsdomäne. Eine Kommunikationsdomäne fasst Adressfamilien und Protokollfamilien zusammen. Eine Adressfamilie umfasst Adressen mit gleicher Adress-Struktur. Eine Protokollfamilie definiert einen Satz von Protokollen, die Socket-Typen in der Domäne implementieren. Zweck der Kommunikationsdomänen ist die Zusammenfassung gemeinsamer Eigenschaften von Prozessen, die über Sockets kommunizieren.
Die Socket-Schnittstelle im BS2000/OSD unterstützt die Internet-Kommunikationsdomänen AF_INET und AF_INET6 sowie die rechner-lokale Kommunikationsdomäne AF_UNIX.

Entsprechend den Kommunikationseigenschaften der Sockets gibt es verschiedene Socket-Typen. Derzeit werden zwei verschiedene Typen von Sockets unterstützt:

– Stream-Sockets
– Datagramm-Sockets

2.3.1 Stream-Sockets (verbindungsorientiert)

Stream-Sockets unterstützen die verbindungsorientierte Kommunikation in den Internet-Kommunikationsdomänen AF_INET und AF_INET6 sowie in der lokalen Domäne AF_UNIX. Ein Stream-Socket bietet bidirektionalen, gesicherten und sequenziellen Datenfluss. Somit gewährleisten Stream-Sockets, dass die Daten nur einmal und in der richtigen Reihenfolge übertragen werden. Die Satzgrenzen der Daten bleiben bei der verbindungsorientierten Kommunikation mit Stream-Sockets nicht erhalten.

Mit Stream-Sockets werden verbindungsorientierte Kommunikationsanwendungen auf der Basis des TCP-Protokolls entwickelt.

2.3.2 Datagramm-Sockets (verbindungslos)

Datagramm-Sockets unterstützen die verbindungslose Kommunikation in den Internet-Kommunikationsdomänen AF_INET und AF_INET6 sowie in der lokalen Domäne AF_UNIX. Ein Datagramm-Socket ermöglicht bidirektionalen Datenfluss. Dabei gewährleisten Datagramm-Sockets jedoch weder eine gesicherte noch eine sequenzielle Übertragung der Daten. Außerdem kann nicht ausgeschlossen werden, dass die Daten mehrmals übertragen werden. Ein Prozess, der Nachrichten auf einem Datagramm-Socket empfängt, kann somit die Nachrichten möglicherweise doppelt und/oder in einer von der Sende-Reihenfolge abweichenden Reihenfolge vorfinden. Es ist deshalb Aufgabe der Anwendung, den korrekten Empfang der Daten zu überprüfen und sicherzustellen. Eine wichtige Eigenschaft von Datagramm-Sockets ist die Erhaltung der Satzgrenzen der übertragenen Daten.

Mit Datagramm-Sockets werden verbindungslose Kommunikationsanwendungen auf der Basis des UDP-Protokolls entwickelt.

2.4 Socket-Adressierung

Ein Socket wird zunächst ohne Namen bzw. Adresse erzeugt. Damit Prozesse den Socket adressieren können, müssen Sie dem Socket mit der Funktion *bind()* einen Namen (Adresse) entsprechend seiner Adressfamilie zuordnen (siehe Abschnitt „Einem Socket einen Namen zuordnen“ auf Seite 19). Danach können über den Socket Nachrichten empfangen werden.

2.4.1 Socket-Adressen verwenden

Bei Aufruf der Funktionen *bind()*, *connect()*, *getpeername()*, *getsockname()*, *recvfrom()*, *recvmsg()*, *sendto()* und *sendmsg()* wird als aktueller Parameter ein Zeiger auf einen Namen (Adresse) übergeben. Zuvor muss das Programm den Namen gemäß der Adress-Struktur der verwendeten Adressfamilie bereitstellen. Diese Adress-Struktur ist je nach verwendeter Adressfamilie unterschiedlich aufgebaut (siehe Abschnitt „Adress-Struktur sockaddr_in der Adressfamilie AF_INET“ auf Seite 14, Abschnitt „Adress-Struktur sockaddr_in6 der Adressfamilie AF_INET6“ auf Seite 15 und Abschnitt „Adress-Struktur sockaddr_un der Adressfamilie AF_UNIX“ auf Seite 15).

Vor der Parameterübergabe muss der Zeiger, der die Adresse übergibt, mit dem cast-Operator konvertiert werden und zwar vom Typ „Zeiger auf die Struktur der verwendeten Adressfamilie“ in den Typ „Zeiger auf *struct sockaddr*“. Die Struktur *sockaddr* ist die in den Socket-Funktionen verwendete allgemeine, domänen-unabhängige Adress-Struktur.

Im Folgenden werden die Adress-Strukturen für die Adressfamilien AF_INET, AF_INET6 und AF_UNIX beschrieben. Die Strukturen für Rechner-, Protokoll- und Service-Namen sind im Kapitel „Adressumwandlung bei SOCKETS(POSIX)“ auf Seite 43 dargestellt.

2.4.2 Adressierung mit Internet-Adressen

Bei SOCKETS(POSIX) werden sowohl IPv4- als auch IPv6-Adressen unterstützt. IPv4- und IPv6-Adressen unterscheiden sich in der Länge und werden deshalb durch unterschiedliche Adressfamilien identifiziert:

– AF_INET unterstützt die 4 byte lange IPv4-Internetadresse.

– AF_INET6 unterstützt die 16 byte lange IPv6-Internetadresse.

Struktur und Erscheinungsformen dieser Adressen sind im Handbuch „openNet Server V3.0 (BS2000/OSD)“ beschrieben. Detaillierte Erläuterungen zur IPv6-Funktionalität finden Sie außerdem im „IPv6 Einführung und Umstellhandbuch Stufe 1“.

2.4.2.1 Adress-Struktur sockaddr_in der Adressfamilie AF_INET

Bei der Adressfamilie AF_INET besteht ein Name aus einer Internet-Adresse und einer Portnummer. Für die Adressfamilie AF_INET verwenden Sie die Adress-Struktur *sockaddr_in*.
Die Struktur *sockaddr_in* ist in der Include-Datei <netinet/in.h> wie folgt deklariert:

```
struct sockaddr_in {
   sa_family_t      sin_family;      /* Adressfamilie */
   in_port_t        sin_port;        /* 16 bit Portnummer */
   struct in_addr   sin_addr;        /* 32 bit Internet-Adresse */;
   unsigned char    sin_zero[8];
   };

struct in_addr {
   in_addr_t s_addr;
   };
```

Mit den folgenden Anweisungen versorgen Sie eine Variable *server* vom Typ *struct sockaddr_in* mit einem Namen:

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

Ein Zeiger auf die Variable *server* kann nun als aktueller Parameter z.B. bei einem *bind()*-Aufruf übergeben werden, um den Namen an einen Socket zu binden:

```
bind (..., (struct sockaddr *)&server, ...)  /* bind()-Aufruf mit
                                                Typ-Konvertierung */
```

2.4.2.2 Adress-Struktur sockaddr_in6 der Adressfamilie AF_INET6

Bei der Adressfamilie AF_INET6 besteht ein Name aus einer 16 byte langen Internet-Adresse und einer Portnummer. Für die Adressfamilie AF_INET6 verwenden Sie die Adress-Struktur *sockaddr_in6*.

Die Struktur *sockaddr_in6* ist in der Include-Datei <netinet/in.h> wie folgt deklariert:

```
struct sockaddr_in6 {
sa_family_t sin6_family;    /* Adressfamilie AF_INET6 */
in_port_t sin6_port;        /* 16 bit Portnummer  */
uint32_t sin6_flowinfo
struct in6_addr sin6_addr;  /* IPv6-Adresse */
uint32_t sin6_scope_id;
};
```

Mit den folgenden Anweisungen können Sie eine Variable *server* vom Typ *struct sockaddr_in6* mit einem Namen versorgen:

```
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);
```

Ein Zeiger auf die Variable *server* kann nun als aktueller Parameter, z.B. bei einem *bind()*-Aufruf, übergeben werden, um den Namen an einen Socket zu binden:

```
bind(..., &server, ...)  /* bind()-Aufruf mit Typ-Konvertierung */
```

2.4.2.3 Adress-Struktur sockaddr_un der Adressfamilie AF_UNIX

Bei der Adressfamilie AF_UNIX besteht ein Name (Adresse) aus einem Pfadnamen. Für die Adressfamilie AF_UNIX verwenden Sie die Adress-Struktur *sockaddr_un*.

Die Struktur *sockaddr_un* ist in der Include-Datei <sys/un.h> wie folgt deklariert:

```
struct sockaddr_un {
   sa_family_t  sun_family;       /* Adressfamilie */
   char         sun_path[108];    /* Pfadname */
   };
```

Eine Variable *server* vom Typ *struct sockaddr_un* versorgen Sie z.B. durch folgende Anweisungen mit einem Namen:

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

Ein Zeiger auf die Variable *server* kann nun als aktueller Parameter z.B. bei einem *bind()*-Aufruf übergeben werden, um den Namen an einen Socket zu binden:

```
bind(..., (struct sockaddr *)&server, ...)  /* bind()-Aufruf mit
                                               Typ-Konvertierung */
```

2.5 Socket erzeugen

Ein Socket wird mit der Funktion *socket()* erzeugt:

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

Der Aufruf *socket()* erzeugt einen Socket in der Domäne *domain* mit dem Typ *type* und liefert einen Deskriptor (Integer-Wert) als Rückgabewert. Über diesen Deskriptor kann der neu erzeugte Socket in allen weiteren Aufrufen von Socket-Funktionen identifiziert werden.

Die Domänen sind als Konstanten in der Include-Datei <sys/socket.h> definiert. Unterstützt werden folgende Domänen:

– Internet-Kommunikationsdomäne AF_INET

– Internet-Kommunikationsdomäne AF_INET6

– rechnerlokale Kommunikationsdomäne AF_UNIX

Für *domain* geben Sie deshalb AF_INET, AF_INET6 oder AF_UNIX an.

Die Socket-Typen *type* sind ebenfalls in der Datei <sys/socket.h> definiert:

– Wenn Sie eine verbindungsorientierte Kommunikationsbeziehung über einen Stream-Socket aufbauen wollen, geben Sie SOCK_STREAM für *type* an.

– Wenn Sie eine verbindungslose Kommunikationsbeziehung über einen Datagramm-Socket aufbauen wollen, geben Sie SOCK_DGRAM für *type* an.

Mit der Angabe 0 für *protocol* spezifizieren Sie das Standardprotokoll:

– TCP beim Socket-Typ SOCK_STREAM

– UDP beim Socket-Typ SOCK_DGRAM

2.5.1 Socket in der Domäne AF_INET erzeugen

Der folgende Aufruf erzeugt einen Stream-Socket in der Internet-Domäne AF_INET:

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

In diesem Fall bietet das TCP-Protokoll die darunter liegende Kommunikationsunterstützung.

Der folgende Aufruf erzeugt einen Datagramm-Socket in der Internet-Domäne:

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

Das in diesem Fall verwendete UDP-Protokoll leitet die Datagramme ohne weitere Kommunikationsunterstützung an die darunter liegenden Netzdienste weiter.

2.5.2 Socket in der Domäne AF_INET6 erzeugen

Der folgende Aufruf erzeugt einen Stream-Socket in der IPv6-Internet-Domäne AF_INET6:

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

In diesem Fall bietet das TCP-Protokoll die darunter liegende Kommunikationsunterstützung.

Der folgende Aufruf erzeugt einen Datagramm-Socket in der IPv6-Internet-Domäne AF_INET6:

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

Das in diesem Fall verwendete UDP-Protokoll leitet die Datagramme ohne weitere Kommunikationsunterstützung an die darunter liegenden Netzdienste weiter.

2.6 Einem Socket einen Namen zuordnen

Ein mit *s=socket()* erzeugter Socket hat zunächst keinen Namen. Dem Socket muss also ein Name, d.h. eine lokale Adresse gemäß seiner Adressfamilie, zugeordnet werden. Erst danach ist der Socket adressierbar und es können Daten über ihn gesendet und empfangen werden. Mit der Funktion *bind()* binden Sie einen Namen an den Socket, d.h. Sie ordnen dem Socket eine lokale Adresse zu.

bind() rufen Sie wie folgt auf:

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

Je nach Adressfamilie (AF_INET, AF_INET6 oder AF_UNIX) ist der Name *name*, der dem Socket *s* zugeordnet wird, unterschiedlich aufgebaut.

– In der Kommunikationsdomäne AF_INET besteht *name* aus einer 4 byte langen IPv4-Adresse und einer Portnummer. *name* wird übergeben in einer Variablen vom Typ *struct sockaddr_in* (siehe Seite 14).

– In der Kommunikationsdomäne AF_INET6 besteht *name* aus einer 16 byte langen IPv6-Adresse und einer Portnummer. *name* wird übergeben in einer Variablen vom Typ *struct sockaddr_in6* (siehe Seite 15).

namelen enthält die Länge der Datenstruktur, die den Namen beschreibt.

2.6.1 bind()-Aufruf bei AF_INET

Bei AF_INET besteht *name* aus einer IPv4-Adresse und einer Portnummer. *name* wird übergeben in einer Variablen vom Typ *struct sockaddr_in* (siehe Seite 14).

Der folgende Programmauschnitt skizziert, wie einem Socket ein Name zugeordnet wird.

```
#include <sys/types.h>
#include <netinet/in.h>
 ...
struct sockaddr_in sin;
 ...
/* Hier müssen die Anweisungen stehen, die sin
   mit Internet-Adresse und Portnummer versorgen.*/
 ...
bind(s, (struct sockaddr *)&sin, sizeof sin);
```

2.6.2 bind()-Aufruf bei AF_INET6

Bei AF_INET6 besteht *name* aus einer IPv6-Adresse und einer Portnummer. *name* wird übergeben in einer Variablen vom Typ *struct sockaddr_in6* (siehe Seite 15).

Der folgende Programmauschnitt skizziert, wie einem Socket ein Name zugeordnet wird.

```
#include <sys/types.h>
#include <netinet/in.h>
 ...
struct sockaddr_in6 sin6;
 ...
/* Hier müssen die Anweisungen stehen, die sin6
   mit Internet-Adresse und Portnummer versorgen.*/
 ...
bind(s, (struct sockaddr *)&sin6, sizeof sin6);
```

2.6.3 Abhängigkeiten zu Portnummern

Bei der Wahl der Portnummer ist zu beachten:

– Portnummern kleiner als IPPORT_RESERVED (1024) sind für privilegierte Benutzer reserviert.

– Portnummern im Bereich von 1024 bis PRIVPORT# dürfen nicht mit fest vergebenen Portnummern für privilegierte Anwendungen übereinstimmen (siehe Abschnitt „Abhängigkeiten vom BS2000/OSD-Transportsystem BCAM“ auf Seite 316).

– Für einige Standardanwendungen gibt es feste Reservierungen bestimmter Portnummern. Diese weltweit gültige Zuordnung ist in der Datei */etc/inet/services* hinterlegt. Für lokale Netze ist eine Erweiterung dieser Datei zur Protokollierung belegter Portnummern möglich.

2.6.4 bind()-Aufruf bei AF_UNIX

Im Fall AF_UNIX besteht *name* aus einem Pfadnamen. Der Pfadname wird übergeben in einer Variablen vom Typ *struct sockaddr_un* (siehe Seite 15).

Der folgende Programmausschnitt skizziert, wie einem Socket ein Name zugeordnet wird:

```
#include <sys/types.h>
#include <sys/un.h>
 ...
struct sockaddr_un sun;
 ...
/* Hier müssen die Anweisungen stehen, die sun mit dem Pfadnamen versorgen.*/
 ...
bind(s, (struct sockaddr *)&sun, sizeof sun);
```

Der in der Komponente *sun.sun_path* anzugebende Pfadname wird durch *bind()* als Datei im Dateisystem erzeugt. Der Prozess, der *bind()* aufruft, muss daher die Schreibberechtigung für das Verzeichnis besitzen, in dem die Datei angelegt werden soll. Die Datei wird vom System nicht gelöscht. Deshalb sollte der Prozess die Datei löschen, wenn er sie nicht mehr benötigt.

2.6.5 Adressen mit Wildcards zuordnen (AF_INET, AF_INET6)

Wildcard-Adressen vereinfachen die lokale Adresszuordnung in den Internet-Domänen AF_INET und AF_INET6.

Internet-Adresse mit Wildcard zuordnen

Mit der Funktion *bind()* ordnen Sie einem Socket einen lokalen Namen (Adresse) zu (siehe Seite 19). Dabei können Sie an Stelle einer konkreten Internet-Adresse auch INADDR_ANY (bei AF_INET) oder IN6ADDR_ANY (bei AF_INET6) als Internet-Adresse angeben. INADDR_ANY und IN6ADDR_ANY sind als feste Konstanten in <netinet/in.h> definiert.

Wenn Sie einem Socket *s* mit *bind()* einen Namen zuordnen, dessen Internet-Adresse mit INADDR_ANY oder IN6ADDR_ANY spezifiziert ist, wirkt sich das folgendermaßen aus:

- Nachrichten empfangen:
  - Der mit INADDR_ANY gebundene Socket *s* kann Nachrichten über alle IPv4-Netzwerk-Schnittstellen seines Rechners empfangen. Somit kann der Socket *s* alle Nachrichten empfangen, die an die Portnummer von *s* und eine beliebige gültige IPv4-Adresse des Rechners adressiert sind, auf dem der Socket *s* liegt. Hat der Rechner beispielsweise die IPv4-Adressen 128.32.0.4 und 10.0.0.78, so kann eine Task, welcher der Socket *s* zugeordnet ist, Verbindungsanforderungen annehmen, die an 128.32.0.4 und 10.0.0.78 adressiert sind.
  - Der mit IN6ADDR_ANY gebundene Socket *s* kann Nachrichten über alle IPv4- bzw. IPv6-Netzwerk-Schnittstellen seines Rechners empfangen. Somit kann der Socket *s* alle Nachrichten empfangen, die an die Portnummer von *s* und eine beliebige gültige IPv4- bzw. IPv6-Adresse des Rechners adressiert sind, auf dem der Socket *s* liegt. Hat der Rechner beispielsweise die IPv4- bzw. IPv6-Adressen 128.32.0.4 und 3FFE:0:0:0:A00:6FF:FE08:9A6B, so kann eine Task, welcher der Socket *s* zugeordnet ist, Verbindungsanforderungen annehmen, die an 128.32.0.4 und 3FFE:0:0:0:A00:6FF:FE08:9A6B adressiert sind.
- Nachrichten versenden:
  - Der mit INADDR_ANY gebundene Socket *s* kann Nachrichten über eine IPv4-fähige Netz-Schnittstelle seines Rechners verschicken.
  - Der mit IN6ADDR_ANY gebundene Socket *s* kann Nachrichten über eine beliebige Netz-Schnittstelle seines Rechners verschicken.

  Somit kann der mit INADDR_ANY gebundene Socket *s* jeden beliebigen anderen Socket adressieren, der über eine IPv4-fähige Netzwerk-Schnittstelle des Rechners erreichbar ist, auf dem der Socket *s* liegt.
  Der mit IN6ADDR-ANY gebunde Socket *s* kann hingegen jeden anderen Socket adressieren, der über eine beliebige Netzwerk-Schnittstelle des Rechners erreichbar ist, auf dem der Socket s liegt.

Die folgenden Beispiele zeigen, wie eine Task ohne Angabe einer Internet-Adresse einen lokalen Namen an einen Socket binden kann. Die Task muss lediglich die Portnummer spezifizieren:

Bei 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);
```

Bei 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);
```

Portnummer mit Wildcard zuordnen

Ein lokaler Port kann unspezifiziert (Angabe 0) bleiben. In diesem Fall wählt das System für ihn eine passende Portnummer. Die folgenden Beispiele zeigen, wie eine Task einem Socket eine lokale Adresse zuordnet, ohne die lokale Portnummer zu spezifizieren:

Bei 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);
```

Bei 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 Automatische Adresszuordnung durch das System

Auch wenn einem Socket noch keine Adresse zugeordnet ist, können Sie für den Socket eine Funktion aufrufen, die einen gebundenen Socket voraussetzt (z.B. *connect()*, *sendto()* etc.). In diesem Fall führt das System für den betreffenden Socket einen impliziten *bind()*-Aufruf mit Wildcards für Internet-Adresse und Portnummer durch. Dazu wird der Socket mit INADDR_ANY auf alle IPv4-Adressen des Rechners und mit IN6ADDR_ANY auf alle IPv6- bzw. IPv4-Adressen gebunden und erhält eine freie Portnummer aus dem Bereich der nichtprivilegierten Portnummern.

2.7 Verbindungsorientierte Kommunikation

Miteinander kommunizierende Sockets werden über eine Zuordnung miteinander verbunden. In der Internet-Domäne besteht eine Zuordnung aus einer lokalen Adresse und einer lokalen Portnummer sowie einer fernen Adresse und einer fernen Portnummer.

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

Beim Einrichten eines Sockets müssen zunächst nicht beide Adressierungspaare angegeben werden. Der Aufruf *bind()* spezifiziert die lokale Hälfte der Zuordnung:

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

Die Aufrufe der nachfolgend in diesem Abschnitt vorgestellten Funktionen *connect()* und *accept()* vervollständigen die Socket-Zuordnung beim Verbindungsaufbau.
Der Verbindungsaufbau zwischen zwei Prozessen verläuft in der Regel asymmetrisch, wobei ein Prozess die Rolle des Clients und der andere die Rolle des Servers übernimmt.

2.7.1 Verbindungsanforderung durch den Client

Der Client fordert Services vom Server an, indem er mit der Funktion *connect()* eine Verbindungsanforderung zum Socket des Servers schickt. Auf der Seite des Clients veranlasst der Aufruf *connect()* den Aufbau einer Verbindung.

In der Internet-Domäne AF_INET verläuft eine Verbindungsanforderung nach folgendem Schema:

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

In der Internet-Domäne AF_INET6 verläuft eine Verbindungsanforderung nach folgendem Schema:

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

Der Parameter *server* übergibt die IPv4- bzw. IPv6-Adresse und die Portnummer des Servers, mit dem der Client kommunizieren möchte.
Falls dem Socket des Client-Prozesses zum Zeitpunkt des Aufrufs von *connect()* noch kein Name zugeordnet ist, sucht das System automatisch einen Namen aus und ordnet ihn dem Socket zu.

Wenn der Verbindungsaufbau nicht erfolgreich ist, wird ein Fehler-Code zurückgeliefert. Dies kann z.B. der Fall sein, wenn der Server noch nicht bereit ist, eine Verbindung anzunehmen (siehe folgender Abschnitt). Jedoch bleiben auch bei nicht erfolgreichem Verbindungsaufbau alle Namen erhalten, die vom System automatisch zugeordnet wurden.

2.7.2 Verbindungsannahme durch den Server

Wenn der Server bereit ist, seine speziellen Services anzubieten, ordnet er einem seiner Sockets den für den betreffenden Service festgelegten Namen (Adresse) zu. Um die Verbindungsanforderung eines Clients annehmen zu können, muss der Server außerdem die beiden folgenden Schritte durchführen:

1. Mit der Funktion *listen()* markiert der Server den Socket für eingehende Verbindungsanforderungen als „abhörend“. Danach hört der Server den Socket ab, d.h. er wartet passiv auf eine Verbindungsanforderung für diesen Socket. Ein beliebiger Partner kann nun zum Server Kontakt aufnehmen.

   Durch *listen()* wird das POSIX-Subsystem außerdem veranlasst, an den betreffenden Socket gerichtete Verbindungsanforderungen in eine Warteschlange zu stellen. Auf diese Weise geht normalerweise keine Verbindungsanforderung verloren, während der Server eine andere Verbindungsanforderung bearbeitet.

2. Mit *accept()* nimmt der Server die Verbindung für den als „abhörend“ markierten Socket an.

Nach der Verbindungsannahme mit *accept()* ist die Verbindung zwischen Client und Server aufgebaut und die Datenübertragung kann beginnen.

Der folgende Programmausschnitt skizziert die Verbindungsannahme durch den Server in der Internet-Domäne AF_INET:

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

Der folgende Programmausschnitt skizziert die Verbindungsannahme durch den Server in der Internet-Domäne AF_INET6:

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

Als erster Parameter beim Aufruf von *listen()* wird der Deskriptor *s* des Sockets übergeben, über den die Verbindung aufgebaut werden soll. Der zweite Parameter gibt an, wie viele Verbindungsanforderungen maximal in der Warteschlange auf die Annahme durch den Server-Prozess warten können. Derzeit werden maximal 50 wartende Verbindungsanforderungen unterstützt.

Als erster Parameter beim Aufruf von *accept()* wird der Deskriptor *s* des Sockets übergeben, über den die Verbindung aufgebaut werden soll. Nach Ausführung von *accept()* enthält der Parameter *from* die Adresse der Partneranwendung und *fromlen* enthält die Länge dieser Adresse. Bei der Annahme einer Verbindung mit *accept()* wird ein Deskriptor für einen neuen Socket erzeugt. Diesen Deskriptor liefert *accept()* als Ergebnis zurück. Über den neu erzeugten Socket können nun Daten ausgetauscht werden. Über den Socket *s* kann der Server weitere Verbindungen annehmen.

Ein Aufruf von *accept()* blockiert normalerweise, weil die Funktion *accept()* solange nicht zurückkehrt, bis eine Verbindung angenommen ist. Außerdem hat der Server-Prozess beim Aufruf von *accept()* keine Möglichkeit anzuzeigen, dass er Verbindungswünsche nur von einem oder mehreren bestimmten Partnern annehmen möchte. Deshalb muss der Server-Prozess darauf achten, woher die Verbindung kommt. Der Server-Prozess muss die Verbindung beenden, wenn er nicht mit einem bestimmten Client kommunizieren möchte.

Im Kapitel „Erweiterte Funktionen von SOCKETS(POSIX)“ auf Seite 51 ist näher beschrieben,

- wie ein Server-Prozess auf mehr als einem Socket Verbindungen annimmt,
- wie ein Server-Prozess verhindert, dass der Aufruf von *accept()* blockiert.

2.7.3 Datenübertragung bei verbindungsorientierter Kommunikation

Sobald eine Verbindung aufgebaut ist, können Daten übertragen werden. Wenn die Kommunikationsendpunkte der beiden Kommunikationspartner über das Adressierungspaar fest miteinander verbunden sind, kann ein Benutzerprozess Nachrichten senden oder empfangen, ohne jedes Mal das Adressierungspaar anzugeben.

Es gibt mehrere Funktionen zum Senden und Empfangen von Daten. Wahlweise können Sie die Funktionen *read()* und *write()* bzw. *readv()* und *writev()* verwenden:

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

Diese Funktionen gehören zum Grundumfang der POSIX-Schnittstelle. Sie sind beschrieben im Handbuch „C-Bibliotheksfunktionen (BS2000/OSD) für POSIX-Anwendungen“. Socket-spezifische Besonderheiten dieser Funktionen sind im Abschnitt „Verwendung von POSIX-Standardfunktionen für Sockets“ auf Seite 136 des vorliegenden Handbuchs beschrieben.

Alternativ können Sie die folgenden, socket-spezifischen Funktionen verwenden:

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

Die socket-spezifischen Funktionen sind ausführlich beschrieben im Abschnitt „Beschreibung der Funktionen“ auf Seite 80.

2.7.4 Beispiele für eine verbindungsorientierte Client-/Server-Kommunikation

Die beiden folgenden Programmbeispiele zeigen, wie eine Streams-Verbindung in der Internet-Domäne vom Client initiiert und vom Server angenommen wird.

Die Beispielprogramme sind nur für die Kommunikationsdomäne AF_INET ausgeführt, bei Änderungen entsprechend den Abschnitten „Socket-Adressierung“ auf Seite 13 und „Socket erzeugen“ auf Seite 17 gelten sie auch für die Domäne AF_INET6.

Beispiel 1: Initiieren einer Streams-Verbindung durch den Client

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

```
/*
 * Dieses Programm erzeugt einen Socket und initiiert eine Verbindung mit
 * dem in der Kommandozeile übergebenen Socket.
 * Über die Verbindung wird eine Nachricht gesendet.
 * Dann wird der Socket geschlossen und die Verbindung beendet.
 * Das Programm wird wie folgt aufgerufen:
 * programmname rechnername  portnummer
 */

main(argc, argv)
      int argc;
      char *argv[];
{
      int sock;
      struct sockaddr_in server;
      struct hostent *hp;

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

      /*
       * Verbindungsaufbau unter Verwendung des in der Kommandozeile
       * angegebenen Namens.
       */
      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);
}
```

Beispiel 2: Annehmen der Streams-Verbindung durch den Server

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

/*
 * Dieses Programm erzeugt einen Socket und geht dann in eine
 * Endlos-Schleife. Bei jedem Schleifen-Durchlauf nimmt es eine
 * Verbindung an und gibt Nachrichten von ihr aus.
 * Bricht die Verbindung ab oder wird eine Beendigungs-Nachricht
 * übergeben, nimmt das Programm eine neue Verbindung an.
 */

main()
{
      int sock, length;
      struct sockaddr_in server, client;
      int msgsock;
      char buf[1024];
      int rval;

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

/* Dem Socket wird unter Verwendung von Wildcards ein Name zugeordnet. */
      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);
      }
```

```
/* Zugehörige Portnummer herausfinden und ausgeben. */
      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));

/* Beginn der Annahme von Verbindungswünschen. */
      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);

      /*
       * Da dieses Programm in einer Endlos-Schleife läuft, wird der
       * Socket "sock" niemals explizit geschlossen.
       * Allerdings werden alle Sockets automatisch geschlossen,
       * wenn ein Prozess abgebrochen wird oder sein normales Ende erreicht.
       */

       exit(0);
}
```

2.8 Verbindungslose Kommunikation in AF_INET und AF_INET6

Neben der im vorhergehenden Abschnitt beschriebenen verbindungsorientierten Kommunikation wird in den Domänen AF_INET und AF_INET6 außerdem die verbindungslose Kommunikation über das UDP-Protokoll unterstützt.

Die verbindungslose Kommunikation wird über Datagramm-Sockets (SOCK_DGRAM) abgewickelt. Ein Datagramm-Socket bietet eine symmetrische Schnittstelle zum Datenaustausch über Datagramme. Anders als bei der verbindungsorientierten Kommunikation, wo Client und Server über eine feste Verbindung miteinander kommunizieren, wird bei der Übertragung von Datagrammen keine Verbindung aufgebaut. Stattdessen enthält jede Nachricht die Zieladresse.

Wie Datagramm-Sockets erzeugt werden, ist im Abschnitt „Socket erzeugen“ auf Seite 17 beschrieben. Wenn eine bestimmte lokale Adresse benötigt wird, muss vor der ersten Datenübertragung die Funktion *bind()* aufgerufen werden (siehe Seite 19). Andernfalls ordnet das System die lokale Internet-Adresse und/oder die Portnummer zu, wenn zum ersten Mal Daten gesendet werden (siehe Seite 24).

2.8.1 Datenübertragung bei verbindungsloser Kommunikation

Mit der Funktion *sendto()* senden Sie Daten von einem Socket zu einem anderen Socket:

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

Die Parameter *s*, *buf*, *buflen* und *flags* verwenden Sie genauso wie bei verbindungsorientierten Sockets. Die Zieladresse übergeben Sie mit *to* und die Länge der Adresse mit *tolen*. Bei Benutzung einer Datagramm-Schnittstelle werden dem Sender keine Fehler mitgeteilt. Verfügt das System lokal über die Information, dass eine Nachricht nicht übertragen werden kann (z.B. wenn ein Netz nicht erreichbar ist), liefert der Aufruf *sendto()* den Returnwert „-1“ und die globale Variable *errno* enthält den entsprechenden Fehler-Code.

Für den Empfang einer Nachricht über einen Datagramm-Socket verwenden Sie die Funktion *recvfrom()*:

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

Der Parameter *fromlen* enthält zu Beginn die Größe des Puffers *from*. Bei Rückkehr der Funktion *recvfrom()* gibt *fromlen* die Größe der Adresse des Sockets an, von dem das Datagramm empfangen wurde.

Wahlweise können Sie vor einem *sendto()*- bzw. *recvfrom()*-Aufruf mit *connect()* eine bestimmte Zieladresse für einen Datagramm-Socket festlegen. In diesem Fall führt ein Aufruf von *sendto()* bzw. *recvfrom()* zu folgendem Verhalten:

– Daten, die der Prozess mit *sendto()* ohne explizite Angabe einer Zieladresse abschickt, werden automatisch an den Partner mit der im *connect()*-Aufruf angegebenen Zieladresse gesendet.

– Ein Benutzer-Prozess erhält mit *recvfrom()* ausschließlich Daten vom Partner mit der im *connect()*-Aufruf spezifizierten Adresse.

Für einen Datagramm-Socket kann mit *connect()* zu einem bestimmten Zeitpunkt immer nur **eine** Zieladresse spezifiziert sein. Mit einem weiteren *connect()*-Aufruf können Sie jedoch eine andere Zieladresse für den Socket festlegen.

Ein *connect()*-Aufruf für einen Datagramm-Socket kehrt sofort zurück; das System speichert lediglich die Adresse des Kommunikationspartners.

2.8.2 Beispiele für eine verbindungslose Kommunikation

Die beiden folgenden Programmbeispiele zeigen, wie bei verbindungsloser Kommunikation Datagramme empfangen und gesendet werden.

Die Beispielprogramme sind nur für die Kommunikationsdomäne AF_INET dargestellt, bei Änderungen entsprechend den Abschnitten „Socket-Adressierung“ auf Seite 13 und „Socket erzeugen“ auf Seite 17 gelten sie auch für die Domäne AF_INET6.

Beispiel 1 : Datagramme empfangen

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

/*
 * Die Include-Datei <netinet/in.h> deklariert sockaddr_in wie folgt:
 *
 * struct sockaddr_in {
 *    short   sin_family;
 *    u_short sin_port;
 *    struct in_addr sin_addr;
 *    char    sin_zero[8];
 * };
 *
 * Dieses Programm erzeugt einen Socket, ordnet ihm einen Namen zu
 * und liest dann von dem Socket.
 */
```

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

      /* Erzeugen des Sockets, von dem gelesen werden soll. */
      sock = socket(AF_INET, SOCK_DGRAM, 0);
      if (sock < 0) {
              perror("opening datagram socket");
              exit(1);
      }
      /* Dem Socket unter Verwendung von Wildcards einen Namen zuordnen */
      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);
      }

      /* Zugehörige Portnummer herausfinden und ausgeben. */
      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));
       /* Lesen von dem 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);
}
```

Beispiel 2: Datagramme senden

```
#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 . . ."
/*
```

```
 * Dieses Programm sendet ein Datagramm an einen Empfänger, dessen Name über
 * die Argumente in der Kommandozeile übergeben wird.  Das Format des
 * Kommandos ist:
 * programmname rechnername portnummer
 */

main(argc, argv)
      int argc;
      char *argv[];
{
      int sock;
      struct sockaddr_in name;
      struct hostent *hp;
      /* Socket erzeugen, über den gesendet werden soll */
      sock = socket(AF_INET, SOCK_DGRAM, 0);
      if (sock < 0) {
              perror("opening datagram socket");
              exit(1);
      }

      /*
       * Den Namen des Sockets, an den gesendet werden soll, ohne die
       * Verwendung von Wildcards konstruieren. gethostbyname liefert
       * eine Struktur, die die Internet-Adresse des angegebenen Rechners
       * enthält.  Die Portnummer wird aus der Kommandozeile
       * übernommen.
       */
      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]));
      /* Nachricht senden. */
      if (sendto(sock, DATA, sizeof DATA , 0,
        (struct sockaddr *)&name, sizeof name) < 0)
              perror("sending datagram message");
      close(sock);
      exit(0);
}
```

2.9 Socket schließen

Wenn Sie einen Socket nicht länger benötigen, schließen Sie seinen Deskriptor mit der Funktion *close()*:

```
close(s);
```

Auch diese Funktion gehört zum Grundumfang der POSIX-Schnittstelle (siehe Seite 137 und Handbuch „C-Bibliotheksfunktionen (BS2000/OSD) für POSIX-Anwendungen“).

2.10 Ein-/Ausgabe-Multiplexen

Oft ist es sinnvoll, Ein- und Ausgaben über mehrere Sockets zu verteilen. Für diese Art des Ein-/Ausgabe-Multiplexens verwenden Sie die Funktionen *select()* oder *poll()*. Mit diesen Funktionen kann ein Programm mehrere Verbindungen gleichzeitig überwachen.

Der folgende Programmausschnitt skizziert die Verwendung von *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);
```

Der Aufruf von *select()* benötigt als Parameter drei Zeiger auf jeweils eine Bitmaske, die eine Menge von Socket-Deskriptoren repräsentiert:

– Anhand der mit *readmask* übergebenen Bitmaske prüft *select()*, von welchen Sockets Daten gelesen werden können.

– Anhand der mit *writemask* übergebenen Bitmaske prüft *select()*, auf welche Sockets Daten geschrieben werden können.

– Anhand der mit *exceptmask* übergebenen Bitmaske prüft *select()*, für welche Sockets eine noch nicht ausgewertete Ausnahmebedingung vorliegt.

Der Parameter *nfds* spezifiziert, wieviele Bits bzw. Deskriptoren überprüft werden sollen: *select()* überprüft in jeder Bitmaske die Bits 0 bis *nfds*-1.
Wenn Sie an einer der Informationen (Lesen, Schreiben oder noch nicht ausgewertete Ausnahmebedingungen) kein Interesse haben, übergeben Sie beim *select()*-Aufruf als entsprechenden Parameter den Null-Zeiger.

Die Bitmasken, die die Deskriptormengen repräsentieren, werden als Bitfelder in Integer-Reihungen abgespeichert. Die Größe der Bitfelder ist durch die Konstante FD_SETSIZE festgelegt. FD_SETSIZE ist in <sys/select.h> mit einem Wert definiert, der im Standardfall mindestens so groß ist wie die maximale Anzahl der vom System unterstützten Deskriptoren.

Mit Makros können Sie die Bitmasken bearbeiten. Insbesondere sollten Sie die Bitmasken vor der Bearbeitung auf 0 setzen. Die Makros zur Manipulation von Bitmasken sind erläutert auf Seite 152 bei der Funktionsbeschreibung von *select()*.

Mit dem Parameter *timeout* legen Sie einen Timeout-Wert fest, wenn der Auswahlvorgang nicht länger als eine vorbestimmte Zeit dauern soll. Wenn Sie mit *timeout* den Null-Zeiger übergeben, blockiert die Ausführung von *select()* auf unbestimmte Zeit.

Ein zyklisches Auswahlverhalten (Polling) veranlassen Sie, wenn Sie für *timeout* einen Zeiger auf eine *timeval*-Variable übergeben, deren Komponenten sämtlich auf den Wert 0 gesetzt sind.

Bei erfolgreicher Ausführung spezifiziert der Rückgabewert von *select()* die Anzahl der selektierten Deskriptoren. Die Bitmasken zeigen dann an,

– welche Deskriptoren zum Lesen bereit sind,
– welche Deskriptoren zum Schreiben bereit sind,
– bei welchen Deskriptoren noch nicht ausgewertete Ausnahmebedingungen vorliegen.

Wenn die Ausführung von *select()* wegen Timeout beendet wird, liefert *select()* den Wert 0 zurück. Die Bitmasken sind jedoch bereits verändert.
Wenn *select()* wegen eines Fehlers beendet wird, liefert *select()* den Wert -1 zurück mit dem entsprechenden Fehler-Code in *errno*. Die Bitmasken sind dann unverändert.

Nach der Ausführung von *select()* können Sie mit dem Makro-Aufruf FD_ISSET(*fd*, &*mask*) den Status eines Deskriptors *fd* überprüfen. Der Makro liefert einen Wert ungleich 0, wenn *fd* ein Element der Bitmaske *mask* ist, andernfalls den Wert 0.

Ob auf einem Socket *fd* Verbindungsanforderungen auf ihre Annahme durch *accept()* warten, überprüfen Sie anhand der Lesebereitschaft des Sockets *fd*.
Zu diesem Zweck rufen Sie *select()* und anschließend den Makro FD_ISSET (*fd*, &*mask*) auf. Liefert FD_ISSET einen Wert ungleich 0, so signalisiert dies die Lesebereitschaft des Sockets *fd*: Am Socket *fd* steht also eine Verbindungsanforderung an.

Beispiel: Verwenden von select() zum Überprüfen auf anstehende Verbindungsanforderungen

Mit dem folgenden Programmcode kann ein beliebiger Prozess Daten von zwei Sockets lesen. Der Timeout-Wert beträgt fünf Sekunden.
Das Beispielprogramm ist nur für AF_INET dargestellt, bei Änderungen entsprechend den Abschnitten „Socket-Adressierung“ auf Seite 13 und „Socket erzeugen“ auf Seite 17 gilt es auch für AF_INET6.

```
#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

/*
 * Dieses Programm überprüft mit select, ob jemand versucht, eine
 * Verbindung aufzubauen, und ruft dann accept auf.
 */

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

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

      /* Dem Socket unter Verwendung von Wildcards einen Namen zuordnen */
      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);
      }
```

```
/* Zugehörige Portnummer herausfinden und ausgeben */
      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));

    /* Beginn der Annahme von Verbindungen. */
      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 Zusammenspiel der Funktionen der SOCKETS-Schnittstelle

Die beiden folgenden Abbildungen veranschaulichen das Zusammenspiel der Funktionen der SOCKETS(POSIX)-Schnittstelle. Ausführlich beschrieben sind die einzelnen Funktionen im Abschnitt „Beschreibung der Funktionen“ auf Seite 80.

Bild 2 veranschaulicht das Zusammenspiel der Funktionen der SOCKETS(POSIX)-Schnittstelle bei Stream-Sockets (SOCK_STREAM) in den Internet-Domänen AF_INET und AF_INET6.

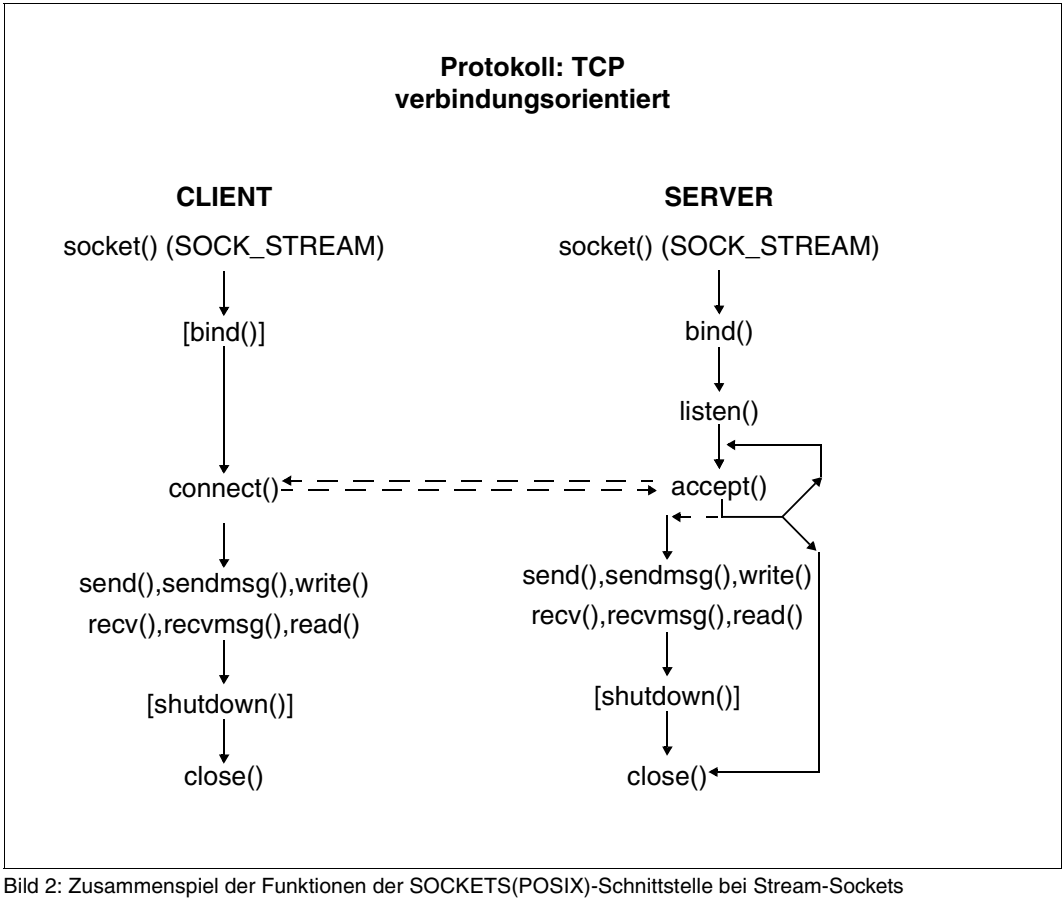

Bild 2: Zusammenspiel der Funktionen der SOCKETS(POSIX)-Schnittstelle bei Stream-Sockets

Bild 3 veranschaulicht das Zusammenspiel der Funktionen der SOCKETS(POSIX)-Schnittstelle bei Datagramm-Sockets (SOCK_DGRAM) in den Internet-Domänen AF_INET und AF_INET6.

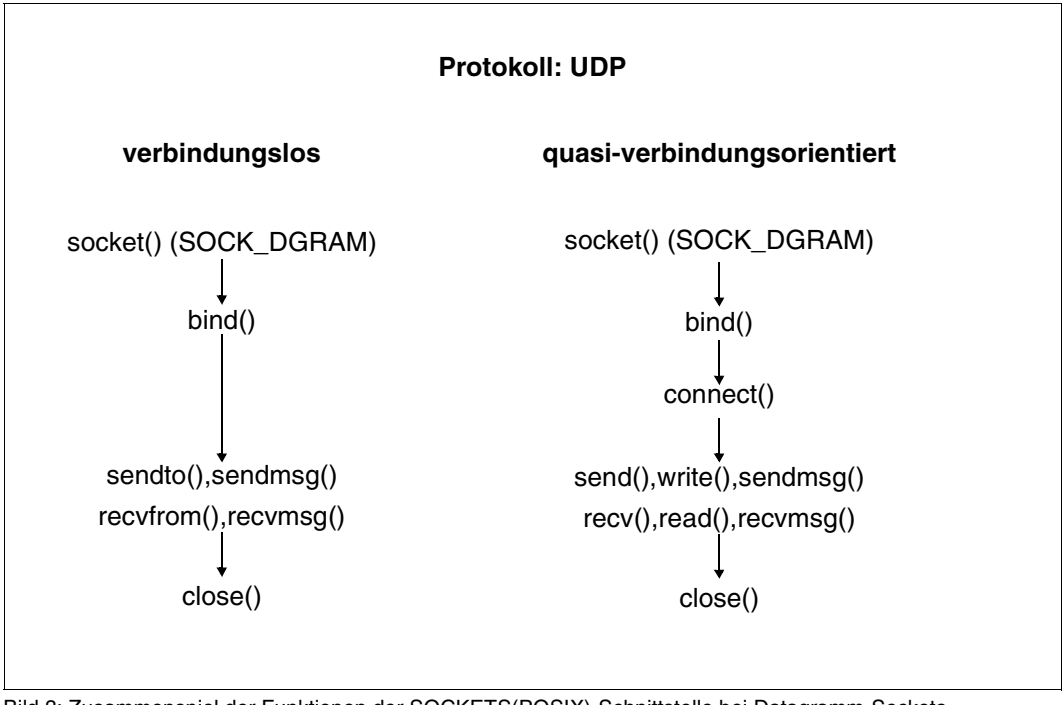

Bild 3: Zusammenspiel der Funktionen der SOCKETS(POSIX)-Schnittstelle bei Datagramm-Sockets

3 Adressumwandlung bei SOCKETS(POSIX)

Damit Prozesse über Sockets miteinander kommunizieren können, müssen Netzadressen ermittelt und erzeugt werden. Zu diesem Zweck stellt die SOCKETS-Bibliothek verschiedene Hilfsfunktionen und Makros bereit, die in diesem Kapitel vorgestellt werden.
Ausführlich beschrieben sind alle Hilfsfunktionen im Kapitel „Benutzerfunktionen von SOCKETS(POSIX)“ auf Seite 73.

Bevor Client und Server miteinander kommunizieren können, muss der Client zunächst den Service auf dem fernen System ermitteln. Die Ermittlung des betreffenden Service erfordert dabei folgende Stufen der Adressumwandlung:

1. Zur besseren Verständlichkeit sind auf Anwenderprogramm-Ebene einem Service und einem Rechner Namen zugeordnet, z.B. der Service *login* auf dem Rechner *Monet*.
2. Das System wandelt einen Service-Namen in eine Service-Nummer (Portnummer) um und einen Rechnernamen in eine Netzadresse (IPv4- bzw. IPv6-Adresse).
3. Anhand von Portnummer und IPv4- bzw. IPv6-Adresse ermittelt das System die Route zu dem Rechner, auf dem der Service angeboten wird.

Es ist nicht sinnvoll, dass anhand der Rechnernamen der Standort, also die physikalische Adresse eines Rechners erkannt wird. Vielmehr ist es Aufgabe von tieferliegenden Netzdiensten, den Standort eines Rechners zum Zeitpunkt herauszufinden, zu dem ein anderer Rechner mit ihm kommunizieren möchte. Auf diese Weise ist es möglich, den physikalischen Standort eines Rechners zu verändern, ohne dass dies Auswirkungen auf die Adressierung durch seine Kommunikationspartner hat.

Folgende Umwandlungsfunktionen werden angeboten:

– Rechnernamen zu Netzadressen, Netzadressen zu Rechnernamen
– Netznamen zu Netznummern
– Protokollnamen zu Protokollnummern
– Service-Namen zu Portnummern und dem zuständigen Protokoll zur Kommunikation mit dem Server

Wenn Sie eine dieser Funktionen verwenden wollen, müssen Sie die Datei <netdb.h> includieren.
Programmbeispiele, die die Verwendung der nachfolgend beschriebenen Umwandlungs-Funktionen zeigen, finden Sie im Kapitel „Client-/Server-Modell bei SOCKETS(POSIX)“ auf Seite 61.

3.1 Rechnernamen in Netzadressen umwandeln und umgekehrt

Für die Umwandlung von Rechnernamen in Netzadressen und umgekehrt gibt es in den Adressfamilien AF_INET und AF_INET6 spezielle Socket-Funktionen.

Socket-Funktionen zur Adressumwandlung in den Adressfamilien AF_INET und AF_INET6

Die Funktion *getipnodebyname()* wandelt einen Rechnernamen in eine IPv4-Adresse oder in eine IPv6-Adresse um. Beim Aufruf von *getipnodebyname()* wird ein Rechnername übergeben.

Die Funktion *getipnodebyaddr()* wandelt eine IPv4- oder IPv6-Adresse in einen Rechnernamen um. Beim Aufruf von *getipnodebyaddr()* wird eine IPv4- oder IPv6-Adresse übergeben.

Die Funktion *inet_ntop()* konvertiert eine Internet-Rechneradresse in eine Zeichenkette. Diese Zeichenkette wird wie folgt zurückgeliefert:

– bei AF_INET6 in der sedezimalen Doppelpunkt-Notation
– bei AF_INET in der dezimalen Punkt-Notation

Die Funktion *inet_pton()* konvertiert eine abdruckbar dargestellte Internet-Rechneradresse

– von einer Zeichenkette in der dezimalen Punkt-Notation in eine binäre IPv4-Adresse (AF_INET).
– von einer Zeichenkette in der sedezimalen Doppelpunkt-Notation in eine binäre IPv6-Adresse (AF_INET6).

Bei AF_INET6 wird die Kurzschreibweise mit zwei aufeinander folgenden Doppelpunkten „::“ nicht unterstützt.

Socket-Funktionen zur Adressumwandlung, die nur in AF_INET unterstützt werden

Die Funktion *gethostbyname()* wandelt einen Rechnernamen in eine IPv4-Adresse um. Beim Aufruf von *gethostbyname()* wird ein Rechnername übergeben.

Die Funktion *gethostbyaddr()* wandelt eine IPv4-Adresse in einen Rechnernamen um. Beim Aufruf von *gethostbyaddr()* wird eine IPv4-Adresse übergeben.

gethostbyname() und *gethostbyaddr()* liefern als Ergebnis einen Zeiger auf ein Objekt vom Datentyp *struct hostent.*

Die Struktur *hostent* ist in <netdb.h> wie folgt deklariert:

```
struct hostent {
     char *h_name;             /* offizieller Rechnername */
     char **h_aliases;         /* Alias-Liste */
     int  h_addrtype;          /* Adresstyp */
     int  h_length;            /* Länge der Adresse (in Bytes) */
     char **h_addr_list;       /* Liste von Adressen für den Rechner, */
                               /* terminiert durch den Null-Zeiger */
};
#define h_addr h_addr_list[0]  /* erste Adresse, Netz-Byte-Reihenfolge */
```

Das von *gethostbyname()* und *gethostbyaddr()* zurückgelieferte *hostent*-Objekt enthält immer folgende Informationen:

– offizieller Name des Rechners
– Liste der alternativen Namen (Aliases) des Rechners
– Adresstyp (Domäne)
– mit dem Null-Zeiger abgeschlossene Liste von Adressen variabler Länge

Die Adressliste wird benötigt, weil ein Rechner möglicherweise viele Adressen hat, die alle demselben Rechnernamen zugeordnet sind. Die Definition von *h_addr* gewährleistet Rückwärts-Kompatibilität und ist definiert als die erste Adresse in der Adressliste der Struktur *hostent*.

Die Funktion *inet_ntoa()* konvertiert eine IPv4-Rechneradresse in eine Zeichenkette gemäß der üblichen Internet-Punktschreibweise.

3.2 Protokollnamen in Protokollnummern umwandeln

Die Funktion *getprotobyname()* wandelt einen Protokollnamen in eine Protokollnummer um. Beim Aufruf von *getprotobyname()* wird der Protokollname übergeben.

getprotobyname() liefert als Ergebnis einen Zeiger auf ein Objekt vom Datentyp *struct protoent*.
Die Struktur *protoent* ist wie folgt deklariert:

```
struct protoent {
     char *p_name;          /* offizieller Protokollname */
     char **p_aliases;      /* Alias-Liste */
     int  p_proto;          /* Protokollnummer */
};
```

3.3 Service-Namen in Portnummern umwandeln und umgekehrt

Von einem Service wird erwartet, dass er sich an einem bestimmten Port befindet und ein einziges Kommunikationsprotokoll verwendet. Diese Sicht ist innerhalb der Internet-Domäne konsistent, gilt aber nicht in anderen Netz-Architekturen. Außerdem kann ein Service an mehreren Ports vorhanden sein. In diesem Fall müssen Bibliotheksfunktionen der höheren Schichten weitergeleitet oder erweitert werden.

Die Funktion *getservbyname()* wandelt einen Service-Namen in eine Portnummer um. Beim Aufruf von *getservbyname()* wird der Service-Name und optional der Name eines qualifizierenden Protokolls übergeben.
Die Funktion *getservbyport()* wandelt eine Portnummer in einen Service-Namen um. Beim Aufruf von *getservbyport()* wird die Portnummer und optional der Name eines qualifizierenden Protokolls übergeben.

getservbyname() und *getservbyport()* liefern als Ergebnis einen Zeiger auf ein Objekt vom Datentyp *struct servent*.
Die Struktur *servent* ist wie folgt deklariert:

```
struct servent {
     char *s_name;          /* offizieller Name des Service */
     char **s_aliases;      /* Alias-Liste */
     int  s_port;           /* Nummer des Ports, auf dem der Service liegt */
     char *s_proto;         /* verwendetes Protokoll */
};
```

Beispiel

Der folgende Programmcode liefert die Portnummer des Service *telnet*, der das TCP-Protokoll verwendet:

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

3.4 Byte-Reihenfolge umwandeln

Wenn Sie die zuvor beschriebenen Funktionen für die Adressumwandlung verwenden, werden Sie in einem Internet-Anwenderprogramm Adressen selten direkt behandeln müssen. Sie können dann Services weitgehend netzunabhängig entwickeln. Ein Rest von Netzabhängigkeit bleibt jedoch bestehen, da in einem Anwenderprogramm die IPv4- bzw. IPv6-Adresse angegeben werden muss, wenn einem Service bzw. einem Socket ein Name zugeordnet wird.

Neben den Bibliotheksfunktionen für die Umwandlung von Namen in Adressen gibt es auch Makros, die die Behandlung von Namen und Adressen vereinfachen.

In einigen Architekturen sind Rechner-Byte-Reihenfolge und Netz-Byte-Reihenfolge unterschiedlich. Folglich müssen Programme manchmal die Byte-Reihenfolge verändern. Die in Tabelle 1 zusammengefassten Makros setzen Bytes und Integers von Rechner-Byte-Reihenfolge in Netz-Byte-Reihenfolge um und umgekehrt.

Aufruf	**Bedeutung**
htonl(*val*)	32-bit-Felder von Rechner- in Netz-Byte-Reihenfolge umwandeln
htons(*val*)	16-bit-Felder von Rechner- in Netz-Byte-Reihenfolge umwandeln
ntohl(*val*)	32-bit-Felder von Netz- in Rechner-Byte-Reihenfolge umwandeln
ntohs(*val*)	16-bit-Felder von Netz- in Rechner-Byte-Reihenfolge umwandeln

Tabelle 1: Bibliotheks-Makros für die Umwandlung von Byte-Reihenfolgen

Die Makros für die Umwandlung von Byte-Reihenfolgen werden benötigt, weil das Betriebssystem die IPv4-Adressen in Netz-Byte-Reihenfolge erwartet. Die Bibliotheksfunktionen, die Netzadressen zurückliefern, liefern diese in Netz-Byte-Reihenfolge, sodass sie einfach in die Strukturen, die dem System zur Verfügung stehen, kopiert werden können. Sie sollten deshalb nur beim Interpretieren von Netzadressen auf Probleme mit Byte-Reihenfolgen treffen.

Bei IPv6-Adressen gibt es per Definition keinen Unterschied zwischen Rechner-Byte-Reihenfolge und Netz-Byte-Reihenfolge. Deshalb gibt es auch keine entsprechenden Umwandlungsfunktionen.

Im BS2000/OSD sind Rechner-Byte-Reihenfolge und Netz-Byte-Reihenfolge identisch. Deshalb sind die in Tabelle 1 aufgelisteten Makros als Null-Makros (Makros ohne Inhalt) definiert. Für die Erstellung portabler Programme ist die Verwendung der Makros jedoch dringend zu empfehlen.

3.5 Beispiel zur Adressumwandlung

Der nachfolgend dargestellte Client-Programmcode des *remote login* demonstriert die Adressumwandlung.

```
#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);
}
```

Das Beispielprogramm ist nur für die Kommunikationsdomäne AF_INET dargestellt. Es gilt auch für die Domäne AF_INET6, wenn Sie folgende Änderungen durchführen:

– *server* ist vom Typ *struct sockaddr_in6*

– *struct sockaddr_in6* wird aus der Socket-Funktion *getipnodebyname* versorgt (nicht aus *gethostbyname* wie in AF_INET)

Eine detaillierte Beschreibung dazu finden Sie in den Abschnitten „Socket-Adressierung“ auf Seite 13 und „Socket erzeugen“ auf Seite 17.

4 Erweiterte Funktionen von SOCKETS(POSIX)

In den meisten Fällen werden die in den vorhergehenden Kapiteln beschriebenen Techniken für die Entwicklung verteilter Anwendungen ausreichen. Gelegentlich kann es jedoch erforderlich sein, zusätzlich auf folgende Leistungsmerkmale von SOCKETS(POSIX) zurückzugreifen:

- nicht-blockierende Sockets
- Broadcast-Nachrichten
- Socket-Optionen
- Multicast-Nachrichten
- interrupt-gesteuerte Socket-Ein-/Ausgabe

4.1 Nicht-blockierende Sockets

Bei nicht-blockierenden Sockets werden die Funktionen *accept()* und *connect()* sowie alle Ein-/Ausgabe-Funktionen abgebrochen, wenn sie nicht sofort ausgeführt werden können. Die betreffende Funktion liefert dann einen Fehlercode zurück. Im Gegensatz zu gewöhnlichen Sockets verhindern nicht-blockierende Sockets also, dass ein Prozess unterbrochen wird, weil er auf die Beendigung von *accept()*, *connect()* oder Ein-/Ausgabe-Funktionen warten muss. Einen mit *s=socket()* erzeugten Socket können Sie mit der Funktion *fcntl()* wie folgt als nicht-blockierend markieren:

```
#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);
}
 ...
```

Die Funktion *fcntl()* gehört zum Grundumfang der POSIX-Schnittstelle. *fcntl()* ist beschrieben auf Seite 138 sowie im Handbuch „C-Bibliotheksfunktionen (BS2000/OSD) für POSIX-Anwendungen“.

Wenn auf nicht-blockierenden Sockets die Funktionen *accept()*, *connect()* oder Ein-/Ausgabe-Funktionen ausgeführt werden, sollten Sie sorgfältig auf den Fehler EWOULDBLOCK achten. EWOULDBLOCK wird in der globalen Variablen *errno* abgelegt und tritt auf, wenn auf einem nicht-blockierenden Socket eine normalerweise blockierende Funktion ausgeführt wird.

Die Funktionen *accept()*, *connect()* sowie alle Schreib- und Leseoperationen können den Fehlercode EWOULDBLOCK liefern. Daher sollten Prozesse auf die Behandlung solcher Returnwerte vorbereitet sein: Auch wenn z.B. die Funktion *send()* nicht vollständig ausgeführt wird, kann es bei Stream-Sockets dennoch sinnvoll sein, wenigstens einen Teil der Schreiboperationen auszuführen. In diesem Fall berücksichtigt *send()* nur die Daten, die sofort gesendet werden können. Der Returnwert zeigt die Menge der bereits gesendeten Daten an.

4.2 Broadcast-Nachrichten (AF_INET)

Bei der Verwendung eines Datagramm-Sockets ist es möglich, Broadcast-Pakete an viele mit dem System verbundene Netze zu senden. Das Netz selbst muss Broadcasts unterstützen, da das System keine Simulation von Broadcasts in der Software unterstützt. Broadcast-Nachrichten können eine hohe Netzlast erzeugen, da sie jeden Rechner im Netz zwingen, sie zu bedienen.

Broadcasting wird nur in der AF_INET-Adressfamilie angeboten, da es in IPv6 keinen Broadcast-Mechanismus gibt.

Broadcasting wird in der Regel aus einem der beiden folgenden Gründe verwendet:

– In einem lokalen Netz soll eine Ressource gefunden werden, deren Adresse zunächst unbekannt ist.

– Wichtige Funktionen, wie z.B. die Routing-Funktion, wollen Informationen an alle erreichbaren Rechner senden.

Um eine Broadcast-Nachricht zu senden, muss zunächst ein Datagramm-Socket erzeugt werden:

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

Dann erhält der Socket die Markierung, dass er Broadcast-Nachrichten senden darf:

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

Schließlich wird dem Socket eine Portnummer zugeordnet:

```
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);
```

Die Zieladresse der Nachricht, die als Broadcast-Nachricht gesendet werden soll, hängt vom Netz oder den Netzen ab, auf dem oder denen die Nachricht gesendet wird. Die Internet-Domäne unterstützt eine Kurzbezeichnung für Broadcast im lokalen Netz, nämlich die Adresse INADDR_BROADCAST (definiert in <netinet/in.h>).

Um die Liste von Adressen für alle per Broadcast erreichbaren Rechner festzulegen, werden Kenntnisse über die Topologie der Netze benötigt, an denen der Rechner angeschlossen ist.

Da diese Informationen rechnerunabhängig gehalten werden sollten und möglicherweise nicht nachzuvollziehen sind, unterstützt das BS2000/OSD eine Methode, mit der diese Informationen den Datenstrukturen des Systems entnommen werden können.

Der *ioctl()*-Aufruf SIOCGIFCONF liefert für ipv4 die Netzwerk-Konfiguration eines Rechners als einfache Struktur *ifconf*. *ifconf* enthält einen Datenbereich, der aus einer Liste von *ifreq*-Strukturen besteht. In dieser Liste gibt es eine *ifreq*-Struktur für jede Adressdomäne, die von jedem Netzwerk-Interface unterstützt wird, über das der Rechner angeschlossen ist.

Die Struktur *ifreq* ist in <net/if.h> wie folgt deklariert:

```
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 */
};
```

Der folgende Programmcode liefert die IPv4-Interface-Konfiguration:

```
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) {
  ...
}
```

Nach Aufruf von *ioctl()* enthält *buf* eine Liste von *ifreq*-Strukturen: eine *ifreq*-Struktur für jedes Netz, an das der Rechner angeschlossen ist. Diese *ifreq*-Strukturen können zunächst nach dem Interface-Namen und dann nach den unterstützten Adressfamilien geordnet werden. Die Komponente *ifc.ifc_len* spezifiziert die Größe des Speicherbereichs (in Bytes), den die *ifreq*-Strukturen benötigen.

Jede *ifreq*-Struktur enthält einen Satz von Interface-Flags. Diese Interface-Flags geben an, ob das zu dem betreffenden Interface gehörende Netz aktiv ist oder nicht, oder ob es sich um ein Broadcast-Netz handelt usw.

Die im *ioctl()*-Aufruf angegebene Steuerfunktion SIOCGIFFLAGS sucht diese Flags für ein Interface, das durch eine *ifreq*-Struktur wie folgt spezifiziert ist:

```
struct ifreq *ifr;

ifr = ifc.ifc_req;

for (n=ifc.ifc_len/sizeof (struct ifreq);
      --n >= 0; ifr++) {
      /*
       * Es ist darauf zu achten, kein Interface zu verwenden, das einer
       * anderen Adressdomäne zugeordnet ist, als der gewünschten.
       */
      if (ifr->ifr_addr.sa_family != AF_INET)
            continue;
      if (ioctl(s, SIOCGIFFLAGS, (char *) ifr) < 0) {
          ...
      }
      /*
       * Triviale Fälle werden übergangen.
       */
      if ((ifr->ifr_flags & IFF_UP) == 0 ||
          (ifr->ifr_flags & IFF_LOOPBACK) ||
          (ifr->ifr_flags &
          (IFF_BROADCAST) == 0)
              continue;
}
```

Sobald die Flags ermittelt sind, muss die Broadcast-Adresse festgestellt werden. In Broadcast-Netzen verwenden Sie hierfür SIOCGIFBRDADDR in einem *ioctl()*-Aufruf.

```
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);
}
```

Nachdem durch Ausführen des *ioctl()*-Aufrufs die Broadcast- oder Zieladresse (jetzt in *dst*) bekannt ist, können Sie die Funktion *sendto()* wie folgt aufrufen:

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

In der oben gezeigten Schleife wird *sendto()* für jedes mit dem Rechner verbundene Interface durchlaufen, das Broadcast- oder Punkt-zu-Punkt-Adressierung unterstützt. Wenn ein Prozess Broadcast-Nachrichten nur an ein bestimmtes Netz schicken möchte, kann ein ähnlicher Programmcode verwendet werden. Es ist jedoch notwendig, dass in der Schleife die richtige Zieladresse gefunden wird.

Da dem sendenden Datagramm-Socket ein Name zugeordnet sein muss, enthalten empfangene Broadcast-Nachrichten immer Adresse und Portnummer des Senders.

Mit dem BCAM-Kommando BCOPTION kann gesteuert werden, ob ein Rechner Broadcast-Nachrichten empfangen darf (siehe Handbuch „openNet Server V3.0 (BS2000/OSD)“). Mit den Socket-Funktionen *setsockopt()* und *getsockopt()* kann diese Einstellung weder beeinflusst noch ermittelt werden. Deshalb ist bei Nutzung des Broadcast-Mechanismus sicherzustellen, dass bei den relevanten Rechnern diese Option gesetzt ist.

4.3 Socket-Optionen

Mit den Funktionen *setsockopt()* und *getsockopt()* setzen Sie verschiedene Optionen für Sockets bzw. fragen Sie die aktuellen Werte ab.

Optionen setzen Sie z.B. zu folgenden Zwecken:

– einen Socket für das Senden von Broadcast-Nachrichten kennzeichnen

– einen Socket veranlassen, mit dem Verbindungsabbau zu warten, bis alle Daten übertragen sind

Die allgemeine Form der Aufrufe lautet:

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

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

s bezeichnet den Socket, für den die Option gesetzt bzw. abgefragt wird.
level gibt die Protokollebene an, der die Option angehört. Normalerweise ist dies die Socket-Ebene, die durch die symbolische Konstante SOL_SOCKET angezeigt wird. SOL_SOCKET ist definiert in <sys/socket.h>.

In *optname* wird die Socket-Option angegeben. Die Socket-Option ist ebenfalls eine in <sys/socket.h> definierte symbolische Konstante.

optval ist ein Zeiger auf den Wert der Option. Bei *setsockopt()* schalten Sie mit *optval* die Option *optname* für den Socket *s* ein oder aus. Bei *getsockopt()* informiert Sie *optval*, ob die Option *optname* für den Socket *s* ein- oder ausgeschaltet ist.

optlen gibt die Länge des Option-Werts wie bei *setsockopt()* an.
optlenp ist ein Zeiger, der bei Aufruf von *getsockopt()* die Größe des Speicherplatzes angibt, auf den *optval* zeigt. Nach Rückkehr von *getsockopt()* spezifiziert *optlenp* die aktuelle Länge des in **optval* zurückgelieferten Option-Werts.

4.4 Multicast-Nachrichten (AF_INET)

Bei Datagramm-Sockets ist es möglich, Multicast-Nachrichten zu senden oder zu empfangen.

In der Adressfamilie AF_INET wird die Übertragung von Multicast-Nachrichten durch folgende Socket-Optionen unterstützt:

- IP_ADD_MEMBERSHIP: Anmeldung an eine Multicast-Gruppe
- IP_DROP_MEMBERSHIP: Abmeldung von einer Multicast-Gruppe
- IP_MULTICAST_TTL: Mulicast-Hop-Limit zeigen oder festlegen

4.5 Interrupt-gesteuerte Socket-Ein-/Ausgabe

Das SIGIO-Signal informiert einen Prozess, sobald ein Socket (oder allgemein ein Dateideskriptor) über Daten verfügt, die gelesen werden können.

Damit ein Prozess auf das SIGIO-Signal reagieren kann, müssen Sie im zu Grunde liegenden Programmcode folgende Vorkehrungen treffen:

1. Definieren Sie eine Funktion zur Signalbehandlung. Verwenden sie hierfür die Funktion *sigaction()* (siehe Handbuch „C-Bibliotheksfunktionen (BS2000/OSD) für POSIX-Anwendungen“).
2. Setzen Sie entweder die Prozessnummer oder die Prozessgruppennummer, damit die eigene Prozessnummer bzw. Prozessgruppennummer über wartende Eingaben benachrichtigt werden kann. Setzen Sie Prozessnummer oder Prozessgruppennummer mit der Funktion *fcntl()*. Die voreingestellte Prozessgruppe eines Sockets ist Gruppe 0.
3. Ermöglichen Sie mit einem weiteren *fcntl()*-Aufruf die asynchrone Benachrichtigung über wartende Ein-/Ausgabe-Anforderungen.

Der folgende Programm-Code skizziert, wie ein Prozess für den Empfang von SIGIO-Signalen vorbereitet wird. Durch Aufruf einer benutzerdefinierten Funktion *signal()* zur SIGIO-Behandlung wird der Prozess asynchron benachrichtigt, wenn Daten gelesen bzw. geschrieben werden können.

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

/* Stellt den Prozess zum Empfang der SIGIO-Signale ein. */

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

/* Erlaubt den Empfang asynchroner E/A Signale */

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

5 Client-/Server-Modell bei SOCKETS(POSIX)

Das am häufigsten verwendete Modell bei der Entwicklung von verteilten Anwendungen ist das Client-/Server-Modell. Im Client-/Server-Modell fordern Client-Anwendungen Dienste von einer Server-Anwendung an. Dies impliziert die bereits im Kapitel „Grundlagen von SOCKETS(POSIX)“ auf Seite 7 beschriebene Asymmetrie beim Aufbau von Verbindungen zwischen Client und Server. Das vorliegende Kapitel geht anhand von Beispielen näher auf die Interaktion zwischen Client und Server ein und zeigt einige Probleme und deren Lösung bei der Entwicklung von Client-/Server-Anwendungen.

Bevor ein Service gewährt und angenommen werden kann, erfordert die Kommunikation zwischen Client und Server eine für beide Seiten bekannte Menge von Übereinkünften. Diese Übereinkünfte sind in einem Protokoll festgelegt, das auf beiden Seiten einer Verbindung implementiert sein muss. Je nach Situation kann das Protokoll symmetrisch oder asymmetrisch sein. In einem symmetrischen Protokoll können beide Seiten die Server- oder die Client-Rolle übernehmen. Bei einem asymmetrischen Protokoll wird die eine Seite unveränderlich als der Server angesehen und die andere Seite unveränderlich als der Client.

Unabhängig davon, ob ein symmetrisches oder ein asymmetrisches Protokoll für einen Service verwendet wird, gibt es beim Zugriff auf einen Service einen Client und einen Server.

Die folgenden Abschnitte beschreiben:

- den verbindungsorientierten Server
- den verbindungsorientierten Client
- den verbindungslosen Server
- den verbindungslosen Client

5.1 Verbindungsorientierter Server

Normalerweise wartet der Server an einer bekannten Adresse auf Service-Anforderungen. Der Server verhält sich solange inaktiv, bis ein Client eine Verbindungsanforderung an die Adresse des Servers schickt. Zu diesem Zeitpunkt „erwacht“ der Server und bedient den Client, indem er die passenden Aktionen für die Anforderung des Clients ausführt.
Auf den Server wird über die bekannte Internet-Adresse zugegriffen.

Die Programmierung der Programm-Hauptschleife wird im folgenden Beispiel gezeigt. Im Beispielprogramm benutzt der Server die folgenden Funktionen der Socket- bzw. POSIX-Schnittstelle:

- *socket()*: Socket erzeugen
- *bind()*: einem Socket einen Namen zuordnen
- *listen()*: einen Socket auf Verbindungsanforderungen „abhören“
- *accept()*: eine Verbindung auf einem Socket annehmen
- *recv()*: Daten von einem Socket lesen
- *close()*: Socket schließen

Beispiel: Verbindungsorientierter 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;

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

```
    /* Dem Socket einen Namen zuordnen */
    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);
       }

    /* Beginn mit der Annahme von Verbindungsanforderungen */
    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);
}
```

Der Server erzeugt mit der Funktion *socket()* einen Kommunikationsendpunkt (Socket) und den zugehörigen Deskriptor. Mit der Funktion *bind()* wird dem Server-Socket eine definierte Portnummer zugeordnet. Über diese Portnummer kann er im Netz adressiert werden.

Mit der Funktion *listen()* stellt der Server fest, ob Verbindungsanforderungen anstehen. Verbindungsanforderungen kann der Server mit *accept()* annehmen. Der Returnwert von *accept()* wird überprüft, um sicherzustellen, dass die Verbindung erfolgreich aufgebaut wurde. Sobald die Verbindung aufgebaut ist, werden die Daten mit der Funktion *recv()* vom Socket gelesen. Mit der Funktion *close()* schließt der Server den Socket.

Das Beispielprogramm ist nur für die Kommunikationsdomäne AF_INET dargestellt.
Es gilt auch für die Domäne AF_INET6, wenn Sie die entsprechenden Änderungen durchführen. Eine detaillierte Beschreibung dazu finden Sie in den Abschnitten „Socket-Adressierung“ auf Seite 13 und „Socket erzeugen“ auf Seite 17.

5.2 Verbindungsorientierter Client

Die Seite des Clients wurde bereits im Beispiel auf Seite 62 dargestellt. Im Programmcode können Sie deutlich die separaten, asymmetrischen Rollen von Client und Server erkennen. Die Server wartet als passive Instanz auf Verbindungsanforderungen des Clients, während der Client als aktive Instanz eine Verbindung anstößt.

Im Folgenden werden die Schritte näher betrachtet, die vom Client-Prozess des *remote login* durchgeführt werden. Wie im Server-Prozess muss zunächst die Service-Definition für ein *remote login* herausgefunden werden.

Im Beispielprogramm verwendet der Client die folgenden Funktionen der Socket- bzw. POSIX-Schnittstelle:

- *socket()*: Socket erzeugen
- *setsockopt()*: Optionen für den Socket setzen
- *gethostbyname()*: Rechnereintrag abfragen
- *connect()*: auf dem Socket eine Verbindung anfordern
- *send()*: Daten auf den Socket schreiben
- *close()*: Socket schließen

Beispiel: Verbindungsorientierter 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;
```

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

    /* Ausfüllen der Adress-Struktur */
    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);

    /* Verbindung starten */
    if ( connect(sock, (struct sockaddr *)&server,
                                    sizeof(server) ) < 0 )
       { perror("Connect stream socket");
         exit(1);
       }

    /* Auf den Socket schreiben */
    if ( send(sock, DATA, sizeof DATA, 0) < 0)
       { perror("Write on stream socket");
         exit(1);
       }

    close(sock);
}
```

Der Client erzeugt mit der Funktion *socket()* einen Kommunikationsendpunkt (Socket) und den zugehörigen Deskriptor.
Mit *gethostbyname()* ermittelt der Client die Adresse des Rechners (der Rechnername wird als Parameter übergeben). Danach muss für den gewünschten Rechner eine Verbindung zum Server hergestellt werden. Zu diesem Zweck initialisiert der Client die Adress-Struktur. Die Verbindung wird mit *connect()* aufgebaut.
Nach dem Verbindungsaufbau werden mit der Funktion *send()* Daten auf den Socket geschrieben.
Mit der Funktion *close()* wird der erzeugte Socket wieder geschlossen.

Das Beispielprogramm ist nur für die Kommunikationsdomäne AF_INET dargestellt.
Es gilt auch für die Domäne AF_INET6, wenn Sie die entsprechenden Änderungen durchführen. Eine detaillierte Beschreibung dazu finden Sie in den Abschnitten „Socket-Adressierung“ auf Seite 13 und „Socket erzeugen“ auf Seite 17.
Beachten Sie auch, dass Sie die Socket-Funktion *getipnodebyname()* verwenden (anstelle von *gethostbyname()* bei AF_INET).

5.3 Verbindungsloser Server

Die meisten Services arbeiten verbindungsorientiert. Einige Services basieren jedoch auf der Verwendung von Datagramm-Sockets und arbeiten somit verbindungslos.

Im Beispielprogramm verwendet der Server die folgenden Funktionen der Socket- bzw. POSIX-Schnittstelle:

– *socket()*: Socket erzeugen
– *bind()*: einem Socket einen Namen zuordnen
– *recvfrom()*: Nachricht von einem Socket lesen
– *close()*: Socket schließen

Das Programm wird in zwei Varianten vorgestellt:

– In der ersten Variante (Beispiel 1) wird das Programm beendet, wenn eine Nachricht ankommt (*read()*).
– In der zweiten Variante (Beispiel 2) wartet das Programm in einer Endlosschleife auf weitere Nachrichten, nachdem eine Nachricht gelesen wurde.

Beispiel 1: Verbindungsloser Server ohne Programmschleife

```
#include <stdio.h>

#include <sys/socket.h>
#include <ioctl.h>
#include <signal.h>
#include <netinet/in.h>
#include <netdb.h>

#define TESTPORT 2222

/*
 * Dieses Programm erzeugt einen Datagramm-Socket, ordnet ihm einen
 * definierten Port zu und liest dann Daten vom Socket.
 */

main()
{
    int sock;
    int length;
    struct sockaddr_in server;
    char buf[1024];
```

```
/* Erzeugen des Sockets, von dem gelesen werden soll. */
    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock < 0 )
       { perror("Socket datagram");
         exit(1);
       }

    /*
     * Dem Server "server" unter Verwendung von Wildcards einen
     * Namen zuordnen
     */
    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);
       }

    /* Beginn des Lesens vom 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);
}
```

Beispiel 2: Verbindungsloser Server mit Programmschleife

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


/*
 * Dieses Programm erzeugt einen Datagramm-Socket, ordnet ihm einen
 * definierten Port zu und liest dann Daten vom Socket.
 */


main()
{
    int sock;
    int length;
    struct sockaddr_in server;
    char buf[1024];

    /* Erzeugen des Sockets, von dem gelesen werden soll. */
    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock < 0 )
       { perror("Socket datagram");
         exit(1);
       }

    /* Dem Server "server" unter Verwendung von Wildcards einen
     * Namen zuordnen */
    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);
       }

    /* Beginn des Lesens vom 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);
    }

    /* Da dieses Programm in einer Endlos-Schleife läuft, wird der
     * Socket "sock" niemals explizit geschlossen. Allerdings werden alle
     * Sockets automatisch geschlossen, wenn der Prozess abgebrochen wird
     * oder sein normales Ende erreicht.
     */
}
```

Im Programm werden folgende Arbeitsschritte durchgeführt:

– Der Server erzeugt mit der Funktion *socket()* einen Kommunikationsendpunkt (Socket) und den zugehörigen Deskriptor.

– Mit der Funktion *bind()* wird dem Server-Socket eine definierte Portnummer zugeordnet, sodass er über diese Portnummer vom Netz aus adressiert werden kann.

– Von einem Socket des Typs SOCK_DGRAM kann mit der Funktion *recvfrom()* gelesen werden. Als Ergebnis wird die Länge der gelesenen Nachricht geliefert.

– Wenn keine Nachricht vorhanden ist, wird der Prozess blockiert, bis eine Nachricht eintrifft.

Die Beispielprogramme sind nur für die Kommunikationsdomäne AF_INET dargestellt. Sie gelten auch für die Domäne AF_INET6, wenn Sie die entsprechenden Änderungen durchführen. Eine detaillierte Beschreibung dazu finden Sie in den Abschnitten „Socket-Adressierung“ auf Seite 13 und „Socket erzeugen“ auf Seite 17.

5.4 Verbindungsloser Client

In diesem Programmbeispiel verwendet der Client die folgenden Funktionen der Socket- bzw. POSIX-Schnittstelle:

- *socket()*: Socket erzeugen
- *gethostbyname()*: Rechner-Eintrag abfragen
- *sendto()*: Nachricht an einen Socket senden
- *close()*: Socket schließen

Beispiel: Verbindungsloser 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

/*
 * Dieses Programm sendet ein Datagramm an einen Empfänger, dessen Name als
 * Argument in der Kommandozeile übergeben wird. Das Format des Kommando ist:
 *                    progname hostname
 */

main(argc,argv)
    int argc;
    char *argv[];

{
    int sock;
    struct sockaddr_in to;
    struct hostent *hp, *gethostbyname();

    /* Erzeugen des Sockets, an den gesendet werden soll. */
    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock < 0 )
       { perror("Socket datagram");
         exit(1);
       }
```

```
/* Konstruieren des Namens des Sockets, an den gesendet werden soll,
     * ohne Verwendung von Wildcards. gethostbyname liefert eine Struktur,
     * die die Netzadresse des angegebenen Rechners enthält.
     * Die Portnummer wird aus der Konstante TESTPORT entnommen.
     */
    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);

    /* Nachricht senden. */
    if (sendto(sock, DATA, sizeof DATA, 0,
                        (struct sockaddr *)&to, sizeof to) < 0) {
         perror("Sending datagram message");
         exit(1);
    }
    close(sock);
}
```

Im Programm werden folgende Arbeitsschritte durchgeführt:

- Der Client erzeugt mit *socket()* einen Kommunikationsendpunkt (Socket) und den zugehörigen Deskriptor.
- Mit *gethostbyname()* fragt der Client die Adresse des Rechners ab; es wird der Rechnername als Parameter übergeben.
- Danach wird die Adress-Struktur initialisiert.
- Mit *sendto()* sendet der Client ein Datagramm. *sendto()* liefert die Anzahl der übertragenen Zeichen zurück.
- Mit *close()* schließt der Client den Socket.

Das Beispielprogramm ist nur für die Kommunikationsdomäne AF_INET dargestellt.
Es gilt auch für die Domäne AF_INET6, wenn Sie die entsprechenden Änderungen durchführen. Eine detaillierte Beschreibung dazu finden Sie in den Abschnitten „Socket-Adressierung“ auf Seite 13 und „Socket erzeugen“ auf Seite 17.

Beachten Sie auch, dass Sie die Socket-Funktion *getipnodebyname()* verwenden (anstelle von *gethostbyname()* bei AF_INET).

6 Benutzerfunktionen von SOCKETS(POSIX)

In diesem Kapitel sind die Funktionen der SOCKETS(POSIX)-Schnittstelle für BS2000/OSD beschrieben.

Zunächst wird das Format vorgestellt, in dem die einzelnen Funktionen beschrieben sind. Die darauf folgende Übersicht fasst jeweils mehrere Funktionen unter aufgabenorientierten Gesichtspunkten zusammen. Im Anschluss daran sind alle Funktionen der SOCKETS-Schnittstelle in alphabetischer Reihenfolge beschrieben.

Die Funktionen für die Behandlung von Dateideskriptoren werden von der POSIX-Schnittstelle zur Verfügung gestellt. Dies betrifft die Funktionen *read()*, *readv()*, *write()*, *writev()*, *ioctl()*, *fcntl()* und *close()* sowie die Funktionen *poll()* und *select()*. Diese Funktionen sind im Handbuch „ C-Bibliotheksfunktionen (BS2000/OSD) für POSIX-Anwendungen“ beschrieben. Besonderheiten für die Anwendung dieser Funktionen mit Sockets sind am Ende des vorliegenden Kapitels ab Seite 136 beschrieben.

6.1 Beschreibungsformat

Die Benutzerfunktionen von SOCKETS(POSIX) sind in einem einheitlichen Format beschrieben. Die Beschreibung der Funktionen ist wie folgt aufgebaut:

Funktionsname - Kurzbeschreibung der Funktionalität

```
#include < ... >
#include < ... >
     ...
Syntax der Funktion
```

Beschreibung

Ausführliche Beschreibung der Funktionalität und Erklärung der Parameter.

Returnwert

Auflistung und Beschreibung möglicher Returnwerte der Funktion.
Nicht jede Funktion liefert einen Returnwert. In solchen Fällen und bei der Beschreibung von externen Variablen fehlt der Abschnitt „Returnwert“.

Fehler

Aufzählung und Beschreibung der Fehlercodes, die bei einem fehlerhaften Aufruf oder Ablauf der Funktion in der externen Variablen *errno* abgelegt werden. Dieser Abschnitt kann fehlen.

Hinweis

Begriffserklärungen oder Informationen über das Zusammenwirken mit anderen Funktionen oder Tipps für die Anwendung. Dieser Abschnitt kann fehlen.

Siehe auch

Querverweise auf Funktionsbeschreibungen, Dateien, andere Handbuchteile oder andere Handbücher. Mit „[1]“ wird dabei auf das Handbuch „C-Bibliotheksfunktionen (BS2000/OSD) für POSIX-Anwendungen“ verwiesen. Dieser Abschnitt kann fehlen.

6.2 Übersicht über die Funktionen

Die folgende Übersicht über die Funktionen der SOCKETS-Schnittstelle fasst jeweils mehrere Funktionen unter aufgabenorientierten Gesichtspunkten zusammen. Mit „[1]“ wird in den nachfolgenden Übersichten auf das Handbuch „C-Bibliotheksfunktionen (BS2000/OSD) für POSIX-Anwendungen“ verwiesen.

Die Spalten INET und INET6 zeigen an, in welchen Adressfamilien (AF_INET bzw. AF_INET6) die betreffenden Funktionen unterstützt sind.

Verbindungen über Sockets aufbauen und beenden

Funktion	Beschreibung	siehe	INET	INET6
socket()	Socket erzeugen	Seite 131	x	x
bind()	einem Socket einen Namen zuordnen	Seite 84	x	x
connect()	Kommunikation über einen Socket initiieren (z.B. durch einen Client)	Seite 87	x	x
listen()	Socket auf anstehende Verbindungen überprüfen (z.B. durch einen Server)	Seite 120	x	x
accept()	Verbindung über einen Socket annehmen (z.B. durch einen Server)	Seite 81	x	x
close()	Socket schließen	[1] und Seite 137	x	x
shutdown()	Voll-Duplex-Verbindung beenden	Seite 130	x	x
socketpair()	ein Paar von verbundenen Sockets erzeugen	Seite 134	x	

Daten zwischen zwei Sockets übertragen

Funktion	Beschreibung	siehe	INET	INET6
read(), readv()	Nachricht von einem Socket mit aufgebauter Verbindung empfangen	[1] und Seite 149	x	x
recv()	Nachricht von einem Socket mit aufgebauter Verbindung empfangen	Seite 122	x	x
recvfrom()	Nachricht von einem Socket empfangen	Seite 122	x	x
recvmsg()	Nachricht von einem Socket empfangen	Seite 122	x	x
send()	Nachricht von Socket zu Socket über eine Verbindung senden	Seite 126	x	x
sendto()	Nachricht von Socket zu Socket senden	Seite 126	x	x
sendmsg()	Nachricht von Socket zu Socket senden	Seite 126	x	x
write(), writev()	Nachricht von Socket zu Socket über eine Verbindung senden	[1] und Seite 154	x	x
poll()	Ein-/Ausgabe multiplexen	Seite 146	x	x
select()	Ein-/Ausgabe multiplexen	Seite 151	x	x

Informationen über Sockets erhalten

Funktion	Beschreibung	siehe	INET	INET6
getsockopt()	Socket-Optionen abfragen	Seite 112	x	x
setsockopt()	Socket-Optionen setzen	Seite 112	x	x
getpeername()	Namen des Kommunikationspartners abfragen	Seite 106	x	x
getsockname()	Namen des Sockets abfragen	Seite 111	x	x

Konfigurationswerte überprüfen

Funktion	Beschreibung	siehe	INET	INET6
gai_strerror()	Beschreibung des Error-Codes von *getaddrinfo()* ermitteln	Seite 92	x	x
getaddrinfo()	Informationen über Rechnernamen, Rechneradressen und Services abfragen	Seite 93	x	x
gethostbyaddr()	Namen von erreichbaren Rechnern abfragen	Seite 97	x	
gethostbyname()	Adressen von erreichbaren Rechnern abfragen	Seite 97	x	
gethostname()	Namen des aktuellen Rechners abfragen	Seite 99	x	x
getipnodebyaddr()	den zu einer IPv4- oder IPv6-Adresse gehörenden Rechnernamen erfragen	Seite 100	x	x
getipnodebyname()	die zu einem Rechnernamen gehörende IPv4- oder IPv6-Adresse erfragen	Seite 100	x	x
getnameinfo()	die zu IP-Adresse und Portnummer gehörenden Rechner- und Servicenamen erfragen	Seite 102	x	x
getnetbyaddr()	Namen eines Netzes abfragen	Seite 104	x	x
getnetbyname()	Netzadresse abfragen	Seite 104	x	x
getprotobyname()	Nummer eines Protokolls abfragen	Seite 107	x	x
getprotobynumber()	Namen eines Protokolls abfragen	Seite 107	x	x
getservbyname()	Portnummer eines Service abfragen	Seite 109	x	x
getservbyport()	Namen eines Service abfragen	Seite 109	x	x
sethostent()	Rechner-Datenbasis öffnen	Seite 97	x	x
gethostent()	Eintrag aus der Rechner-Datenbasis lesen	Seite 97	x	x
endhostent()	Rechner-Datenbasis schließen	Seite 97	x	x
setnetent()	Netz-Datenbasis öffnen	Seite 104	x	x
getnetent()	Eintrag aus der Netz-Datenbasis lesen	Seite 104	x	x
endnetent()	Netz-Datenbasis schließen	Seite 104	x	x
setprotoent()	Protokoll-Datenbasis öffnen	Seite 107	x	x
getprotoent()	Eintrag aus der Protokoll-Datenbasis lesen	Seite 107	x	x
endprotoent()	Protokoll-Datenbasis schließen	Seite 107	x	x

Funktion	Beschreibung	siehe	INET	INET6
setservent()	Services-Datenbasis öffnen	Seite 109	x	x
getservent()	Eintrag aus der Services-Datenbasis lesen	Seite 109	x	xx
endservent()	Services-Datenbasis schließen	Seite 109	x	x

Internet-Adresse manipulieren

Funktion	Beschreibung	siehe	INET	INET6
inet_addr()	Zeichenkette von Punktschreibweise in ganzzahligen Wert konvertieren (Internetadresse)	Seite 115	x	
inet_network()	Zeichenkette von Punktschreibweise in ganzzahligen Wert konvertieren (Subnetz-Anteil)	Seite 115	x	
inet_makeaddr()	Internet-Adresse erstellen aus Subnetz-Anteil und subnetz-lokalem Adressteil	Seite 115	x	
inet_lnaof()	aus der Internet-Rechneradresse die lokale Netzadresse in der Byte-Reihenfolge des Rechners extrahieren	Seite 115	x	
inet_netof()	aus der Internet-Rechneradresse die Netznummer in der Byte-Reihenfolge des Rechners extrahieren	Seite 115	x	
inet_ntoa()	Internet-Rechneradresse in eine Zeichenkette gemäß der üblichen Internet-Punktschreibweise konvertieren	Seite 115	x	
inet_pton()	eine IP-Adresse aus der Punkt- bzw. Doppelpunkt-Notation in die binäre Adresse umwandeln	Seite 118	x	x
inet_ntop()	eine binäre IP-Adresse in die Punkt- oder Doppelpunkt-Notation umwandeln	Seite 118	x	x

Hilfsfunktionen

Funktion	Beschreibung	siehe	INET	INET6
freeaddrinfo()	Speicher einer *addrinfo*-Struktur freigeben	Seite 90	x	x
freehostent()	Speicher einer *hostent*-Struktur freigeben	Seite 91	x	x
htonl()	32-bit-Felder von Rechner- in Netz-Byte-Reihenfolge umsetzen	Seite 86	x	
htons()	16-bit-Felder von Rechner- in Netz-Byte-Reihenfolge umsetzen	Seite 86	x	x
ntohl()	32-bit-Felder von Netz- in Rechner-Byte-Reihenfolge umsetzen	Seite 86	x	
ntohs()	16-bit-Felder von Netz- in Rechner-Byte-Reihenfolge umsetzen	Seite 86	x	x

Steuerfunktionen

Funktion	Beschreibung	siehe	INET	INET6
fcntl()	Sockets steuern	[1] und Seite 138	x	x
ioctl()	Sockets steuern	[1] und Seite 140	x	x

Testmakros für AF_INET6

Folgende Testmakros für die Adressfamilie AF_INET6 sind in <netinet/in.h> definiert. Der Parameter *p* ist eine *in6_addr*-Struktur.

Funktion	Beschreibung
IN6_IS_ADDR_UNSPECIFIED (p)	IPv6-Adresse = 0 ?
IN6_IS_ADDR_LOOPBACK (p)	IPv6-Adresse = Loopback ?
IN6_IS_ADDR_LINKLOCAL (p)	IPv6-Adresse = Linklocal ?
IN6_IS_ADDR_SITELOCAL (p)	IPv6-Adresse = Sitelocal ?
IN6_IS_ADDR_V4MAPPED (p)	IPv6-Adresse = IPv4-mapped ?
IN6_IS_ADDR_V4COMPAT (p)	IPv6-Adresse = IPv4-kompatibel ?
IN6_ARE_ADDR_EQUAL	Adresse1 = Adresse2 ?

6.3 Beschreibung der Funktionen

In diesem Abschnitt sind alle Benutzerfunktionen der SOCKETS-Schnittstelle in alphabetischer Reihenfolge beschrieben.

accept() - Eine Verbindung über einen Socket annehmen

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

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

Beschreibung

Mit der Funktion *accept()* nimmt der Server-Prozess über einen Socket eine Verbindung an, die ein Client mit der Funktion *connect()* angefordert hat. *accept()* wird nur bei dem verbindungsorientierten Socket-Typ SOCK_STREAM verwendet.

Der Parameter *s* bezeichnet den Socket, der nach dem Aufruf von *listen()* auf eine Verbindungsanforderung wartet.
addr zeigt bei Rückkehr von *accept()* auf die Adresse der Partneranwendung, wie sie auf der Kommunikationsebene bekannt ist. Das exakte Format von **addr* (d.h. der Adresse) wird durch die Domäne bestimmt, in der die Kommunikation stattfindet.
Der Abschnitt „Socket-Adressierung“ auf Seite 13 beschreibt, wie Sie dem Socket eine Adresse zuordnen.
addrlen zeigt auf ein *size_t*-Objekt, das zum Zeitpunkt des Aufrufs von *accept()* die Größe des von *addr* referenzierten Speicherbereichs enthält. Bei Rückkehr der Funktion *accept()* enthält dieses *size_t*-Objekt (d.h. **addrlen*) die Länge (in Bytes) der zurückgelieferten Adresse.

Wenn in der durch die Funktion *listen()* eingerichteten Warteschlange mindestens eine Verbindungsanforderung vorliegt, verfährt *accept()* wie folgt:

1. *accept()* wählt aus der Warteschlange der anstehenden Verbindungsanforderungen die erste Verbindung aus.
2. *accept()* erzeugt einen neuen Socket, der die gleichen Eigenschaften hat wie der Socket *s*.
3. *accept()* liefert als Ergebnis den Descriptor für den neuen Socket. Wenn der Socket *s* nicht-blockierend ist, ist auch der neue Socket nicht-blockierend (siehe Funktion *fcntl()* auf Seite 138).

Stehen in der Warteschlange keine Verbindungsanforderungen an, sind zwei Fälle zu unterscheiden:

– Wenn der Socket **nicht** als nicht-blockierend markiert ist, blockiert *accept()* den aufrufenden Prozess solange, bis eine Verbindung möglich ist.

– Wenn der Socket als nicht-blockierend markiert ist, liefert *accept()* die Fehlermeldung EWOULDBLOCK.

Um sicher zu gehen, dass der *accept()*-Aufruf nicht blockieren wird, kann der Benutzer vor dem Aufruf von *accept()* zunächst mit *select()* oder *poll()* die Lesebereitschaft des betreffenden Sockets prüfen.

Wenn *accept()* für den Socket *s* eine Verbindung akzeptiert hat, können über den mit *accept()* neu erzeugten Socket Daten ausgetauscht werden mit dem Socket, der die Verbindung angefordert hat. Weitere Verbindungen können über den neu erzeugten Socket nicht hergestellt werden. Der ursprüngliche Socket *s* bleibt geöffnet, um weitere Verbindungen anzunehmen.

Returnwert

≥ 0:
bei Erfolg. Der Wert ist der Deskriptor für den akzeptierten Socket.

-1:
bei Fehler. *errno* wird gesetzt, um den Fehler anzuzeigen.

Fehler

EBADF
s ist kein gültiger Deskriptor.

EFAULT
addr zeigt nicht auf den beschreibbaren Bereich des Benutzer-Adressbereichs.

EINTR
Die Funktion *accept()* wurde durch ein Signal unterbrochen, das vor Eingang einer Verbindungsanforderung empfangen wurde.

EINVAL
Der Socket akzeptiert keine Verbindungsanforderungen.

EMFILE
Im aufrufenden Prozess sind zurzeit OPEN_MAX Dateideskriptoren geöffnet.

ENETDOWN
Die Verbindung zum Netzwerk ist nicht mehr aktiv.

ENOBUFS
Es ist kein Pufferplatz verfügbar.

ENOTSOCK
Der Deskriptor verweist nicht auf einen Socket.

EOPNOTSUPP
Der referenzierte Socket ist nicht vom Typ SOCK_STREAM.

EPROTO
Es ist ein Protokollfehler aufgetreten.

EWOULDBLOCK
Der Socket ist als nicht-blockierend gekennzeichnet, und es stehen keine freien Verbindungen zur Verfügung.

Siehe auch

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

bind() - Einem Socket einen Namen zuordnen

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

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

Beschreibung

Die Funktion *bind()* ordnet einem mit der Funktion *socket()* erzeugten, zunächst namenlosen Socket einen Namen zu. Nachdem ein Socket mit der Funktion *socket()* erzeugt worden ist, existiert dieser Socket zwar innerhalb eines Namensbereichs (Adressfamilie), hat aber noch keinen Namen.

Der Parameter *s* bezeichnet den Socket, dem mit *bind()* ein Name zugeordnet werden soll. *name* zeigt auf den Namen (Adresse), der dem Socket zugeordnet wird. *namelen* spezifiziert die Länge der Datenstruktur, die den Namen beschreibt.

Returnwert

0:
bei Erfolg.

-1:
bei Fehler. *errno* wird gesetzt, um den Fehler anzuzeigen.

Fehler

EACCES
Der angegebene Name ist geschützt, und der aufrufende Benutzer hat nicht die Berechtigung, darauf zuzugreifen.

EADDRINUSE
Der angegebene Name wird bereits benutzt.

EADDRNOTAVAIL
Der angegebene Name kann vom lokalen System nicht an den Socket gebunden werden (siehe auch Abschnitt „Abhängigkeiten vom BS2000/OSD-Transportsystem BCAM“ auf Seite 316).

EAFNOSUPPORT
Die angegebene Adressfamilie stimmt nicht mit der Adressfamilie des Sockets überein.

EBADF
s ist kein gültiger Deskriptor.

EFAULT
name zeigt nicht auf den beschreibbaren Bereich des Benutzer-Adressbereichs.

EINVAL
Dem Socket ist bereits ein Name zugeordnet oder *namelen* hat nicht die Größe einer gültigen Adresse für die angegebene Adressfamilie.

ENETDOWN
Die Verbindung zum Netzwerk ist nicht mehr aktiv.

ENOBUFS
Die Ressourcen reichen für die Ausführung von *bind()* nicht aus.

ENOTSOCK
Der Deskriptor referenziert eine Datei, keinen Socket.

Wenn AF_UNIX die Adressfamilie des Sockets ist, kann die Ausführung von bind() auch aus folgenden Gründen zu einem Fehler führen:

EACCES
Der angegebene Name ist geschützt, oder für den angegebenen Namen hat der aufrufende Benutzer keine Schreibberechtigung.

EDESTADDRREQ
Der Parameter *name* ist der Null-Zeiger.

ENAMETOOLONG
Eine Komponente eines Pfadnamens überschreitet NAME_MAX Zeichen, oder der gesamte Pfadname ist länger als PATH_MAX Zeichen.

ENOENT
Eine Komponente des Pfadnamens verweist auf eine nicht vorhandene Datei, oder der Pfadname ist leer.

ENOTDIR
Eine Komponente im Pfadnamen ist kein Verzeichnis.

Siehe auch

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

Byteorder-Makros - Byte-Reihenfolgen umsetzen

```
#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);
```

Beschreibung

Die Makros *htonl()*, *htons()*, *ntohl()* und *ntohs()* setzen Shorts und Integers von Rechner-Byte-Reihenfolge in Netz-Byte-Reihenfolge um bzw. umgekehrt.
Die Definitionen der Datentypen *in_addr_t* und *in_port_t* in <arpa/inet.h> entsprechen den Definitionen in <netinet/in.h>.

Diese Makros werden meistens im Zusammenhang mit IPv4-Adressen und Portnummern verwendet, wie sie z.B. die Funktionen *gethostbyname()* und *getservent()* liefern (siehe Seiten 97 und 109). Die Makros werden nur an Systemen benötigt, an denen Rechner- und Netz-Byte-Reihenfolge unterschiedlich ist. Die Makros werden in der Include-Datei <arpa/inet.h> als Null-Makros (Makros ohne Funktion) bereitgestellt:

- *htonl()* setzt 32-bit-Felder von Rechner- in Netz-Byte-Reihenfolge um.
- *htons()* setzt 16-bit-Felder von Rechner- in Netz-Byte-Reihenfolge um.
- *ntohl()* setzt 32-bit-Felder von Netz- in Rechner-Byte-Reihenfolge um.
- *ntohs()* setzt 16-bit-Felder von Netz- in Rechner-Byte-Reihenfolge um.

Returnwert

htonl() und *htons()* liefern den in die Netz-Byte-Reihenfolge konvertierten Wert des Eingabeparameters zurück.

ntohl() und *ntohs()* liefern den in die Rechner-Byte-Reihenfolge konvertierten Wert des Eingabeparameters zurück.

Siehe auch

gethostbyaddr(), gethostbyname(), gethostent(), getservent()

connect() - Verbindung über einen Socket initiieren

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

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

Beschreibung

Mit *connect()* initiiert ein Prozess über einen Socket die Kommunikation mit einem anderen Prozess.

Der Parameter *s* bezeichnet den Socket, über den der Prozess die Kommunikation mit einem anderen Prozess initiiert.
name ist ein Zeiger auf die Adresse des Kommunikationspartners. **name* ist eine Adresse im Adressbereich des Sockets, zu dem die Verbindung initiiert werden soll. Jeder Adressbereich interpretiert den Parameter *name* auf seine eigene Art.
namelen enthält die Länge (in Bytes) der Adresse des Kommunikationspartners.

Je nachdem, ob es sich um einen Socket vom Typ SOCK_STREAM oder SOCK_DGRAM handelt, verfährt *connect()* unterschiedlich:

– Bei einem Socket vom Typ SOCK_STREAM (Stream-Socket) sendet *connect()* eine Verbindungsanforderung an einen Partner und versucht so, eine Verbindung zu diesem herzustellen. Der Partner wird durch den Parameter *name* spezifiziert. Mit *connect()* initiiert z.B. ein Client-Prozess über einen Stream-Socket eine Verbindung zu einem Server.
  Generell können Stream-Sockets nur einmal eine Verbindung mit *connect()* herstellen.

– Bei einem Socket vom Typ SOCK_DGRAM (Datagramm-Socket) legt ein Prozess mit *connect()* den Namen des Kommunikationspartners fest, mit dem der Datenaustausch erfolgen soll. An diesen Kommunikationspartner sendet der Prozess dann die Datagramme. Außerdem ist dieser Kommunikationspartner der einzige Socket, von dem der Prozess Datagramme empfangen kann.
  Bei Datagramm-Sockets kann *connect()* mehrmals verwendet werden, um die Kommunikationspartner zu wechseln. Durch Angabe eines Null-Zeigers beim Parameter *name* kann die Zuordnung zu einem bestimmten Partner beendet werden.

Returnwert

0:
bei Erfolg.

-1:
bei Fehler. *errno* wird gesetzt, um den Fehler anzuzeigen.

Fehler

EADDRINUSE
Die angegebene Adresse wird bereits benutzt.

EADDRNOTAVAIL
Die angegebene Adresse ist keine gültige Adresse.

EAFNOSUPPORT
Adressen in der angegebenen Adressfamilie können mit diesem Socket nicht verwendet werden.

EALREADY
Es handelt sich um einen nicht-blockierenden Socket, und eine vorher eingetroffene Verbindungsanforderung wurde noch nicht abgeschlossen.

EBADF
s ist kein gültiger Deskriptor.

ECONNREFUSED
Der Verbindungsversuch wurde zurückgewiesen. Das aufrufende Programm muss den Socket-Deskriptor mit *close()* schließen und mit einem neuen Aufruf von *socket()* einen neuen Deskriptor anfordern. Danach kann es mit *connect()* den Verbindungsversuch wiederholen.

EFAULT
Der Parameter *name* zeigt auf einen Bereich außerhalb des Prozess-Adressbereichs.

EINTR
Der Verbindungsaufbauversuch wurde durch ein Signal unterbrochen.

EINVAL
Der Parameter *namelen* hat nicht die Größe einer gültigen Adresse für die angegebene Adressfamilie.

EISCONN
Der Socket hat bereits eine Verbindung.

ENETUNREACH
Das Netz ist von diesem Rechner aus nicht erreichbar.

ENETDOWN
Die Verbindung zum Netzwerk ist nicht mehr aktiv.

ENOBUFS
Die Ressourcen reichen für die Ausführung von *connect()* nicht aus.

ENOTSOCK
Der Deskriptor referenziert eine Datei, keinen Socket.

ETIMEDOUT
Die Verbindung konnte nicht innerhalb einer bestimmten Zeitspanne aufgebaut werden.

Wenn AF_UNIX die Adressfamilie des Sockets ist, kann die Ausführung von connect() auch aus folgenden Gründen zu einem Fehler führen:

EACCES
Für eine Komponente des Pfadnamens wird die Zugriffsberechtigung verweigert, oder die Berechtigung zum Schreiben auf den angegebenen Socket wird verweigert.

EDESTADDRREQ
Der Parameter *name* ist der Null-Zeiger.

ENAMETOOLONG
Eine Komponente eines Pfadnamens überschreitet NAME_MAX Zeichen, oder der gesamte Pfadname ist länger als PATH_MAX Zeichen.

ENOENT
Eine Komponente des Pfadnamens verweist auf eine nicht vorhandene Datei, oder der Pfadname ist leer.

ENOTDIR
Eine Komponente im Pfadnamen ist kein Verzeichnis.

Siehe auch

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

freeaddrinfo() - Speicher für addrinfo-Struktur freigeben

```
#include <netdb.h>

void freeaddrinfo(struct addrinfo *ai)
```

Beschreibung

Die Funktion *freeaddrinfo()* gibt den Speicherplatz für eine verkettete Liste von *struct addrinfo*-Objekten frei, der zuvor mit der Funktion *getaddrinfo()* angefordert wurde.

Der Parameter *ai* ist ein Zeiger und zeigt auf das erste *addrinfo*-Objekt in einer Liste von mehreren miteinander verketteten *addrinfo*-Objekten.

Die Struktur *addrinfo* ist wie folgt deklariert:

```
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 oder IPPROTO_xxx für IP */
   size_t ai_addrlen;         /* Länge von ai_addr */
   char *ai_canonname;        /* kanonischer Name */
   struct sockaddr *ai_addr;  /* Socket-Adress-Struktur */
   struct addrinfo *ai_next;  /* nächste Struktur der Liste */
};
```

freehostent() - Speicher für hostent-Struktur freigeben

```
#include <netdb.h>
void freehostent(struct hostent *ptr)
```

Beschreibung

Die Funktion *freehostent()* gibt den Speicherplatz für ein Objekt vom Typ *struct hostent* frei, das zuvor mit den Funktionen *getipnodebyname()* oder *getipnodebyaddr()* angefordert wurde.

Der Parameter *ptr* zeigt auf ein Objekt vom Typ *struct hostent.*

Die Deklaration der Struktur *hostent* finden Sie im Abschnitt „Rechnernamen in Netzadressen umwandeln und umgekehrt“ auf Seite 44.

gai_strerror() - Textausgabe für den Error-Code von getaddrinfo()

```
#include <netdb.h>

char *gai_strerror(int ecode)
```

Beschreibung

Die Funktion *gai_strerror()* gibt zu einem in <netdb.h> definierten Error-Code einen erklärenden Text-String aus. Der Parameter *ecode* spezifiziert einen in <netdb.h> definierten Error-Code.

Returnwert

gai_strerror() liefert einen Zeiger auf den String zurück, der den erklärenden Text enthält. Wenn der Wert von *ecode* mit keinem der in <netdb.h> für *getaddrinfo()* definierten Error-Codes übereinstimmt, ist der Returnwert ein Zeiger auf einen String, der einen Hinweis auf einen unbekannten Fehler enthält.

getaddrinfo() - Informationen über Rechnernamen, Rechneradressen und Services protokollunabhängig abfragen

```
#include <netdb.h>

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

Beschreibung

Mit der Funktion *getaddrinfo()* fragen Sie protokollunabhängig Rechnerinformationen für die Adressfamilien AF_INET und AF_INET6 ab. Die Werte werden entweder über den Domain Name Service (DNS) oder über systemspezifische Tabellen ermittelt.

Parameter nodename und servname

Beim Aufruf von *getaddrinfo()* muss mindestens einer der Parameter *nodename* oder *servname* vom Null-Zeiger verschieden sein. *nodename* und *servname* sind entweder ein Null-Zeiger oder ein mit dem Null-Byte abgeschlossener String. Der Parameter *nodename* kann sowohl ein Name sein, als auch eine IPv4-Adresse in dezimaler Punkt-Notation oder eine IPv6-Adresse in sedezimaler Doppelpunkt-Notation. Der Parameter *servname* kann entweder ein Service-Name oder eine dezimale Portnummer sein.

Parameter hints

Mit dem Parameter *hints* wird optional eine *addrinfo*-Struktur übergeben. Andernfalls muss der Parameter *hints* der Null-Zeiger sein.

Die Struktur *addrinfo* ist wie folgt deklariert:

```
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 oder IPPROTO_xxx für IP */
    size_t ai_addrlen;         /* Länge von ai_addr */
    char*ai_canonname;         /* kanonischer Name */
    struct sockaddr *ai_addr;  /* Socket-Adress-Struktur */
    struct addrinfo *ai_next;  /* nächste Struktur der Liste */
};
```

In dem mit *hints* übergebenen Objekt vom Typ *struct addrinfo* müssen alle Elemente außer *ai_flags*, *ai_family* und *ai_socktype* den Wert 0 haben bzw. der Null-Zeiger sein.

Mit den Werten legen Sie für die *addrinfo*-Komponenten *ai_flags, ai_family* und *ai_socktype* eine Auswahl fest:

- *ai_family* = PF_UNSPEC: Jede Protokollfamilie wird gewünscht.
- *ai_socktype* = 0: Jeder Socket-Typ wird akzeptiert.
- *ai_flags* = AI_PASSIVE: Die zurückgelieferte Sockets-Adress-Struktur wird für einen *bind()*-Aufruf verwendet. Wenn *nodename* = NULL ist (siehe oben), wird das IP-Adress-element bei einer IPv4-Adresse mit INADDR_ANY und bei einer IPv6-Adresse mit IN6ADDR_ANY gesetzt.
- Wenn das AI_PASSIVE-Bit nicht gesetzt ist, wird die zurückgelieferte Socket-Adress-Struktur wie folgt verwendet:
  - für einen *connect()*-Aufruf bei *ai_socktype* = SOCK_STREAM
  - für einen *connect()-, sendto()-, sendmsg()*-Aufruf bei *ai_socktype* = SOCK_DGRAM

  Wenn in diesen Fällen *nodename* der Null-Zeiger ist, wird die IP-Adresse von *sockaddr* mit dem Wert der loopback-Adresse versorgt.
- Wenn in den *ai_flags* der *hints*-Struktur das Bit AI_CANONNAME gesetzt ist, enthält – bei erfolgreicher Ausführung von *getaddrinfo()* – die erste zurückgegebene *addrinfo*-Struktur im Element *ai_canonname* den Socket-Hostnamen des ausgewählten Rechners. Dieser Socket-Hostname ist mit dem Null-Byte abgeschlossen.
- Wenn in den *ai_flags* der *hints*-Struktur das Bit AI_NUMERICHOST gesetzt ist, muss ein vom Null-Zeiger verschiedener *nodename* entweder ein IPv4-Adress-String in dezimaler Punkt-Notation oder ein IPv6-Adress-String in sedezimaler Doppelpunkt-Notation sein. Andernfalls wird der Returnwert EAI_NONAME geliefert. Das Flag verhindert einen Aufruf zur Namensauflösung durch einen DNS-Service oder interne Rechnertabellen.

hints = NULL bewirkt das Gleiche wie eine mit 0 initialisierte *addrinfo*-Struktur mit *ai_family* = PF_UNSPEC.

Parameter res

Bei erfolgreicher Ausführung von *getaddrinfo()* wird in *res* ein Zeiger auf eine oder mehrere miteinander verkettete *addrinfo*-Strukturen übergeben. Das Element *ai_next* = NULL kennzeichnet das letzte Kettenelement. Jede der zurückgelieferten *addrinfo*-Strukturen enthält in den Elementen *ai_family* und *ai_socktype* einen zum *socket()*-Aufruf korrespondierenden Wert. *ai_addr* zeigt immer auf eine Socket-Adress-Struktur, deren Länge in *ai_addrlen* spezifiziert ist.

Returnwert

0:
bei Erfolg.

>0:
Fehler, Returnwert ist ein in <netdb.h> definierter Fehlercode EAI_xxx.

-1:
Fehler, *errno* wird gesetzt, um den Fehler anzuzeigen

Fehler

EAFNOSUPPORT
Die Funktion wird an diesem System nicht unterstützt. Vergleichen Sie dazu auch den Abschnitt „Abhängigkeiten vom BS2000/OSD-Transportsystem BCAM“ auf Seite 316.

In <netdb.h> definierte Fehlercodes:

EAI_ADDRFAMILY
Die Adressfamilie wird für den angegebenen Rechner nicht unterstützt.

EAI_AGAIN
Temporärer Fehler beim Zugriff auf die Rechnernamen-Information (z.B. DNS-Fehler). Der Aufruf der Funktion sollte wiederholt werden.

EAI_BADFLAGS
Ungültiger Wert für den Operanden *ai_flags*.

EAI_FAIL
Fehler beim Zugriff auf die Rechnernameninformation.

EAI_FAMILY
Die Protokoll-Familie wird nicht unterstützt.

EAI_MEMORY
Fehler bei Speicheranforderung.

EAI_NODATA
Keine zum Rechnernamen korrespondierende Adresse gefunden.

EAI_NONAME
Rechner- oder Service-Name werden nicht unterstützt, oder sind unbekannt.

EAI_SERVICE
Service wird für den Socket-Typ nicht unterstützt.

EAI_SOCKTYPE
Der Socket-Typ wird nicht unterstützt.

EAI_SYSTEM
System-Fehler, wird in *errno* näher spezifiziert.

Hinweis

Der Speicher für die von *getaddrinfo()* gelieferten *addrinfo*-Strukturen wird dynamisch angefordert und muss mit der Funktion *freeaddrinfo()* wieder freigegeben werden.

gethostent(), gethostbyname(), gethostbyaddr(), sethostent(), endhostent() - Informationen über Rechnernamen und -adressen abfragen

```
#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);
```

Beschreibung

Die Funktionen *gethostbyname()* und *gethostbyaddr()* liefern aktuelle Informationen über die im Netz erreichbaren Rechner durch Aufruf einer BCAM-Informationsschnittstelle. Dabei wird das DNS-Konzept unterstützt, wenn der DNS-Resolver aus dem Produkt *inter*Net Services (früher TCP-IP-SV) in POSIX installiert ist oder das Subsystem SOCKETS (BS2000) gestartet ist.
Dagegen greift die Funktion *gethostent()* auf die UFS-Datei */etc/inet/hosts* zu, die im Standardfall nur einen Eintrag für den lokalen Rechner enthält.

Die Funktionen *gethostbyname()*, *gethostbyaddr()* und *gethostent()* liefern einen Zeiger auf ein Objekt der nachfolgend beschriebenen Struktur *hostent*.

Die Struktur *hostent* entspricht den Feldern einer Zeile der Rechner-Datenbasis und ist wie folgt deklariert:

```
struct hostent {
   char *h_name;           /* offizieller Rechnername */
   char **h_aliases;       /* Alias-Liste */
   int h_addrtype;         /* Adresstyp */
   int h_length;           /* Länge der Adresse (in Bytes) */
   char **h_addr_list;     /* Liste von Adressen für den Rechner, */
                           /* terminiert durch den Null-Zeiger */
};
```

Beschreibung der *hostent*-Komponenten:

h_name
: Name des Rechners

h_aliases
: Eine durch null abgeschlossene Liste mit alternativen Namen für den Rechner. Alias-Namen werden derzeit nicht unterstützt.

h_addrtype
: Typ der Adresse, die geliefert wird (immer AF_INET)

length
: Länge der Adresse (in Bytes)

**h_addr_list
: Ein Zeiger auf eine Liste von Netzadressen für den Rechner. Diese Adressen werden in Netz-Byte-Reihenfolge geliefert.

Im Falle von *gethostbyaddr()* ist *addr* ein Zeiger auf die Adresse in binärem Format mit der Länge *len* (kein Character-String).

gethostent() liest die nächste Zeile der Datei. Wenn nötig, öffnet *gethostent()* vorher die Datei.

sethostent() öffnet die Datei und setzt sie auf den Anfang zurück. Wenn das Flag *stayopen* ungleich null ist, wird die Datenbasis nach keinem Aufruf von *gethostent()* geschlossen (weder direkt noch indirekt durch einen der anderen *gethost...()*-Aufrufe).

Returnwert

Bei Fehler oder Dateiende wird der Null-Zeiger zurückgeliefert.

Hinweis

Alle Informationen befinden sich in einem statischen Bereich. Die Informationen müssen somit kopiert werden, wenn sie gesichert werden sollen.
Es wird nur das Adressformat der Internet-Adresse unterstützt.

gethostname() - Namen des aktuellen Rechners abfragen

```
#include <unistd.h>

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

Beschreibung

Die Funktion *gethostname()* liefert im Parameter *name* den Socket-Hostnamen für den aktuellen Rechner zurück. Beim Aufruf von *gethostname()* muss im Parameter *namelen* die Länge der String-Variablen *name* spezifiziert werden.

Genügt die durch *namelen* spezifizierte Länge der String-Variablen *name* für die Aufnahme des Hostnamens, so wird der Hostname durch das Null-Byte terminiert. Andernfalls werden die überzähligen Stellen des Socket-Hostnamens abgeschnitten, und es ist undefiniert, ob der so zurückgelieferte Hostname durch ein Null-Byte terminiert ist.

Returnwert

0:
bei Erfolg.

-1:
bei Fehler. *errno* wird gesetzt, um den Fehler anzuzeigen.

getipnodebyaddr(), getipnodebyname() - Informationen über Rechnernamen und -adressen abfragen

```
#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);
```

Beschreibung

getipnodebyaddr() und *getipnodebyname()* sind Erweiterungen der Funktionen *gethostbyaddr()* und *gethostbyname()* für die IPv6-Unterstützung.

Die Funktionen *getipnodebyaddr()* und *getipnodebyname()* liefern aktuelle Informationen über die im Netz bekannten Rechner. Dabei beschaffen sie sich die benötigten Informationen über den in SOCKETS(BS2000) V2.0 integrierten DNS-Resolver (Rechnername bzw. Rechneradresse) von einem DNS-Server. Falls dies nicht erfolgreich ist, wird die Information aus der BCAM-Prozessor-Tabelle ermittelt (siehe Handbuch „openNet Server V3.0 (BS2000/OSD)“).

Bei *getipnodebyaddr()* ist *addr* ein Zeiger auf die Rechneradresse. Die Rechneradresse muss in binärem Format mit der Länge *len* vorliegen.

Bei *getipnodebyname()* muss für *name* der Rechnername (Socket-Hostname) angegeben werden. Der Name kann in Form eines vollqualifizierten DNS-Namens, also mit Rechnernamen- und Domain-Anteil (z.B. rechnerx.fujitsu-siemens.com) oder als teilqualifizierter DNS-Name (z.B. rechnerx.) oder nur als Rechnername (z.B. rechnerx) angegeben werden. Alternativ kann für *name* auch eine IPv4-Adresse in Punkt-Notation oder eine IPv6-Adresse in sedezimaler Doppelpunkt-Notation angegeben werden.

Im Parameter *af* wird beim Aufruf die Adressfamilie spezifiziert: AF_INET oder AF_INET6. Bei *getipnodebyname* kann auch AF_UNSPEC angegeben werden, wenn eine IP-Adresse in Punkt- bzw. Doppelpunkt-Notation als *name* angegeben ist.

Mit dem Parameter *flags* kann die Ausgabe der gewünschten Adressfamilie gesteuert werden. Hat *flags* den Wert 0, dann wird eine der in *ai* spezifizierten Adressfamilie entsprechende Adresse zurückgeliefert.

AI_V4MAPPED
: Der Aufrufer akzeptiert eine IPv4-Mapped-Adresse, wenn keine IPv6-Adresse zur Verfügung steht.

AI_ALL
: Nur wenn auch AI_V4MAPPED gesetzt ist: Es werden, wenn vorhanden, IPv6-Adressen und IPv4-Mapped-Adressen zurückgeliefert. *af* muss den Wert AF_INET6 haben.

AI_ADDRCONFIG
: Es wird nur eine IPv6- oder IPv4-Adresse abhängig vom Wert von *af* zurückgegeben, wenn auch der Rechner, auf dem die Funktion aufgerufen wird, eine Interface-Adresse des gleichen Typs besitzt.

AI_DEFAULT
: ist gleich AI_ADDRCONFIG || AI_V4MAPPED. Wenn *af* = AF_INET6 gesetzt ist, dann wird für den angegebenen Rechnernamen eine IPv6-Adresse zurückgeliefert, wenn der Rechner, auf dem die Funktion aufgerufen wird, eine IPv6-Interface-Adresse hat. Wenn der Rechner, auf dem die Funktion aufgerufen wird, nur eine IPv4-Interface-Adresse besitzt, wird eine IPv4-Mapped-IPv6-Adresse zurückgegeben.

getipnodebyaddr() und *getipnodebyname()* liefern einen Zeiger auf ein Objekt vom Typ *struct hostent* zurück. Für dieses Objekt wird der Speicher dynamisch angefordert und muss mit der Funktion *freehostent()* vom Aufrufer wieder freigegeben werden.

Der Abschnitt „Rechnernamen in Netzadressen umwandeln und umgekehrt“ auf Seite 44 beschreibt die Struktur *hostent*.

Returnwert

Zeiger auf ein Objekt vom Typ *struct hostent*. Im Fehlerfall wird der Null-Zeiger zurückgeliefert und die Variable *err* mit einem der folgenden Werte versorgt:

HOST_NOT_FOUND
: Rechner unbekannt.

NO_ADDRESS
: Zu dem angegebenen Namen ist keine Rechneradresse verfügbar.

NO_RECOVERY
: Es ist ein nicht behebbarer Server-Fehler aufgetreten.

TRY_AGAIN
: Zugriff muss wiederholt werden.

-1:
: [als alternativer Wert für die Variable *err*]
Fehler; *errno* wird gesetzt, um den Fehler anzuzeigen.

Fehler

EAFNOSUPPORT
: Die Funktion wird an diesem System nicht unterstützt. Vergleichen Sie dazu auch den Abschnitt „Abhängigkeiten vom BS2000/OSD-Transportsystem BCAM“ auf Seite 316.

getnameinfo() - Namen des Kommunikationspartners abfragen

```
#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);
```

Beschreibung

Die Funktion *getnameinfo()* gibt den Namen, welcher der beim Aufruf angegebenen IP-Adresse und Portnummer zugeordnet ist, als Text-String zurück. Die Werte werden entweder über den in SOCKETS(BS2000) V2.0 integrierten DNS-Resolver von einem DNS-Server oder über systemspezifische Tabellen ermittelt.

Der Parameter *sa* ist ein Zeiger auf eine *sockaddr*-Struktur, die die IP-Adresse und Portnummer enthält. Das tatsächliche Format der *sockaddr*-Struktur hängt von der Adressfamilie ab und ist im Abschnitt „Socket-Adressierung“ auf Seite 13 beschrieben. Das exakte Format von **sa* wird durch die Domäne bestimmt, in der die Kommunikation stattfindet. *salen* gibt die Länge dieser Struktur an.

host bzw. *serv* sind Zeiger auf zwei Bereiche in denen nach erfolgreicher Ausführung der entsprechende Socket-Hostname bzw. Servicename stehen (mit Null-Byte abgeschlossen). Die Längen der Bereiche sind in *hostlen* bzw. *servlen* anzugeben. Diese müssen groß genug sein, um den Socket-Hostnamen bzw. den Servicenamen (einschließlich Null-Byte) aufzunehmen. Wird für *hostlen* oder *servlen* beim Aufruf der Wert Null angegeben, dann wird angezeigt, dass der Socket-Hostname bzw. der Servicename nicht zurückgeliefert werden soll.

Die maximalen Längen von Socket-Host- und Servicenamen sind in der Include-Datei <netdb.h> definiert:

```
#define NI_MAXHOST 1025

#define NI_MAXSERV 32
```

Der Parameter *flags* ändert die Art der Ausführung von *getnameinfo()*, bei dem standardmäßig der vollqualifizierte Domänen-Name des Rechners aus dem DNS ermittelt und zurückgeliefert wird. In Abhängigkeit des Wertes von *flags* sind folgende Fälle zu unterscheiden:

NI_NOFQDN
: Es soll nur der Rechnernamen-Anteil des vollen DNS-Namens (Socket-Hostname) zurückgeliefert werden.

NI_NUMERICHOST
: Es soll der numerische Hostname nach Adressumwandlung in abdruckbarer Form zurückgeliefert werden. Dasselbe ist der Fall, wenn der Rechnername weder im DNS noch durch lokale Information ermittelt werden kann und NI_NAMEREQD nicht gesetzt ist.

NI_NAMEREQD
: Es soll ein Fehler gemeldet werden, falls der Rechnername im DNS nicht ermittelt werden kann.

NI_NUMERICSERV
: Anstatt des Servicenamens soll die Portnummer in abdruckbarer Form zurückgeliefert werden.

Returnwert

0:
: bei Erfolg

<> 0:
: bei Fehler

Fehler

EAFNOSUPPORT
: Die Funktion wird an diesem System nicht unterstützt. Vergleichen Sie dazu auch den Abschnitt „Abhängigkeiten vom BS2000/OSD-Transportsystem BCAM“ auf Seite 316.

EINVAL
: Ungültige Adressfamilie im Parameter *sa* angegeben oder die Längen der Ausgabebereiche *host* bzw. *serv* sind zu klein.

getnetent(), getnetbyname(), getnetbyaddr(), setnetent(), endnetent() - Informationen über Netznamen abfragen

```
#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);
```

Beschreibung

Die Funktionen *getnetbyname()*, *getnetbyaddr()* und *getnetent()* liefern Informationen über Namen und Adressen der erreichbaren Netze. Die Informationen über lokale Netznamen sind in BCAM nicht vorhanden. Die Zuordnung erfolgt mittels einer UFS-Datei */etc/inet/networks*, wie in UNIX-Systemen üblich. Der Inhalt der Datei ist in EBCDIC codiert.

Die Funktionen *getnetbyname()*, *getnetbyaddr()* und *getnetent()* liefern einen Zeiger auf ein Objekt der nachfolgend beschriebenen Struktur *netent*.

Die Struktur *netent* entspricht den Feldern einer Zeile der Netz-Datenbasis und ist wie folgt deklariert:

```
struct netent {
   char *n_name;           /* offizieller Name des Netzes*/
   char **n_aliases;       /* Alias-Liste */
   int n_addrtype;         /* Adress-Typ */
   in_addr_t n_net;        /* Netzadresse */
};
```

Beschreibung der *netent*-Komponenten:

n_name
: Offizieller Name des Netzes

n_aliases
: Eine durch null abgeschlossene Liste mit alternativen Namen für das Netz

n_addrtype
Typ der Adresse, die geliefert wird (immer AF_INET)

n_net
Adresse des Netzes. Netzadressen werden in Rechner-Byte-Reihenfolge zurückgeliefert.

getnetent() liest die nächste Zeile der Datei. Wenn nötig, öffnet *getnetent()* vorher die Datei.

setnetent() öffnet die Datei und setzt sie auf den Anfang zurück. Wenn das Flag *stayopen* ungleich null ist, wird die Datenbasis nach keinem Aufruf von *getnetent()* geschlossen (weder direkt noch indirekt durch einen der anderen *getnet...()*-Aufrufe).

endnetent() schließt die Datei.

getnetbyname() und *getnetbyaddr()* durchsuchen die Datei sequenziell von Anfang an, bis

– ein passender Netzname gefunden ist oder
– die passende Netzadresse gefunden ist oder
– das Ende der Datei erreicht ist.

Returnwert

Erreicht die Suche das Ende der Datei, so wird der Null-Zeiger zurückgeliefert.

Hinweis

Alle Informationen befinden sich in einem statischen Bereich. Die Informationen müssen somit kopiert werden, wenn sie gesichert werden sollen.
Es werden nur die Internet-Protokolle unterstützt.

Siehe auch

/etc/inet/networks

getpeername() - Namen des Kommunikationspartners abfragen

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

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

Beschreibung

Die Funktion *getpeername()* liefert den Namen des Kommunikationspartners, der mit dem Socket *s* verbunden ist.

name zeigt auf einen Speicherbereich. **name* enthält nach erfolgreicher Ausführung von *getpeername()* die Adresse des Kommunikationspartners.

Die *size_t*-Variable, auf die der Parameter *namelen* zeigt, gibt zu Beginn die Größe des durch *name* referenzierten Speicherbereichs an. Bei Rückkehr der Funktion enthält **namelen* die aktuelle Größe (in Bytes) des zurückgelieferten Namens.

Returnwert

0:
bei Erfolg

-1:
bei Fehler. *errno* wird gesetzt, um den Fehler anzuzeigen.

Fehler

EBADF
Der Parameter *s* ist kein gültiger Deskriptor.

EFAULT
Der Parameter *name* zeigt auf einen Bereich außerhalb des Prozess-Adressbereichs.

ENOTCONN
Der Socket hat keine Verbindung.

ENOTSOCK
Der Deskriptor *s* referenziert eine Datei, keinen Socket.

Siehe auch

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

getprotoent(), getprotobynumber(), getprotobyname(), setprotoent(), endprotoent() - Informationen über Protokolle abfragen

```
#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);
```

Beschreibung

Die Funktionen *getprotobyname()*, *getprotobynumber()* und *getprotoent()* liefern Informationen über die verfügbaren Services. Diese Funktionen greifen auf die UFS-Datei */etc/inet/protocols* zu. Die Schnittstelle wird aus Kompatibilitätsgründen angeboten. Der Inhalt der Datei ist in EBCDIC kodiert.

Die Funktionen *getprotobyname()*, *getprotobynumber()* und *getprotoent()* liefern einen Zeiger auf ein Objekt der nachfolgend beschriebenen Struktur *protoent* zurück.

Die Struktur *protoent* entspricht den Feldern einer Zeile der Protokoll-Datenbasis */etc/inet/protocols* und ist wie folgt deklariert:

```
struct protoent {
   char *p_name;           /* offizieller Name des Protokolls*/
   char **p_aliases;       /* Alias-Liste */
   int p_proto;            /* Protokollnummer */
};
```

Beschreibung der **protoent()*-Komponenten:

p_name
: Name des Protokolls

p_aliases
: Eine durch null abgeschlossene Liste mit alternativen Namen für das Protokoll

p_proto
: Nummer des Protokolls

getprotoent() liest die nächste Zeile der Datei. Wenn nötig, öffnet *getprotoent()* vorher die Datei.

setprotoent() öffnet die Datei und setzt sie auf den Anfang zurück. Wenn das Flag *stayopen* ungleich null ist, wird die Datenbasis nach keinem Aufruf von *getprotoent()* geschlossen (weder direkt noch indirekt durch einen der anderen *getproto...()*-Aufrufe).

endprotoent() schließt die Datei.

getprotobyname() und *getprotobynumber()* durchsuchen die Datei sequenziell von Anfang an, bis

– ein passender Protokollname gefunden ist oder
– die passende Protokollnummer gefunden ist oder
– das Ende der Datei erreicht ist.

Returnwert

Erreicht die Suche das Ende der Datei, so wird der Null-Zeiger zurückgeliefert.

Hinweis

Alle Informationen befinden sich in einem statischen Bereich. Die Informationen müssen somit kopiert werden, wenn sie gesichert werden sollen.

Siehe auch

/etc/inet/protocols

getservent(), getservbyport(), getservbyname(), setservent(), endservent() - Informationen über Services abfragen

```
#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);
```

Beschreibung

Die Funktionen *getservbyport()*, *getservbyname()* und *getservent()* liefern Informationen über die verfügbaren Services. Jede dieser Funktionen liefert einen Zeiger auf ein Objekt der nachfolgend beschriebenen Struktur *servent*.

Die Struktur *servent* entspricht den Feldern einer Zeile der Service-Datenbasis */etc/inet/services* und ist wie folgt deklariert:

```
struct servent {
   char *s_name;            /* Name des Service */
   char **s_aliases;        /* Alias-Liste */
   int s_port;              /* Nummer des Ports, auf dem der Service liegt */
   char *s_proto;           /* verwendetes Protokoll */
};
```

Beschreibung der *servent*-Komponenten:

s_name
: Name des Service

s_aliases
: Eine durch null abgeschlossene Liste mit alternativen Namen für den Service

s_port
: Portnummer, die dem Service zugeordnet ist. Portnummern werden in Netz-Byte-Reihenfolge zurückgeliefert.

s_proto
Name des Protokolls, das verwendet werden muss, um den Service anzusprechen.

getservent() liest die nächste Zeile der Datei. Wenn nötig, öffnet *getservent()* vorher die Datei.

setservent() öffnet die Datei und setzt sie auf den Anfang zurück. Ist das Flag *stayopen* ungleich null, so wird die Datenbasis nach keinem Aufruf von *getservent()* geschlossen (weder direkt noch indirekt durch einen der anderen *getserv...()*-Aufrufe).

endservent() schließt die Datei.

getservbyname() und *getservbyport()* durchsuchen die Datei sequenziell von Anfang an, bis

– ein passender Service-Name gefunden ist oder
– die passende Portnummer gefunden ist oder
– das Ende der Datei erreicht ist.

Sofern ein Protokollname (nicht NULL) angegeben ist, suchen *getservbyname()* und *getservbyport()* nach dem Service, der das passende Protokoll verwendet.

Returnwert

Erreicht die Suche das Ende der Datei, so wird der Null-Zeiger zurückgeliefert.

Hinweis

Die Informationen über Services und deren Portnummern sind in BCAM nicht vorhanden, da diese Bestandteil der OSI-Schicht 7 sind. Da die Zuordnung von Portnummern zu Services statisch ist, kann die Implementierung mittels einer UFS-Datei */etc/inet/services* (wie in UNIX-Systemen üblich) gelöst werden, indem die Services dort eingetragen werden.

Alle Informationen befinden sich in einem statischen Bereich. Die Informationen müssen somit kopiert werden, wenn sie gesichert werden sollen.

Siehe auch

getprotoent(), /etc/inet/services

getsockname() - Namen des Sockets abfragen

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

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

Beschreibung

Die Funktion *getsockname()* liefert den aktuellen Namen für den Socket *s*.

name zeigt auf einen Speicherbereich. **name* enthält nach erfolgreicher Ausführung von *getsockname()* den Namen (Adresse) des Sockets *s*. Das tatsächliche Format der *sockaddr*-Struktur hängt von der Adressfamilie ab und ist im Abschnitt „Socket-Adressierung“ auf Seite 13 beschrieben.

Die *size_t*-Variable, auf die der Parameter *namelen* zeigt, gibt zu Beginn die Größe des durch *name* referenzierten Speicherbereichs an. Bei Rückkehr der Funktion enthält **name* die aktuelle Größe (in Bytes) des zurückgelieferten Namens.

Returnwert

0:
bei Erfolg.

-1:
bei Fehler. *errno* wird gesetzt, um den Fehler anzuzeigen

Fehler

EBADF
Der Parameter *s* ist kein gültiger Deskriptor.

EFAULT
Der Parameter *name* zeigt auf einen Bereich außerhalb des Prozess-Adressbereichs.

ENOTSOCK
Der Deskriptor *s* referenziert eine Datei, keinen Socket.

Siehe auch

bind(), getpeername(), socket()

getsockopt(), setsockopt() - Socket-Optionen abfragen und setzen

```
#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);
```

Beschreibung

Die Funktionen *getsockopt()* und *setsockopt()* greifen auf Optionen zu, die für einen Socket festgelegt sind. Optionen kann es auf verschiedenen Protokoll-Ebenen geben. Immer gibt es sie auf der höchsten Protokoll-Ebene.

Für den Zugriff auf eine Socket-Option muss der Name *optname* der Option angegeben werden sowie die Protokoll-Ebene *level*, auf der die Option interpretiert wird. Um auf Optionen auf der Socket-Ebene zuzugreifen, muss der Benutzer SOL_SOCKET bzw. IPPROTO_IPV6 für *level* angeben.

Bei der Funktion *setsockopt()* kann der Benutzer über die Parameter *optval* und *optlen* auf Option-Werte zugreifen. Bei der Funktion *getsockopt()* identifizieren *optval* und *optlen* einen Puffer, in dem der Wert der gewünschten Option(en) zurückgeliefert wird. Bei *getsockopt()* enthält **optlen* zu Beginn die Größe des Puffers, auf den *optval* zeigt. Bei Rückkehr der Funktion *getsockopt()* enthält **optlen* die aktuelle Größe des zurückgelieferten Puffers. Hat die Option keinen Wert, der zurückgeliefert werden kann, so erhält **optval* den Wert 0.

optname und alle angegebenen Optionen werden ohne Interpretation an das zuständige Protokoll-Modul zur Interpretation weitergereicht. Die Include-Datei <sys/socket.h> enthält Definitionen für die Optionen der Socket-Ebene. Die Optionen sind auf Seite 113 beschrieben.

Bei den meisten Optionen der Socket-Ebene ist *optval* ein Zeiger auf einen Parameter vom Typ Integer. Wenn für *setsockopt()* eine boolesche Operation zugelassen werden soll, darf der Parameter *optval* nicht der Null-Zeiger sein. Wenn eine boolsche Operation nicht zugelassen werden soll, muss der Paramerter *optval* der Null-Zeiger sein. Die Option SO_LINGER verwendet einen Parameter vom Datentyp *struct linger*, der in der Include-Datei <sys/socket.h> definiert ist. Dieser Parameter spezifiziert den gewünschten Status der Option und das Verzögerungs-Interval (siehe Seite 113).

Die nachfolgend beschriebenen Optionen sind auf der Socket-Ebene bekannt. Wenn bei der Beschreibung der Optionen nichts anderes gesagt ist, kann jede Option mit *getsockopt()* abgefragt und mit *setsockopt()* gesetzt werden.

Beschreibung der Socket-Optionen

SO_KEEPALIVE
: gibt an, ob Verbindungen aufrecht erhalten werden oder nicht.
SO_KEEPALIVE veranlasst die regelmäßige Übertragung von Kontrollnachrichten über einen verbundenen Socket. Sollte das verbundene Partnerendsystem diese Nachricht nicht beantworten können, wird die Verbindung als unterbrochen betrachtet. Ein Prozess, der darauf wartet, auf den Socket zu schreiben, erhält ein SIGPIPE-Signal und die Schreib-Operation liefert einen Fehler zurück. Voreingestellt ist, dass sich ein Prozess beendet, wenn er ein SIGPIPE-Signal erhält. Eine Lese-Operation auf dem Socket liefert einen Fehler zurück, erzeugt aber kein SIGPIPE-Signal. Wartet der Prozess bei unterbrochener Verbindung auf einen *select()*-Aufruf, so liefert *select()* den Wert *true* für alle Lese- oder Schreib-Ereignisse, die für den Socket selektiert sind.

SO_LINGER
: zeigt an, ob nach einem Aufruf von *close()* das Schließen des Sockets hinausgezögert wird, wenn noch Daten auf dem Socket zum Übertragen anstehen. SO_LINGER steuert die Aktion, die ausgelöst wird, wenn nicht-gesendete Nachrichten in der Warteschlange des Sockets warten und die Funktion *close()* aufgerufen wird. Wenn der Socket eine gesicherte Übertragung von Daten garantiert und SO_LINGER gesetzt ist, blockiert das System den Prozess, der versucht den Socket zu schließen. Beim Aufruf von *setsockopt()* wird das Timeout-Zeitintervall festgelegt, wenn SO_LINGER eingeschaltet ist. Wenn SO_LINGER bei Aufruf der Funktion *close()* ausgeschaltet ist, führt das System die Funktion *close()* sofort aus und der Prozess kann so schnell wie möglich fortgesetzt werden.

SO_BROADCAST
: zeigt an, ob Broadcast-Nachrichten übermittelt werden dürfen oder nicht.
Da im BS2000 Broadcast-Nachrichten immer gesendet werden dürfen, hat diese Option keine funktionelle Bedeutung.
Zu beachten ist jedoch, dass der Empfang von Brodcast-Nachrichten nicht zugelassen sein kann (siehe BCAM-Kommando BCOPTION im Handbuch „openNet Server V3.0 (BS2000/OSD)“).

SO_REUSEADDR
: Gibt an, dass die Regeln für die Gültigkeitsprüfung der für *bind()* angegebenen Adressen die Wiederverwendung lokaler Adressen zulassen sollen, sofern dies vom Protokoll unterstützt wird. Für die Option ist ein ganzzahliger Wert (*int*) erforderlich.

SO_TYPE
fragt den Socket-Typ ab.
SO_TYPE wird nur von *getsockopt()* verwendet. SO_TYPE liefert den Typ des Sockets, also entweder SOCK_STREAM oder SOCK_DGRAM. Dies kann für Server nützlich sein, die bei ihrem Start Sockets erben.

SO_ACCEPTCONN
zeigt an, ob der Socket empfangsbereit für Verbindungsanforderungen ist.
SO_ACCEPTCONN kann nur mit *getsockopt()* verwendet werden.

Returnwert

0:
bei Erfolg.

-1:
bei Fehler. *errno* wird gesetzt, um den Fehler anzuzeigen.

Fehler

EBADF
Der Parameter *s* ist kein gültiger Deskriptor.

EFAULT
optval zeigt nicht auf einen gültigen Teil des Prozess-Adressbereichs in der Länge *optlen*.

EINVAL
Einer der Parameter *level*, *optval* oder *optlen* hat einen ungültigen Wert.

ENOPROTOOPT
Die Option ist für die bezeichnete Ebene unbekannt.

ENOTSOCK
Der Deskriptor *s* referenziert keinen Socket.

EOPNOTSUPP
Die Option wird nicht unterstützt.

Siehe auch

socket(), getprotoent()

inet_addr(), inet_network(), inet_makeaddr(), inet_lnaof(), inet_netof(), inet_ntoa() - IPv4-Internet-Adresse manipulieren

```
#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);
```

Beschreibung

Die Verwendung der Funktionen *inet_addr()*, *inet_lnaof()*, *inet_makeaddr()*, *inet_netof()*, *inet_network()* und *inet_ntoa()* ist nur in der Adressfamilie AF_INET sinnvoll.

Die Funktion *inet_addr()* konvertiert die Zeichenkette, auf die der Parameter *cp* zeigt, von der im Internet üblichen Punktschreibweise in einen ganzzahligen Wert. Dieser ganzzahlige Wert kann dann als Internet-Adresse verwendet werden.

Die Funktion *inet_lnaof()* extrahiert aus der im Parameter *in* übergebenen Internet-Rechneradresse die lokale Netzadresse in der Byte-Reihenfolge des Rechners.

Die Funktion *inet_makeaddr()* erstellt eine Internet-Adresse aus

– dem im Parameter *net* angegebenen Subnetz-Anteil der Internet-Adresse und
– dem im Parameter *lna* angegebenen subnetz-lokalen Adressteil.

Subnetz-Anteil der Internet-Adresse und subnetz-lokaler Adressteil werden jeweils in Byte-Reihenfolge des Rechners übergeben.

Die Funktion *inet_netof()* extrahiert aus der im Parameter *in* übergebenen Internet-Rechneradresse die Netznummer in der Byte-Reihenfolge des Rechners.

Die Funktion *inet_network()* konvertiert die Zeichenkette, auf die der Zeiger *cp* zeigt von der im Internet üblichen Punktschreibweise in einen ganzzahligen Wert. Dieser ganzzahlige Wert kann dann als Subnetz-Anteil der Internet-Adresse verwendet werden.

Die Funktion *inet_ntoa()* konvertiert die im Parameter *in* übergebene Internet-Rechneradresse in eine Zeichenkette gemäß der üblichen Internet-Punktschreibweise.

Alle Internet-Adressen werden in der Netz-Byte-Reihenfolge zurückgeliefert. In der Netz-Byte-Reihenfolge sind die Bytes von links nach rechts angeordnet.

In Punktschreibweise angegebene Werte können in den folgenden Formaten vorliegen:

– a.b.c.d
  Bei Angabe einer vierteiligen Adresse wird jeder Teil als ein Daten-Byte interpretiert und von links nach rechts den vier Bytes einer Internet-Adresse zugeordnet.

– a.b.c
  Bei Angabe einer dreiteiligen Adresse wird der letzte Teil als 16-bit-Sequenz interpretiert und in den beiden rechten Bytes der Internet-Adresse abgelegt. Auf diese Weise können dreiteilige Adressformate problemlos zur Angabe von Class-B-Adressen verwendet werden (z.B. 128.*net.host*).

– a.b
  Bei Angabe einer zweiteiligen Adresse wird der letzte Teil als 24-bit-Sequenz interpretiert und in den drei rechten Bytes der Internet-Adresse abgelegt. Auf diese Weise können zweiteilige Adressformate problemlos zur Angabe von Class-A-Adressen verwendet werden (z.B. *net.host*).

– a
  Bei Angabe einer einteiligen Adresse wird der Wert ohne Änderung der Byte-Reihenfolge direkt in der Netzadresse abgelegt.

Bei den Zahlen, die als Adressteile in Punktschreibweise angegeben sind, kann es sich um Dezimal-, Oktal- oder Sedezimalzahlen handeln:

– Zahlen, denen weder 0 noch 0x bzw. 0X vorangestellt ist, werden als Dezimalzahlen interpretiert.

– Zahlen, denen 0 vorangestellt ist, werden als Oktalzahlen interpretiert.

– Zahlen, denen 0x oder 0X vorangestellt ist, werden als Sedezimalzahlen interpretiert.

Returnwert

Bei erfolgreicher Ausführung liefert *inet_addr()* die Internet-Adresse zurück. Andernfalls wird (*in_addr_t*) -1 zurückgeliefert.

Bei erfolgreicher Ausführung liefert *inet_network()* die umgesetzte Internet-Nummer zurück. Andernfalls wird (*in_addr_t*) -1 zurückgeliefert.

Die Funktion *inet_makeaddr()* liefert die erstellte Internet-Adresse zurück.

Die Funktion *inet_lnaof()* liefert die lokale Netzadresse zurück.

Die Funktion *inet_netof()* liefert die Netznummer zurück.

Die Funktion *inet_ntoa()* liefert einen Zeiger auf die Netzadresse in der für Internet üblichen Punktschreibweise zurück.

Fehler

Es sind keine Fehler definiert.

Hinweis

Der Return-Wert von *inet_ntoa()* zeigt möglicherweise auf statische Daten, die durch nachfolgende Aufrufe von *inet_ntoa()* überschrieben werden können.

Siehe auch

gethostent(), getnetent()

inet_ntop(), inet_pton() - Internet-Adressen manipulieren

```
#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);
```

Beschreibung

Die Funktion *inet_ntop()* konvertiert die binäre IP-Adresse, auf die der Parameter *addr* verweist, in eine abdruckbare Notation. Dabei legt der Wert des Parametes *af* fest, ob es sich um eine IPv4-Adresse oder um eine IPv6-Adresse handelt:

– AF_INET: Es wird eine IPv4-Adresse konvertiert.
– AF_INET6: Es wird eine IPv6-Adresse konvertiert.

Die Funktion *inet_ntop()* liefert die abdruckbare Adresse in dem Puffer der Länge *size* zurück, auf den der Zeiger *dst* verweist. Eine ausreichende Dimensionierung des Puffers ist gewährleistet, wenn die Größe den Integer-Konstanten INET_ADDRSTRLEN (für IPv4-Adressen) bzw. INET6_ADDRSTRLEN (für IPv6-Adressen) entspricht. Beide Konstanten sind in <netinet/in.h> definiert.

Die Funktion *inet_pton()* konvertiert eine IPv4-Adresse in dezimaler Punkt-Notation oder eine IPv6-Adresse in sedezimaler Doppelpunkt-Notation in eine binäre Adresse. Dabei legt der Wert des Parametes *af* fest, ob es sich um eine IPv4-Adresse oder um eine IPv6-Adresse handelt:

– AF_INET: Es wird eine IPv4-Adresse konvertiert.
– AF_INET6: Es wird eine IPv6-Adresse konvertiert.

Die Funktion *inet_pton()* liefert die binäre Adresse in dem Puffer zurück, auf den der Zeiger *dst* verweist. Der Puffer muss ausreichend dimensioniert sein: 4 byte bei AF_INET und 16 byte bei AF_INET6.

Hinweis

Wenn der Output von *inet_pton()* als Input für eine neue Funktion dienen soll, achten Sie darauf, dass die Anfangsadresse des Zielbereiches *dst* eine Doppelwort-Ausrichtung hat.

Returnwert

inet_ntop() liefert bei erfolgreicher Ausführung einen Zeiger auf den Puffer zurück, in dem der Textstring abgelegt ist. Im Fehlerfall wird ein Null-Zeiger zurückgeliefert.

inet_pton() liefert folgende Werte zurück:

1:
bei erfolgreicher Konvertierung.

0:
wenn der Input kein gültiger Adress-String ist.

-1:
wenn ein Parameter ungültig ist.

Fehleranzeige durch *errno*

EAFNOSUPPORT
Ungültiger Parameter *af* angegeben, oder die Funktion wird an diesem System nicht unterstützt. Vergleichen Sie dazu auch den Abschnitt „Abhängigkeiten vom BS2000/OSD-Transportsystem BCAM“ auf Seite 316.

ENOSPC
Ergebnispuffer zu klein

listen() - Socket auf anstehende Verbindungen überprüfen

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

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

Beschreibung

Die Funktion *listen()* veranlasst, dass der Socket *s* für die Annahme von Verbindungsanforderungen zugelassen wird und überprüft dann den Socket auf anstehende Verbindungsanforderungen. Zu diesem Zweck richtet *listen()* für den Socket *s* eine Warteschlange für eingehende Verbindungsanforderungen ein. Mit dem Parameter *backlog* gibt der Benutzer an, wie viele Verbindungsanforderungen die Warteschlange maximal aufnehmen kann. Der Wert für *backlog* ist auf maximal 50 begrenzt.
Die Funktion *listen()* kann nur für Sockets vom Typ SOCK_STREAM aufgerufen werden.

Damit ein Prozess über einen Socket mit dem Partner kommunizieren kann, der Verbindungsanforderungen schickt, sind folgende Schritte erforderlich:

1. einen Socket erzeugen (*socket()*) und binden (*bind()*)
2. mit *listen()* für den Socket eine Warteschlange für eingehende Verbindungsanforderungen spezifizieren
3. Verbindungsanforderungen mit *accept()* annehmen

Wenn eine Verbindungsanforderung bei voller Warteschlange ankommt, erhält der Socket, der die Verbindungsanforderung geschickt hat, die Fehlermeldung ECONNREFUSED oder ETIMEDOUT.

Returnwert

0:
bei Erfolg.

-1:
bei Fehler. *errno* wird gesetzt, um den Fehler anzuzeigen.

Fehler

EBADF
Der Parameter *s* ist kein gültiger Deskriptor.

ENOTSOCK
Der Deskriptor *s* referenziert eine Datei, keinen Socket.

EOPNOTSUPP
Der Socket-Typ wird von *listen()* nicht unterstützt.

Siehe auch

accept(), connect(), socket()

recv(), recvfrom(), recvmsg() - Nachricht von einem Socket empfangen

```
#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);
```

Beschreibung

Die Funktionen *recv()*, *recvfrom()* und *recvmsg()* empfangen Nachrichten von einem Socket. *recv()* kann Nachrichten nur von einem Socket empfangen, über den eine Verbindung aufgebaut ist (siehe Funktion *connect()* auf Seite 87).
recvfrom() und *recvmsg()* können Nachrichten von einem Socket mit oder ohne Verbindung empfangen.

Der Parameter *s* bezeichnet den Socket, von dem die Nachricht empfangen wird.
Wenn der Parameter *from* nicht der Null-Zeiger ist, wird in dem durch *from* referenzierten Speicherbereich die Absenderadresse der Nachricht abgelegt.
fromlen ist ein Ergebnisparameter. Zu Beginn enthält die *size_t*-Variable, auf die *fromlen* zeigt, die Größe des durch *from* referenzierten Puffers. Bei Rückkehr der Funktion enthält **fromlen* die aktuelle Länge der Adresse, die in **from* gespeichert ist. Die Funktion liefert die Länge der Nachricht zurück.

Bei einem Datagramm-Socket muss die gesamte Nachricht in einer einzigen Operation gelesen werden. Wenn der angegebene Nachrichtenpuffer zu klein und MSG_PEEK im Parameter *flags* nicht gesetzt ist, werden die über die Puffergröße hinausgehenden Daten gelöscht.
Bei einem Stream-Socket werden Nachrichtengrenzen ignoriert. Sobald Daten verfügbar sind, werden sie an den Anrufer zurückgeliefert; es werden keine Daten gelöscht.

Wenn auf dem Socket keine Nachrichten vorhanden sind, wartet der Empfangsaufruf auf eine ankommende Nachricht, es sei denn der Socket ist nicht-blockierend (siehe *ioctl()* auf Seite 140). In diesem Fall wird -1 zurückgeliefert, wobei die Variable *errno* auf den Wert EWOULDBLOCK gesetzt wird.

Mit den Funktionen *poll()* bzw. *select()* kann festgestellt werden, wann weitere Daten ankommen.

Wenn der Prozess, der *recv()*, *recvfrom()* oder *recvmsg()* aufruft, ein Signal erhält, bevor irgendwelche Daten vorhanden sind, wird im Standardfall die betreffende Funktion erneut aufgerufen. Nicht aufgerufen wird die Funktion jedoch, wenn der aufrufende Prozess ausdrücklich mit *sigaction()* das Signal gibt, diese Aufrufe zu unterbrechen (siehe auch Handbuch „C-Bibliotheksfunktionen (BS2000/OSD) für POSIX-Anwendungen“).

Der Parameter *flags* gibt die Art des Nachrichtenempfangs an:

MSG_PEEK
: empfängt eine ankommende Nachricht. Die Daten werden jedoch als ungelesen behandelt und die nächste Empfangsfunktion liefert diese Daten auch wieder zurück.

MSG_WAITALL
: Die Funktion blockiert solange, bis die gesamte angeforderte Datenmenge zurückgegeben werden kann.
In folgenden Fällen kann eine geringere Datenmenge zurückgegeben werden:

  – Ein Signal trifft ein.
  – Die Verbindung wird beendet.
  – Eine Fehlersituation tritt auf.

Die Funktion *recvmsg()* verwendet die Struktur *msghdr*, um die Anzahl der direkt versorgten Parameter zu verringern. Die Struktur *msghdr* ist in der Include-Datei <sys/socket.h> wie folgt deklariert:

```
struct msghdr {
   void         *msg_name;            /* optionale Adresse */
   size_t        msg_namelen;         /* Länge der Adresse */
   struct iovec *msg_iov;             /* scatter/gather-Felder */
   int           msg_iovlen;          /* Anzahl der Elemente in msg-iov */
   caddr_t       msg_accrights;       /* Zugriffsrechte Senden/Empfangen */
   int           msg_accrightslen;
   void         *msg_control;         /* Hilfsdaten */
   size_t        msg_controllen;      /* Länge des Puffers für Hilfsdaten */
   int           msg_flags;           /* flag für empfangene Nachricht */
};
```

Die Elemente *msg_name* und *msg_namelen* enthalten Absende-Adresse und Adresslänge des Partners, wenn der Socket keine aufgebaute Verbindung hat. Die Partneradresse ist eine *sockaddr*-Struktur. Das tatsächliche Format der *sockaddr*-Struktur hängt von der Adressfamilie ab und ist im Abschnitt „Socket-Adressierung“ auf Seite 13 beschrieben. Wenn der Socket eine aufgebaute Verbindung hat, kann *msg_name* als Null-Zeiger übergeben werden.

Die Elemente msg_iov und *msg_iovlen* beschreiben die Scatter- und Gather-Felder. Das Senden und Empfangen von Zugriffsrechten wird nicht unterstützt.

Returnwert

>0:
bei Erfolg. Der Wert gibt die Anzahl der empfangenen Bytes an.

=0:
bei Erfolg. Es können keine Daten mehr empfangen werden. Der Partner hat seine Verbindung ordentlich geschlossen (nur bei Sockets vom Typ SOCK_STREAM).

-1:
bei Fehler. *errno* wird gesetzt, um den Fehler anzuzeigen.

Fehler

EBADF
Der Parameter *s* ist kein gültiger Deskriptor.

ECONNRESET
Die Verbindung zum Partner wurde abgebrochen (nur bei Sockets vom Typ SOCK_STREAM).

EFAULT
Die Daten sollen in einem nicht vorhandenen oder geschützten Teil des Prozess-Adressbereichs empfangen werden.

EINTR
Der aufrufende Prozess hat ein Signal empfangen, bevor irgendwelche Daten empfangen werden konnten. Das Signal zum Unterbrechen des Funktionsaufrufs ist gesetzt.

EINVAL
Es wurden mehr als MSG_MAXIOVLEN Scatter/Gather-Felder spezifiziert.

EIO
Es sind Benutzerdaten verloren gegangen.

ENETDOWN
Die Verbindung zum Netzwerk ist nicht mehr aktiv.

ENOTCONN
Für den Socket besteht keine Verbindung.

ENOTSOCK
Der Deskriptor *s* referenziert eine Datei, keinen Socket.

EOPNOTSUPP
: Der Parameter *flags* enthält einen ungültigen Wert oder *msg->msg_accrights* wurde spezifiziert.

EWOULDBLOCK
: Der Socket ist als nicht-blockierend markiert, und die geforderte Operation würde blockieren.

Siehe auch

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

send(), sendto(), sendmsg() - Nachricht von Socket zu Socket senden

```
#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);
```

Beschreibung

Die Funktionen *send()*, *sendto()* und *sendmsg()* senden Nachrichten von einem Socket an einen anderen. *send()* kann nur bei einem Socket benutzt werden, über den eine Verbindung aufgebaut ist (siehe Funktion *connect()* auf Seite 87). *sendto()* und *sendmsg()* können immer verwendet werden.

Der Parameter *s* bezeichnet den Socket, von dem eine Nachricht gesendet wird. Die Zieladresse wird mit *to* übergeben, wobei *tolen* die Größe der Zieladresse angibt.
Die Länge der Nachricht wird mit *len* angegeben. Wenn die Nachricht zu lang ist, um von der darunter liegenden Protokollebene ganz transportiert zu werden, wird der Fehler EMSGSIZE geliefert und die Nachricht wird nicht übermittelt.

Der Parameter *flags* wird zurzeit nicht unterstützt. Ein Wert ungleich 0 führt zu einem Fehler, wobei die Variable *errno* auf den Wert EOPNOTSUPP gesetzt wird.

Empfängt der Prozess, der *send()*, *sendmsg()* oder *sendto()* aufruft, ein Signal, bevor irgendwelche Daten zum Senden gepuffert werden, so wird im Standardfall der betreffende Funktionsaufruf erneut durchgeführt. Nicht durchgeführt wird der Funktionsaufruf jedoch, wenn der aufrufende Prozess ausdrücklich mit *sigaction()* das Signal setzt, diesen Aufruf zu unterbrechen (siehe auch Handbuch „C-Bibliotheksfunktionen (BS2000/OSD) für POSIX-Anwendungen“).

Die Funktion *sendmsg()* verwendet die Struktur *msghdr*, um die Anzahl der direkt zu versorgenden Parameter zu verringern.

Die Struktur *msghdr* ist in der Include-Datei <sys/socket.h> wie folgt definiert:

```
struct msghdr {
   void         *msg_name;            /* optionale Adresse */
   size_t        msg_namelen;         /* Länge der Adresse */
   struct iovec *msg_iov;             /* scatter/gather-Felder */
   int           msg_iovlen;          /* Anzahl der Elemente in msg-iov */
   caddr_t       msg_accrights        /* Zugriffsrechte Senden/Empfangen */
   int           msg_accrightslen;
   void         *msg_control;         /* Hilfsdaten */
   size_t        msg_controllen;      /* Länge des Puffers für Hilfsdaten */
   int           msg_flags;           /* flag für empfangene Nachricht */
};
```

msg->msg_name und *msg->msg_namelen* spezifizieren die Zieladresse, wenn der Socket keine aufgebaute Verbindung hat. *msg->msg_name* kann als Null-Zeiger übergeben werden, wenn keine Namen gewünscht oder gefordert werden. Wie Sie dem Socket eine Adresse zuweisen erläutert der Abschnitt „Socket-Adressierung“ auf Seite 13.
msg->msg_iov und *msg->msg_iovlen* beschreiben die Scatter- und Gather-Felder. Das Senden und Empfangen von Zugriffsrechten wird nicht unterstützt.

Returnwert

≥0:
bei Erfolg. Der Wert gibt die Anzahl der gesendeten Bytes an.

-1:
bei Fehler. *errno* wird gesetzt, um den Fehler anzuzeigen. Die Deskriptormengen werden dann nicht verändert.

Fehler

EBADF
Der Parameter *s* ist kein gültiger Deskriptor.

ECONNRESET
Die Verbindung zum Partner wurde abgebrochen (nur bei Sockets vom Typ SOCK_STREAM).

EDESTADDRREQ
Der Socket ist nicht verbindungsorientiert, ein fester Partner wurde nicht festgelegt und im Aufruf wurde kein Partner angegeben.

EFAULT
Die Daten sollen in einen nicht vorhandenen oder geschützten Teil des Prozess-Adressbereichs gesendet werden.

EHOSTUNREACH
Der Zielrechner ist nicht erreichbar.

EINTR
Der aufrufende Prozess hat ein Signal empfangen, bevor irgendwelche Daten zum Senden gepuffert werden konnten, und das Signal zum Unterbrechen des Funktionsaufrufs ist gesetzt.

EINVAL
Ein Parameter spezifiziert einen ungültigen Wert.

EMSGSIZE
Die Nachricht ist zu groß, um auf einmal gesendet zu werden.

ENETDOWN
Die Verbindung zum Netzwerk ist nicht mehr aktiv.

ENOBUFS
Die Ausgabe-Warteschlange für ein Netz-Interface ist voll. Dies führt generell dazu, dass das Interface aufhört zu senden, kann aber auf einem vorübergehenden Stau beruhen.

ENOTCONN
Für den Socket besteht keine Verbindung.

ENOTSOCK
Der Deskriptor *s* referenziert keinen Socket.

EOPNOTSUPP
Der Parameter *flags* oder *msg->msg_accrights* wurde spezifiziert. Dies wird jedoch nicht unterstützt.

EPIPE
Der Socket ist nicht für Schreiben aktiviert, oder der Socket ist verbindungsorientiert und der Partner hat die Verbindung beendet.
Wenn der Socket vom Typ SOCK_STREAM ist, wird das Signal SIGPIPE für den aufrufenden Prozess generiert.

EWOULDBLOCK
Der Socket ist als nicht-blockierend markiert und die geforderte Operation würde blockieren.

EAFNOSUPPORT
Adressen der bei *sendto()* bzw. *sendmsg()* angegebenen Adressfamilie können mit diesem Socket nicht verwendet werden.

Wenn AF_UNIX die Adressfamilie des Sockets ist, kann die Ausführung von send(), sendto() und sendmsg() auch aus folgenden Gründen zu einem Fehler führen:

EACCES
: Für eine Komponente des Pfadnamens wird die Zugriffsberechtigung verweigert, oder die Berechtigung zum Schreiben auf den angegebenen Socket wird verweigert.

ENAMETOOLONG
: Eine Komponente eines Pfadnamens überschreitet NAME_MAX Zeichen, oder der gesamte Pfadname ist länger als PATH_MAX Zeichen.

ENOENT
: Eine Komponente des Pfadnamens verweist auf eine nicht vorhandene Datei, oder der Pfadname ist leer.

ENOTDIR
: Eine Komponente im Pfadnamen ist kein Verzeichnis.

Siehe auch

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

shutdown() - Voll-Duplex-Verbindung beenden

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

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

Beschreibung

Die Funktion *shutdown()* veranlasst, dass eine Seite oder beide Seiten einer Voll-Duplex-Verbindung über einen Socket beendet werden. Der Parameter *s* bezeichnet den betreffenden Socket.
In Abhängigkeit vom Wert des Parameters *how* bewirkt *shutdown()* Folgendes:

– Wenn der Parameter *how* den Wert SHUT_RD hat, verhindert *shutdown()*, dass weiterhin Nachrichten empfangen werden.

– Wenn der Parameter *how* den Wert SHUT_WR hat, verhindert *shutdown()*, dass Nachrichten gesendet werden.

– Wenn der Parameter *how* den Wert SHUT_RDWR hat, verhindert *shutdown()* sowohl das Empfangen als auch das Senden von Nachrichten.

Returnwert

0:
bei Erfolg

-1:
bei Fehler. *errno* wird gesetzt, um den Fehler anzuzeigen.

Fehler

EBADF
Der Parameter *s* ist kein gültiger Deskriptor.

ENOTSOCK
Der Deskriptor *s* referenziert eine Datei, keinen Socket.

ENOTCONN
Der Socket hat keine Verbindung.

Siehe auch

connect(), socket()

socket() - Socket erzeugen

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

int socket(int domain, int type, int protocol);
```

Beschreibung

Die Funktion *socket()* erzeugt einen Kommunikationsendpunkt und liefert einen Deskriptor zurück.

Der Parameter *domain* legt die Kommunikationsdomäne fest, in der die Kommunikation stattfinden soll. Dies bestimmt auch die zu verwendende Protokollfamilie. Die Protokollfamilie entspricht im Allgemeinen der Adressfamilie für die Adressen, die bei späteren Operationen auf dem Socket verwendet werden. Diese Familien werden in der Include-Datei <sys/socket.h> definiert. Es werden die Protokollfamilien AF_INET, AF_INET6 und AF_UNIX unterstützt.

Der Parameter *type* legt den Typ des Sockets und damit die Semantik der Kommunikation fest. Derzeit sind die beiden folgenden Socket-Typen definiert:

– SOCK_STREAM
– SOCK_DGRAM

Der Typ SOCK_STREAM bietet eine sequenzielle, gesicherte, bidirektionale Verbindung. Ein Socket vom Typ SOCK_DGRAM unterstützt die Übertragung von Datagrammen. Datagramme sind verbindungslose, ungesicherte Nachrichten einer festen maximalen Länge.

Der Parameter *protocol* legt ein bestimmtes Protokoll fest, das für den Socket verwendet werden soll. Da diese Implementierung nur die TCP/IP-Protokollfamilie unterstützt, sind hier nur die Werte 0 (Standardprotokoll), IPPROTO_IP, IPPROTO_IPV6, IPPROTO_TCP und IPPROTO_UDP gültig.

Sockets vom Typ SOCK_STREAM sind Voll-Duplex-Datenströme, ähnlich wie Pipes. Ein Stream-Socket muss in einem verbundenen Status sein, bevor irgendwelche Daten auf ihn gesendet oder von ihm empfangen werden können. Eine Verbindung zu einem anderen Socket wird mit der Funktion *connect()* hergestellt. Sind zwei Sockets einmal miteinander verbunden, können Daten mit *read()*- und *write()*-Aufrufen oder vergleichbaren Aufrufen wie *send()* und *recv()* übertragen werden. Wenn eine Sitzung beendet ist, sollte der Benutzer die Funktion *close()* aufrufen.

Die Kommunikationsprotokolle, die zum Implementieren eines Sockets vom Typ SOCK_STREAM verwendet werden, stellen sicher, dass Daten nicht verloren gehen oder verdoppelt werden.

Sockets vom Typ SOCK_DGRAM erlauben das verbindungslose Senden und Empfangen von Datagrammen mit *sendto()* und *recvfrom()* bzw. *sendmsg()* und *recvmsg()*. Bei den Aufrufen dieser Funktionen wird dann die Adresse des Kommunikationspartners als Parameter übergeben.

Mit der Funktion *fcntl()* kann der Benutzer eine Prozessgruppe angeben, um ein SIGIO-Signal zu empfangen, wenn Ein-/Ausgabe-Operationen oder Verbindungsaufbau-Wünsche ankommen.

Socket-Operationen werden von Optionen der Socket-Ebene gesteuert. Diese Operationen sind in der Include-Datei <sys/socket.h> definiert. Mit den Funktionen *getsockopt()* und *setsockopt()* kann der Benutzer diese Optionen überprüfen bzw. setzen.

Returnwert

≥0:
bezeichnet einen nicht-negativen Deskriptor bei Erfolg.

-1:
bei Fehler. *errno* wird gesetzt, um den Fehler anzuzeigen.

Fehler

EACCES
Es liegt keine Erlaubnis zum Erzeugen eines Sockets des angegebenen Typs oder Protokolls vor.

EMFILE
Die Tabelle der Deskriptoren pro Prozess ist voll.

ENFILE
Die Datei-Tabelle des Systems ist voll.

ENOBUFS
Es gibt nicht genug Speicherplatz im Puffer. Der Socket kann nicht erzeugt werden, bis genügend Speicher-Ressourcen frei gemacht werden.

EPROTONOSUPPORT
Der Protokolltyp oder das angegebene Protokoll wird in dieser Domäne nicht unterstützt.

EPROTOTYPE
Das Protokoll hat für den Socket den falschen Typ.

EAFNOSUPPORT
Die im Parameter *domain* angegebene Adressfamilie wird an diesem System nicht unterstützt. Vergleichen Sie dazu auch den Abschnitt „Abhängigkeiten vom BS2000/OSD-Transportsystem BCAM“ auf Seite 316.

Siehe auch

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

socketpair() - Ein Paar von verbundenen Sockets erzeugen

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

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

Beschreibung

Die Funktion *socketpair()* erzeugt ein Paar von miteinander verbundenen Sockets ohne Namen.

socketpair() erzeugt das Socket-Paar in der mit dem Parameter *domain* spezifizierten Adressfamilie (AF_INET oder AF_UNIX), vom Typ *type* (SOCK_STREAM oder SOCK_DGRAM) und unter Verwendung des optional angegebenen Protokolls *protocol*. Der Parameter *protocol* legt ein bestimmtes Protokoll fest, das für den Socket verwendet werden soll. Da diese Implementierung nur die TCP/IP-Protokollfamilie unterstützt, sind hier nur die Werte 0 (Standardprotokoll), IPPROTO_IP, IPPROTO_TCP und IPPROTO_UDP gültig.

In den Parametern *sv[0]* und *sv[1]* werden die Deskriptoren der neuen Sockets zurückgeliefert. Die beiden Sockets sind nicht zu unterscheiden.

Returnwert

0:
bei Erfolg

-1:
bei Fehler. *errno* wird gesetzt, um den Fehler anzuzeigen.

Fehler

EAFNOSUPPORT
Die angegebene Adressfamilie wird an diesem System nicht unterstützt.

EFAULT
Die Adresse *sv* spezifiziert keinen gültigen Teil des Prozess-Adressbereichs.

EMFILE
Die Tabelle der Deskriptoren pro Prozess ist voll.

ENFILE
Die Dateitabelle des Systems ist voll.

EOPNOTSUPP
Das angegebene Protokoll unterstützt nicht das Erzeugen von Socket-Paaren.

EPROTONOSUPPORT
Der Protokolltyp oder das angegebene Protokoll wird in dieser Domäne nicht unterstützt.

ENOMEM
Ein interner Ressourcen-Engpass ist aufgetreten.

Siehe auch

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

6.4 Verwendung von POSIX-Standardfunktionen für Sockets

Die im Folgenden beschriebenen Funktionen sind Standardfunktionen der POSIX-Bibliothek. Es handelt sich um die Funktionen

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

In diesem Abschnitt sind nur die Besonderheiten bei der Verwendung mit Sockets beschrieben.

close() - Socket schließen

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

Beschreibung

close() schließt den Socket *s* in Abhängigkeit von der Option SO_LINGER (siehe Funktion *setsockopt()* auf Seite 112).

Returnwert

0:
bei Erfolg.

-1:
bei Fehler. *errno* wird gesetzt, um den Fehler anzuzeigen.

Fehler

EBADF
Der Parameter *s* ist kein gültiger Deskriptor.

fcntl() - Sockets steuern

```
#include <fnctl.h>

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

Beschreibung

Die Funktion *fcntl()* führt Steuerfunktionen auch für Sockets aus.
s bezeichnet den Socket-Deskriptor. *cmd* wählt die auszuführende Steuerfunktion aus.

Folgende Steuerfunktionen werden für Sockets unterstützt:

F_DUPFD
: dupliziert einen Socket-Deskriptor.

F_GETFD
: ruft das Bit „schließen-bei-exec“ ab, das zu dem Deskriptor *s* gehört. Wenn das niederwertige Bit 0 ist, bleibt der Socket bei einem Aufruf von *exec()* offen, andernfalls wird der Socket bei Aufruf von *exec()* geschlossen.

F_SETFD
: setzt das zu *s* gehörende Bit „schließen-bei-exec“ auf das niederwertige Bit des ganzzahligen Wertes, der als dritter Parameter übergeben wird (0 oder 1 wie oben).

F_GETFL
: ruft das Dateistatus-Bit für *s* ab.

F_SETFL
: setzt das Dateistatus-Bit für *s* auf den ganzzahligen Wert, der als dritter Parameter übergeben wird. Nur bestimmte Bits können gesetzt werden (zum Beispiel O_NONBLOCK für nicht-blockierende Sockets).

F_SETOWN
: Für den spezifizierten Socket kann die Prozess-ID gesetzt werden, wodurch bei Eintreffen einer Nachricht des Prozesses ein SIGIO-Signal zugestellt wird.

F_GETOWN
: Die für den Socket gesetzte Prozess-ID wird zurückgeliefert.

Returnwert

0:
bei Erfolg

-1:
bei Fehler. *errno* wird gesetzt, um den Fehler anzuzeigen.

Fehler

EBADF
Der Parameter *s* ist kein gültiger Deskriptor.

EINVAL
cmd oder *arg* sind für dieses Gerät nicht gültig.

EIO
Es ist ein physikalischer Ein-/Ausgabefehler aufgetreten.

EMFILE
cmd ist F_DUPFD und im aufrufenden Prozess ist die Anzahl der offenen Dateideskriptoren gleich dem in der Konfiguration angegebenen Maximalwert der offenen Dateien für jeden Benutzer.

ioctl() - Sockets steuern

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

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

Beschreibung

Die Funktion *ioctl()* führt Steuerfunktionen auch für Sockets aus.
s bezeichnet den Socket-Deskriptor. Der Datentyp des als aktueller Parameter für *arg* übergebenen Objekts hängt von der jeweiligen Steuerfunktion ab, ist jedoch entweder ein Zeiger auf eine Integervariable (*int*) oder auf eine spezielle Datenstruktur. Beim Aufruf von *ioctl()* ist deshalb eine Typanpassung an „Zeiger auf char“ notwendig.

Folgende Steuerfunktionen werden für Sockets unterstützt:

request	***arg**	**Funktion**
FIONBIO	Int	Blocking-Modus ein- und ausschalten
FIONREAD	Int	Nachrichtenlänge ermitteln
FIOSETOWN	Int	Prozess-ID setzen
FIOGETOWN	Int	Prozess-ID ermitteln
SIOCSPGRP	wie FIOSETOWN	
SIOCGPGRP	wie FIOGETOWN	
SIOCGLIFNUM	struct lifnum	Interface-Anzahl ermitteln
SIOCGLIFCONF	struct lifconf	Interface-Konfiguration ermitteln
SIOCGLIFADDR	struct lifreq	Internet-Adresse des Interface ermitteln
SIOCGLIFINDEX	struct lifreq	Index des Interface ermitteln
SIOCGLIFBRDADDR	struct lifreq	Broadcast-Adresse des Interface ermitteln
SIOCGLIFNETMASK	struct lifreq	Subnetzmaske des Interface ermitteln
SIOCGLIFFLAGS	struct lifreq	Flags des Interface ermitteln
SIOCGIFNUM	Int	Interface-Anzahl ermitteln (nur IPv4)
SIOCGIFCONF	struct ifconf	Interface-Konfiguration ermitteln (nur IPv4)
SIOCGIFADDR	struct ifreq	Internet-Adresse des Interface ermitteln (nur IPv4)
SIOCGIFINDEX	struct ifreq	Index des Interface ermitteln (nur IPv4)

request	*arg	Funktion
SIOCGIFBRDADDR	struct ifreq	Broadcast-Adresse des Interface ermitteln (nur IPv4)
SIOCGIFNETMASK	struct ifreq	Subnetzmaske des Interface ermitteln (nur IPv4)
SIOCGIFFLAGS	struct ifreq	Flags des Interface ermitteln (nur IPv4)

FIONBIO
Diese Option beeinflusst das Ausführungsverhalten von Socket-Funktionen bei Auftreten der Datenflusskontrolle.

- **arg* == 0:
  Socket-Funktionen blockieren, bis die Funktion ausgeführt werden kann.
- **arg* != 0:
  Socket-Funktionen kehren mit dem *errno*-Code EWOULDBLOCK zurück, wenn die Funktion auf Grund der Datenflusskontrolle nicht sofort ausgeführt werden kann.

FIONREAD
Die Länge der aktuell im Eingangspuffer vorhandenen Nachricht wird zurückgeliefert.

FIOSETOWN
Für den spezifizierten Socket kann die Prozess-ID gesetzt werden, wodurch bei Eintreffen einer Nachricht des Prozesses ein SIGIO-Signal zugestellt wird.

SIOCSPGRP
wie FIOSETOWN

FIOGETOWN
Die für den Socket gesetzte Prozess-ID wird zurückgeliefert.

SIOCGPGRP
wie FIOGETOWN

SIOCGLIFNUM
Die Anzahl der Interfaces wird im Element *lifn_count* zurückgeliefert. Es werden nur die Interfaces entsprechend der im Element *lifn_family* angegebenen Adressfamilie (AF_UNSPEC, AF_INET oder AF_INET6) gezählt.

SIOCGLIFCONF
Es wird eine Liste der Netzwerkkonfiguration geliefert. Für jedes Interface, das der im Element *lifc_family* angegebenen Adressfamilie angehört und bei dem die im Element *lifc_flags* angegebenen Flags gesetzt sind, wird ein Eintrag vom Typ *struct lifreq* in den Bereich geschrieben, der durch das Element *lifc_buf* adressiert wird. Wenn die Länge dieses Bereichs (*lifc_len*) nicht ausreichend groß ist, wird der Fehler EINVAL gemeldet. Die Strukturen *lifconf* und *lifreq* sind in der Include-Datei <net/if.h> definiert.

SIOCGLIFADDR
Für das mit dem Element *lifr_name* spezifizierte Interface wird die Internet-Adresse im Element *lifr_addr* zurückgeliefert.

SIOCGLIFINDEX
Für das mit dem Element *lifr_name* spezifizierte Interface wird der Index (die Interface-Nummer) im Element *lifr_index* zurückgeliefert.

SIOCGLIFBRDADDR
Für das mit dem Element *lifr_name* spezifizierte Interface wird die Broadcast-Adresse im Element *lifr_broadaddr* zurückgeliefert. Für IPv4-Interfaces, die nicht Broadcast-fähig sind, und für IPv6-Interfaces wird der Fehler EADDRNOTAVAIL gemeldet.

SICGLIFNETMASK
Für das mit dem Element *lifr_name* spezifizierte Interface wird die Subnetzmaske im Element *lifr_addr* zurückgeliefert. Für IPv6-Interfaces wird der Fehler EADDRNOTAVAIL gemeldet.

SIOCGLIFFLAGS
Für das mit dem Element *lifr_name* spezifizierte Interface werden die Interface-Flags im Element *lifr_flags* zurückgeliefert. Mögliche Flags sind IFF_UP, IFF_LOOPBACK und IFF_BROADCAST.

Die folgenden Optionen werden aus Kompatibilitätsgründen unterstützt. Sie liefern aber nur Informationen über IPv4-Interfaces:

SIOCGIFNUM
Die Anzahl der IPv4-Interfaces wird im Argument zurückgeliefert.

SIOCGIFCONF
Es wird eine Liste der IPv4-Netzwerkkonfiguration geliefert. Für jedes IPv4-Interface wird ein Eintrag vom Typ *struct ifreq* in den Bereich geschrieben, der durch das Element *ifc_buf* adressiert wird. Wenn die Länge dieses Bereichs (*ifc_len*) nicht ausreichend groß ist, wird der Fehler EINVAL gemeldet. Die Strukturen *ifconf* und *ifreq* sind in der Include-Datei <net/if.h> definiert.

SIOCGIFADDR
Für das mit dem Element *ifr_name* spezifizierte Interface wird die Internet-Adresse im Element *ifr_addr* zurückgeliefert.

SIOCGIFINDEX
Für das mit dem Element *ifr_name* spezifizierte Interface wird der Index (die Interface-Nummer) im Element *ifr_index* zurückgeliefert.

SIOCGIFBRDADDR
Für das mit dem Element *ifr_name* spezifizierte Interface wird die Broadcast-Adresse im Element *ifr_broadaddr* zurückgeliefert. Falls das Interface nicht Broadcast-fähig ist, wird der Fehler EADDRNOTAVAIL gemeldet.

SIOCGLIFNETMASK
: Für das mit dem Element ifr_name spezifizierte Interface wird die Subnetzmaske im Element ifr_addr zurückgeliefert. Für IPv6-Interfaces wird der Fehler EADDRNOTAVAIL gemeldet.

SIOCGIFFLAGS
: Für das mit dem Element *ifr_name* spezifizierte Interface werden die Interface-Flags im Element *ifr_flags* zurückgeliefert. Mögliche Flags sind IFF_UP, IFF_LOOPBACK und IFF_BROADCAST.

Returnwert

0:
: bei Erfolg

-1:
: bei Fehler. *errno* wird gesetzt, um den Fehler anzuzeigen.

Fehler

EFAULT
: *request* fordert eine Datenübertragung auf den bzw. von dem Puffer, auf den *arg* zeigt. Dabei liegt jedoch ein Teil des Puffers außerhalb des dem Prozess zugewiesenen Adressraums.

EINVAL
: *request* oder *arg* sind für nicht gültig.
Der angegebene Interface-Name (in *lifr_name* bzw. *ifr_name*) ist nicht gültig.
Die angegebene Adressfamilie (in *lifn_family* bzw. *lifc_family*) ist nicht gültig.
Die Länge des bei SIOCGLIFCONF bzw. SIOCGIFCONF angegebenen Ausgabebereichs (*lifc_len* bzw. *ifc_len*) ist zu klein.

EIO
: Es ist ein physikalischer Ein-/Ausgabefehler aufgetreten.

EOPNOTSUPP
: *request* wird nicht unterstützt.

EADDRNOTAVAIL
: *request* ist für dieses Interface nicht möglich.

Beispiel

Ermitteln aller Interface-Namen und -Adressen mit 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;
}
```

Die Datenstrukturen *lifconf*, *lifreq* und *lifnum* sind in *net/if.h* wie folgt definiert:

```
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() - Ein-/Ausgabe multiplexen

```
#include <poll.h>

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

Beschreibung

Die Funktion *poll()* bietet dem Benutzer einen Mechanismus für das Multiplexen der Ein- und Ausgabe über eine Menge von Dateideskriptoren, die auf geöffnete Dateien verweisen. *poll()* kennzeichnet die Deskriptoren,

– auf denen der Benutzer Nachrichten empfangen kann oder
– auf denen der Benutzer Nachrichten senden kann oder
– auf denen bestimmte Ereignisse aufgetreten sind.

Der Parameter *fds* gibt die zu prüfenden Deskriptoren an sowie die Ereignisse, die für jeden Deskriptor von Interesse sind. *fds* ist ein Zeiger auf einen Vektor mit jeweils einem Element pro offenem Dateideskriptor.

Die Struktur *pollfd* ist wie folgt deklariert:

```
struct pollfd {
    int fd;             /* Dateikennzahl */
    short events;       /* angeforderte Ereignisse */
    short revents;      /* gemeldete Ereignisse */
};
```

Der Dateideskriptor *fd* bezeichnet einen Socket. Die Elemente *events* – für den Socket abzufragende Ereignisse – und *revents* – für den Socket zurückgemeldete Ereignisse – sind Bitmasken, die durch ODER-Verknüpfungen beliebiger Kombinationen der nachfolgend beschriebenen Ereignisanzeiger aufgebaut werden.

POLLIN
: Daten können nicht-blockierend gelesen werden, wenn eine Verbindungsanforderung empfangen wurde.

POLLOUT
: Daten können nicht-blockierend geschrieben werden.

POLLRDNORM
: wie POLLIN

POLLWRNORM
wie POLLOUT

POLLERR
Für den Socket wurde ein Fehler gemeldet. Die Option ist nur in der *revents*-Bitmaske gültig; sie wird nicht in einem *events*-Feld verwendet.

POLLHUP
Ein Hangup ist in der Verbindung aufgetreten. POLLHUP und POLLOUT schließen sich gegenseitig aus; auf einen Socket kann niemals geschrieben werden, wenn ein Hangup aufgetreten ist. Jedoch schließen sich POLLHUP und POLLIN nicht gegenseitig aus. Diese Option ist nur in der *revents*-Bitmaske gültig; sie wird nicht in einem *events*-Feld verwendet.

POLLNVAL
Der angegebene *fds->fd*-Wert gehört nicht zu einer offenen Datei. Diese Option ist nur in *fds->revents* gültig; sie wird nicht in *fds->events* verwendet.

Bei jedem Element des Vektors, auf das *fds* zeigt, prüft *poll()* den angegebenen Dateideskriptor auf das oder die in *fds->events* angegebenen Ereignisse. Die Anzahl der zu prüfenden Dateideskriptoren wird von *nfds* angegeben.

Wenn *fds->fd* kleiner als null ist, wird *fds->events* ignoriert, und *fds->revents* wird bei der Rückkehr von *poll()* in diesem Eintrag auf null gesetzt.

Die Ergebnisse der *poll()*-Anfrage werden in *fds->revents* gespeichert. Zur Anzeige, welche der angeforderten Ereignisse wahr sind, werden Bits in der Bitmaske *fds->revents* gesetzt. Wenn keine Ereignisse wahr sind, wird keines der angegebenen Bits bei der Rückkehr des *poll()*-Aufrufes in *fds->revents* gesetzt. Die Ereignisanzeiger POLLHUP, POLLERR und POLLNVAL werden stets in *fds->revents* gesetzt, wenn die von ihnen angezeigten Bedingungen wahr sind. Dies ist auch der Fall, wenn diese Optionen nicht in *fds ->events* vorhanden waren.

Wenn keines der definierten Ereignisse bei einem der jeweils ausgewählten Dateideskriptoren auftritt, wartet *poll()* wenigstens *timeout* Millisekunden auf das Auftreten eines Ereignisses bei einem der gewählten Dateideskriptoren. Bei einem Rechner, bei dem die Genauigkeit auf Millisekunden nicht zur Verfügung steht, wird *timeout* auf den nächsten zulässigen Wert aufgerundet, der in diesem System zur Verfügung steht.

Wenn der Wert von *timeout* 0 ist, kehrt *poll()* sofort zurück. Wenn der Wert von *timeout* INFTIM (oder -1) ist, bewirkt *poll()* eine Blockierung, bis ein angefordertes Ereignis auftritt, oder bis der Aufruf unterbrochen wird. *poll()* ist von den Schaltern O_NDELAY und O_NONBLOCK nicht betroffen.

Returnwert

0:
zeigt an, dass die Zeit für den Aufruf abgelaufen ist und keine Dateideskriptoren gewählt wurden.

>0:
Ein positiver Wert zeigt die Gesamtanzahl der jeweils ausgewählten Dateideskriptoren an (d.h. Dateideskriptoren, für die *fds->revents* ungleich null ist).

-1:
bei Fehler. *errno* wird gesetzt, um Fehler anzuzeigen.

Fehler

EAGAIN
Die Zuweisung der internen Datenstrukturen war erfolglos, die Anforderung sollte jedoch erneut versucht werden.

EFAULT
Ein Parameter verweist auf eine Adresse außerhalb des zugewiesenen Adressraums.

EINTR
Ein Signal wurde während des *poll()*-Aufrufs abgefangen.

EINVAL
Der Parameter *nfds* ist kleiner als null oder größer als OPEN_MAX.

Siehe auch

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

read(), readv() - Nachricht von einem Socket empfangen

```
#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);
```

Beschreibung

Die Funktionen *read()* und *readv()* empfangen Nachrichten von einem Socket. *read()* und *readv()* können nur bei einem Socket verwendet werden, über den eine Verbindung aufgebaut ist. Es wird die Länge der Nachricht zurückgeliefert.

Der Parameter *s* bezeichnet den Socket, von dem die Nachricht empfangen wird.

Bei *read()* zeigt der Parameter *buf* auf das erste Byte des Empfangspuffers. Der Parameter *len* gibt die Länge des Empfangspuffers (in Bytes) und damit die maximale Nachrichtenlänge an.

Bei *readv()* werden die empfangenen Daten im Vektor mit den Elementen *iov[0],... ,iov[iovcnt-1]* abgelegt. Die Vektorelemente sind Objekte vom Typ *struct iovec*. Im Parameter *iov* wird die Adresse des Vektors übergeben. Jedes Vektorelement enthält Adresse und Länge eines Speicherbereichs, in den *readv()* die vom Socket *s* empfangenen Daten einliest. *readv()* füllt einen Bereich nach dem anderen mit Daten, wobei *readv()* immer erst dann zum nächsten Bereich übergeht, wenn der aktuelle Bereich vollständig mit Daten gefüllt ist.

Die Struktur *struct iovec* ist wie folgt deklariert:

```
struct iovec {
caddr_t iov_base;  /* Puffer für Daten */
size_t iov_len;    /* Pufferlänge */
};
```

iovcnt gibt die Anzahl der Vektorelemente an.
Wenn auf dem Socket keine Nachrichten vorhanden sind, wartet der Empfangsaufruf auf eine ankommende Nachricht, es sei denn, der Socket ist nicht-blockierend. Vergleichen Sie dazu den Abschnitt „ioctl() - Sockets steuern“ auf Seite 140. In diesem Fall liefern *read()* bzw. *readv()* den Wert -1 zurück, wobei die Variable *errno* auf den Wert EWOULDBLOCK gesetzt wird.

Mit den Funktionen *poll()* bzw. *select()* stellen Sie fest, wann weitere Daten ankommen.

Falls der Prozess, der *read()* bzw. *readv()* aufruft, ein Signal erhält, bevor irgendwelche Daten vorhanden sind, wird der Aufruf wiederholt. Nicht wiederholt wird der Aufruf, wenn der aufrufende Prozess ausdrücklich mit *sigaction()* das Signal gibt, diese Aufrufe zu unterbrechen (siehe auch „C-Bibliotheksfunktionen (BS2000/OSD) für POSIX-Anwendungen“).

Returnwert

>0:
bei Erfolg

-1:
bei Fehler. *errno* wird gesetzt, um den Fehler anzuzeigen.

Fehler

EBADF
Der Parameter *s* ist kein gültiger Deskriptor.

ECONNRESET
Die Verbindung zum Partner wurde abgebrochen (nur bei Sockets vom Typ SOCK_STREAM).

EFAULT
Die Daten sollen in einem nicht vorhandenen oder geschützten Teil des Prozess-Adressbereichs empfangen werden.

EINTR
Der aufrufende Prozess hat ein Signal empfangen, bevor irgendwelche Daten empfangen wurden, und das Signal zum Unterbrechen des Funktionsaufrufs ist gesetzt.

EIO
Es sind Benutzerdaten verloren gegangen.

ENETDOWN
Die Verbindung zum Netzwerk ist nicht mehr aktiv.

ENOTCONN
Für den Socket besteht keine Verbindung.

ENOTSOCK
Der Deskriptor *s* referenziert eine Datei, keinen Socket.

EWOULDBLOCK
Der Socket ist als nicht-blockierend markiert, und die geforderte Operation würde blockieren.

Siehe auch

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

select() - Ein-/Ausgabe multiplexen

```
#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;
```

Beschreibung

Die Funktion *select()* überprüft drei verschiedene Mengen von Socket-Deskriptoren, die mit den Parametern *readfds*, *writefds* und *exceptfds* übergeben werden.
select() stellt dabei fest,

- welche Deskriptoren der mit *readfds* übergebenen Menge bereit zum Lesen sind,
- welche Deskriptoren der mit *writefds* übergebenen Menge bereit zum Schreiben sind,
- für welche Deskriptoren der mit *exceptfds* übergebenen Menge eine noch nicht ausgewertete Ausnahmebedingung vorliegt.

Die Deskriptormengen werden als Bitfelder in Integer-Reihungen abgespeichert. Die Größe der Bitfelder und somit auch die Größe der Deskriptormengen ist durch die Konstante FD_SETSIZE festgelegt. FD_SETSIZE ist in <sys/select.h> mit einem Wert definiert, der im Standardfall mindestens so groß ist wie die maximale Anzahl der vom System unterstützten Deskriptoren.

Der Parameter *width* gibt an, wieviele Bits in jeder Bitmaske zu prüfen sind. *select()* prüft in den einzelnen Bitmasken die Bits 0 bis *width*-1. Im Standardfall hat *width* den Wert, der von der Funktion *ulimit()* als die maximale Anzahl von Socket-Deskriptoren geliefert wird. Die Funktion *ulimit()* ist beschrieben im Handbuch „C-Bibliotheksfunktionen (BS2000/OSD) für POSIX-Anwendungen“. *select()* ersetzt die beim Aufruf übergebenen Deskriptormengen durch entsprechende Untermengen. Diese Untermengen enthalten jeweils alle Deskriptoren, die für die betreffende Operation bereit sind.

Mit den folgenden Makros können Sie Bitmasken bzw. Deskriptormengen manipulieren:

FD_ZERO(&fdset)
initialisiert die Deskriptormenge *fdset* als leere Menge.

FD_SET(fd, &fdset)
erweitert die Deskriptormenge *fdset* um den Deskriptor *fd*.

FD_CLR(fd, &fdset)
entfernt den Deskriptor *fd* aus der Deskriptormenge *fdset*.

FD_ISSET(fd, &fdset)
prüft, ob der Deskriptor *fd* ein Element der Deskriptormenge *fdset* ist:

- Rückgabewert != 0: *fd* ist Element von *fdset*.
- Rückgabewert == 0: *fd* ist nicht Element von *fdset*.

Das Verhalten dieser Makros ist undefiniert, wenn der Deskriptor-Wert <0 oder ≥ FD_SETSIZE ist.

Der Parameter *timeout* legt die maximale Zeitspanne fest, die der Funktion *select()* für die vollständige Auswahl der bereiten Deskriptoren zur Verfügung steht. Wenn *timeout* der Null-Zeiger ist, blockiert *select()* auf unbestimmte Zeit.
Ein zyklisches Auswählen (Polling) können Sie veranlassen, wenn Sie für *timeout* einen Zeiger auf ein *timeval*-Objekt übergeben, dessen Komponenten sämtlich den Wert 0 haben.

Wenn die Deskriptoren nicht von Interesse sind, kann als aktueller Parameter für *readfds*, *writefds* und *exceptfds* der Null-Zeiger übergeben werden.

Stellt *select()* nach einem Aufruf von *listen()* die Lesebereitschaft eines Socket-Deskriptors fest, so zeigt dies an, dass ein folgender *accept()*-Aufruf für diesen Deskriptor nicht blockieren wird.

Returnwert

>0:
Der positive Wert gibt die Anzahl der bereiten Deskriptoren in den Deskriptormengen an.

0:
gibt an, dass das Timeout-Limit überschritten wurde.

-1:
bei Fehler. *errno* wird gesetzt, um den Fehler anzuzeigen. Die Deskriptormengen werden dann nicht verändert.

Fehler

EBADF
Eine der Deskriptormengen spezifiziert einen ungültigen Deskriptor.

EFAULT
Einer der Zeiger, die übergeben wurden, zeigt auf einen nicht vorhandenen Bereich im Prozess-Adressbereich.

EINTR
Es wurde ein Signal empfangen, bevor eines der ausgewählten Ereignisse eintraf oder bevor die Zeitbegrenzung abgelaufen ist.

EINVAL
Eine Komponente der angegebenen Zeitbegrenzung liegt außerhalb des gültigen Bereichs.
Der gültige Bereich ist wie folgt festgelegt:

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

Hinweis

In seltenen Fällen kann *select()* anzeigen, dass ein Deskriptor bereit zum Schreiben ist, während ein Schreibversuch tatsächlich aber blockieren würde. Das kann vorkommen, wenn für das Schreiben notwendige System-Ressourcen erschöpft oder nicht vorhanden sind. Wenn es für Ihre Anwendung kritisch ist, dass Schreiboperationen auf einen Dateideskriptor nicht blockieren, sollten Sie den Deskriptor mit einem *fcntl()*-Aufruf auf nichtblockierende Ein-/Ausgabe setzen.

Siehe auch

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

write(), writev() - Nachricht von Socket zu Socket senden

```
#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);
```

Beschreibung

Die Funktionen *write()* bzw. *writev()* senden Nachrichten von einem Socket zu einem anderen. *write()* bzw. *writev()* können nur bei einem Socket benutzt werden, über den eine Verbindung aufgebaut ist.

Der Parameter *s* bezeichnet den Socket, über den die Nachricht gesendet wird.

Bei *write()* zeigt der Parameter *buf* auf das erste Byte des Sendepuffers und *len* spezifiziert die Länge (in Bytes) der Nachricht im Sendepuffer.

Bei *writev()* werden die zu sendenden Daten im Vektor mit den Elementen *iov[0], ... ,iov[iovcnt-1]* bereitgestellt. Die Vektorelemente sind Objekte vom Typ *struct iovec*. Im Parameter *iov* wird die Adresse des Vektors übergeben. Jedes Vektorelement enthält Adresse und Länge eines Speicherbereichs, aus dem *writev()* die zu sendenden Daten liest.

Die Struktur *struct iovec* ist wie folgt deklariert:

```
struct iovec {
    caddr_t iov_base;      /* Puffer für Daten */
    size_t iov_len;        /* Pufferlänge */
};
```

iovcnt gibt die Anzahl der Vektorelemente an.

Ist die Nachricht zu lang, um von der darunterliegenden Protokollebene ganz transportiert zu werden, dann wird der Fehler EMSGSIZE geliefert und die Nachricht wird nicht übermittelt.

Wenn der Prozess, der *write()* bzw. *writev()* aufruft, ein Signal empfängt, bevor irgendwelche Daten zum Senden gepuffert werden, wird der Aufruf wiederholt.
Nicht wiederholt wird der Aufruf, wenn der aufrufende Prozess ausdrücklich mit *sigaction()* das Signal setzt, diesen Aufruf zu unterbrechen (siehe auch Beschreibung von SA_RESTART bei *sigaction()* im Handbuch „C-Bibliotheksfunktionen (BS2000/OSD) für POSIX-Anwendungen“).

Returnwert

0:
bei Erfolg (Anzahl der tatsächlich gesendeten Bytes).

-1:
bei Fehler. *errno* wird gesetzt, um den Fehler anzuzeigen. Die Deskriptormengen werden dann nicht verändert.

Fehler

EBADF
Der Parameter *s* ist kein gültiger Deskriptor.

ECONNRESET
Die Verbindung zum Partner wurde abgebrochen (nur bei Sockets vom Typ SOCK_STREAM).

EFAULT
Die Daten sollen in einen nicht vorhandenen oder geschützten Teil des Prozess-Adressbereichs gesendet werden.

EINTR
Der aufrufende Prozess hat ein Signal empfangen, bevor irgendwelche Daten zum Senden gepuffert werden konnten, und das Signal zum Unterbrechen des Funktionsaufrufs ist gesetzt.

EINVAL
Ein Parameter spezifiziert einen ungültigen Wert.

EMSGSIZE
Die Nachricht ist zu groß, um auf einmal gesendet zu werden.

ENETDOWN
Die Verbindung zum Netzwerk ist nicht mehr aktiv.

ENOBUFS
Das System konnte keinen internen Puffer bereitstellen. Die Operation kann gelingen, wenn Speicherplatz frei wird.
Die Ausgabe-Warteschlange für ein Netz-Interface ist voll. Dies führt generell dazu, dass das Interface aufhört zu senden, kann aber auf einem vorübergehenden Stau beruhen.

ENOTCONN
Für den Socket besteht keine Verbindung.

ENOTSOCK
Der Deskriptor *s* referenziert eine Datei, keinen Socket.

EPIPE
: Der Socket ist nicht für Schreiben aktiviert, oder der Socket ist verbindungsorientiert und der Partner hat die Verbindung beendet.
Wenn der Socket vom Typ SOCK_STREAM ist, wird das Signal SIGPIPE für den aufrufenden Prozess generiert.

EWOULDBLOCK
: Der Socket ist als nicht-blockierend markiert, und die geforderte Operation würde blockieren.

Siehe auch

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

7 Grundlagen von XTI(POSIX)

X/Open Transport Interface (XTI) ist der von X/Open definierte Standard für eine Reihe von Programmier-Schnittstellen, die, ebenso wie die Socket-Schnittstelle, einer Anwendung den Zugang zur Netzwerkebene ermöglichen.

XTI bietet zwei verschiedene Arten von Diensten an:

- verbindungsorientierter Dienst
- verbindungsloser Dienst

Dem Anwender stellt sich XTI als endliche, ereignisgesteuerte Zustandsmaschine dar. Dies bedeutet:

- Für einen Transportendpunkt gibt es eine endliche Menge von definierten Zuständen.
- Jeder dieser Zustände kann nur durch bestimmte Ereignisse erreicht werden.
- In jedem Zustand können nur bestimmte Funktionen ausgeführt werden.

7.1 Verbindungsorientierter Dienst

Der verbindungsorientierte Dienst transportiert Daten über eine einmal geschaffene „virtuelle“ Verbindung. Dieser Dienst ist zugeschnitten auf Anwendungen, die eine sichere, datenflussorientierte Verbindung benötigen.

7.1.1 Phasen des verbindungsorientierten Dienstes

Der verbindungsorientierte Dienst besteht aus vier Phasen:

– lokale Verwaltung
– Verbindungsaufbau
– Datenübertragung
– Verbindungsabbau

Lokale Verwaltung

Die lokale Verwaltung definiert Funktionen zwischen dem Transportbenutzer, dem Transportanbieter und anderen Instanzen, die den Verbindungsaufbau steuern.

Beispiele für lokale Funktionen:

– Der Benutzer muss einen Kommunikationskanal zum Transportanbieter aufbauen. Jeder Kanal zwischen dem Benutzer und dem Transportanbieter wird *Transportendpunkt* genannt. Mit der Funktion *t_open()* wählt der Benutzer einen speziellen Transportanbieter aus und richtet einen Transportendpunkt ein.

– Jeder Benutzer kann einen oder mehrere Transportendpunkte verwalten, die er gegenüber dem Transportanbieter identifizieren muss. Zu diesem Zweck ordnet der Benutzer jedem Transportendpunkt mit der Funktion *t_bind()* eine netzweit eindeutige Transportadresse zu, d.h. er *bindet* eine Transportadresse an den Transportendpunkt. Die Struktur der Transportadresse wird durch den entsprechenden Transportanbieter bestimmt.

Zusätzlich zu *t_open()* und *t_bind()* gibt es weitere Funktionen, die die lokale Verwaltung der Transportschnittstelle unterstützen. In Tabelle 2 sind sie zusammengefasst.

Funktion	Beschreibung
t_alloc()	reserviert Speicher für die Transportschnittstelle.
t_bind()	bindet eine Adresse an einen Transportendpunkt.
t_close()	schließt einen Transportendpunkt.
t_error()	druckt eine Fehlermeldung des Transportanbieters aus.
t_free()	gibt den mit *t_alloc()* reservierten Speicherbereich frei.
t_getinfo()	liefert den Parametersatz des aktuellen Transportanbieters.
t_getstate()	liefert den Zustand des Transportendpunkts.
t_look()	liefert die momentanen Ereignisse des Transportendpunkts.
t_open()	richtet einen Transportendpunkt ein, der an einen bestimmten Transportanbieter gebunden wird.
t_optmgmt()	verhandelt mit dem Transportanbieter über protokollspezifische Optionen.
t_sync()	synchronisiert den Transportendpunkt mit dem Transportanbieter.
t_unbind()	löst eine Adresse von einem Transportendpunkt.

Tabelle 2: Funktionen für die lokale Verwaltung der Transportschnittstelle

Verbindungsaufbau

Beim Verbindungsaufbau wird zwischen zwei Benutzern eine Kommunikationsverbindung aufgebaut.
Der Verbindungsaufbau kann am Beispiel zweier Transportbenutzer veranschaulicht werden, die in einem Client-/Server-Verhältnis zueinander stehen: Ein Transportbenutzer (Server) stellt einer Gruppe von Benutzern (Clients) eine Reihe von Diensten zur Verfügung und wartet dann auf Anforderungen der Clients. Jeder Client kann einen Dienst anfordern, nachdem er zum Server eine Verbindung aufgebaut hat.

Der Client fordert mit der Funktion *t_connect()* eine Verbindung an. Ein Parameter von *t_connect()*, die Adresse, identifiziert den Server, den der Client erreichen will. Der Server muss die Funktion *t_listen()* verwenden, um sich von jeder ankommenden Verbindungsanforderung unterrichten zu lassen. Mit *t_accept()* nimmt der Server eine Anforderung zum Verbindungsaufbau an. Danach ist die Transportverbindung aufgebaut.

Tabelle 3 zeigt die Funktionen für einen Verbindungsaufbau.

Funktion	Beschreibung
t_accept()	nimmt einen Auftrag zum Verbindungsaufbau an.
t_connect()	fordert eine Verbindung mit einem bestimmten Benutzer bei einer spezifizierten Adresse an.
t_listen()	wartet auf eine Anforderung zum Verbindungsaufbau durch einen anderen Benutzer.
t_rcvconnect()	bestätigt einen Auftrag zum Verbindungsaufbau, wenn *t_connect()* im asynchronen Betrieb aufgerufen wurde.

Tabelle 3: Funktionen für den Verbindungsaufbau

Datenübertragung

Die Datenübertragung ermöglicht es zwei Benutzern, Daten über eine bestehende Verbindung in beiden Richtungen auszutauschen. Die Funktionen *t_snd()* und *t_rcv()* senden bzw. empfangen Daten über diese Verbindung. Es ist gewährleistet, dass die gesendeten Daten beim Empfänger in derselben Reihenfolge ankommen, in der sie abgeschickt wurden.

Tabelle 4 zeigt die Funktionen für die verbindungsorientierte Datenübertragung.

Funktion	Beschreibung
t_rcv()	empfängt Daten.
t_snd()	sendet Daten.

Tabelle 4: Funktionen für die verbindungsorientierte Datenübertragung

Verbindungsabbau

Der Benutzer schickt dem Transportanbieter einen Auftrag, eine bestehende Verbindung abzubauen.
Es gibt zwei unterschiedliche Arten von Verbindungsabbau:

- Verbindungsabbruch:
  Der Verbindungsabbruch weist den Transportanbieter an, die Verbindung sofort zu beenden. Dabei können alle zuvor gesendeten Daten verloren gehen, die noch nicht den Empfänger erreicht haben. Mit der Funktion *t_snddis()* kann ein Transportbenutzer einen solchen Verbindungsabbruch herbeiführen. Der Kommunikationspartner, der von diesem Verbindungsabbruch betroffen ist, kann mit der Funktion *t_rcvdis()* die Ursache des Verbindungsabbruchs abfragen. Die Funktion *t_rcvdis()* behandelt ankommende Wünsche nach Abbruch einer Verbindung.

- geordneter Verbindungsabbau:
  Neben dem Verbindungsabbruch ermöglichen einige Transportanbieter einen geordneten Verbindungsabbau, bei dem niemals Daten verloren gehen. Die Funktionen *t_sndrel()* und *t_rcvrel()* realisieren einen geordneten Verbindungsabbau.

  Der geordnete Verbindungsabbau zwischen zwei Benutzern Benutzer1 und Benutzer2 läuft immer in folgenden Schritten ab:

  1. Benutzer1, der als Erster die Verbindung abbauen will, benutzt die Funktion *t_sndrel()*, um an Benutzer2 einen Wunsch nach Abbau der Verbindung zu senden. *t_sndrel()* informiert den Benutzer2, dass Benutzer1 keine weiteren Daten sendet.
  2. Wenn Benutzer2 eine solche Nachricht mit der Funktion *t_rcvrel()* empfängt, kann er weiterhin Daten an Benutzer1 senden.
  3. Wenn Benutzer2 alle Daten übertragen hat, muss er ebenfalls *t_sndrel()* aufrufen. Dies signalisiert dem Benutzer1, dass Benutzer2 nun bereit ist, die Verbindung abzubauen.
  4. Sobald Benutzer1 mit *t_rcvrel()* die Nachricht von Benutzer2 empfangen hat, wird die Verbindung abgebaut.

Tabelle 5 zeigt die Funktionen für einen Verbindungsabbau.

Funktion	**Beschreibung**
t_rcvdis()	informiert über einen Verbindungsabbruch.
t_rcvrel()	zeigt an, dass der Kommunikationspartner einen geordneten Verbindungsabbau wünscht.
t_snddis()	fordert zum Verbindungsabbruch auf oder weist eine Verbindungsanforderung zurück.
t_sndrel()	fordert zum geordneten Verbindungsabbau auf.

Tabelle 5: Funktionen für den Abbau einer Verbindung

Zusammenspiel der Funktionen des verbindungsorientierten Dienstes

In Bild 4 ist das Zusammenspiel der XTI-Funktionen veranschaulicht, die die einzelnen Phasen des verbindungsorientierten Dienstes realisieren. Ausführlich beschrieben sind die einzelnen XTI-Funktionen im Kapitel „Bibliotheksfunktionen von XTI(POSIX)“ auf Seite 235.

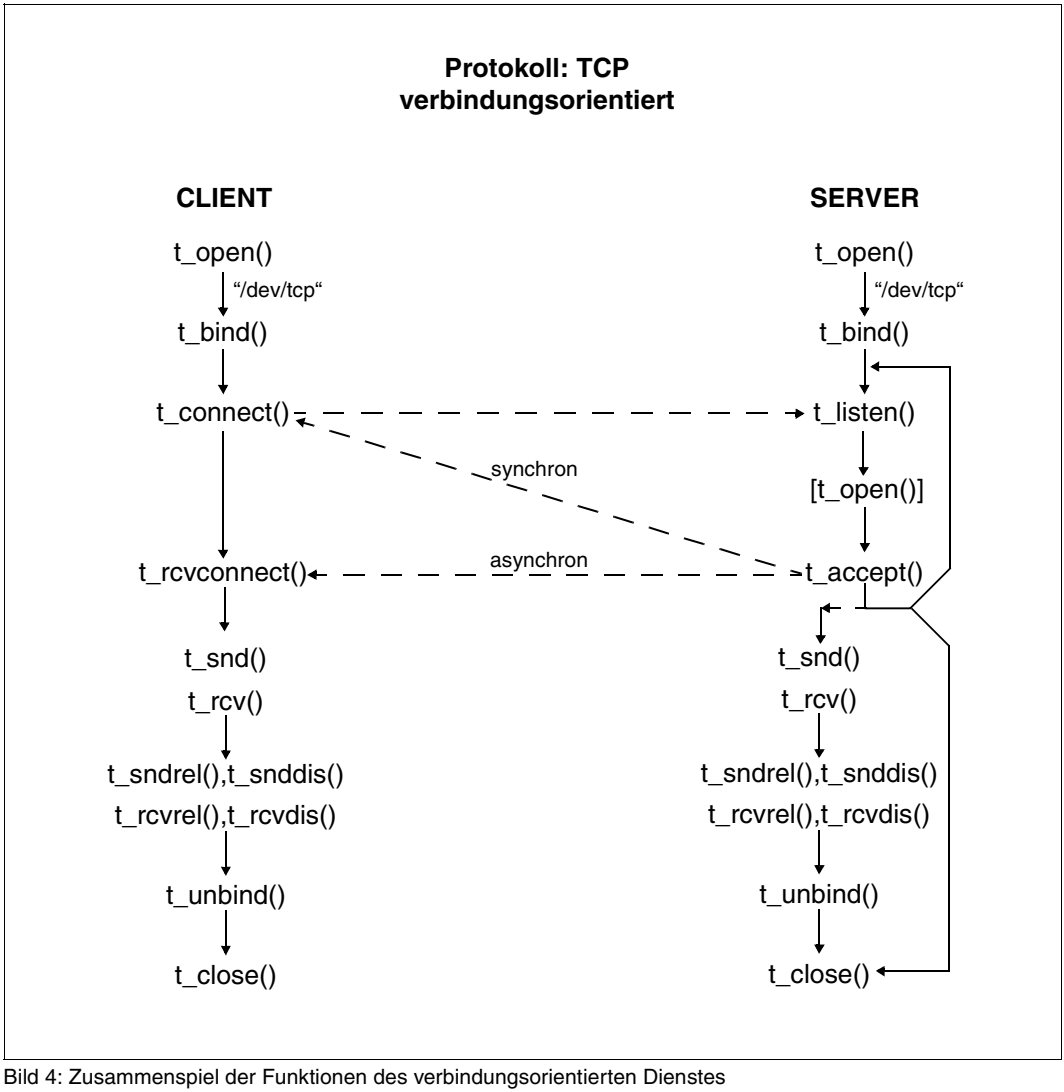

Bild 4: Zusammenspiel der Funktionen des verbindungsorientierten Dienstes

7.1.2 Verbindungsorientiertes Client-/Server-Modell

In diesem Abschnitt werden die einzelnen Phasen des verbindungsorientierten Dienstes näher erläutert anhand eines Programmbeispiels für folgende einfache Client-/Server-Anwendung:

1. Client und Server führen die lokale Verwaltung durch.
2. Zwischen Client und Server wird eine Verbindung aufgebaut.
3. Der Server überträgt eine Datei an den Client. Der Client empfängt die Datei vom Server und gibt sie auf seiner Standardausgabe aus.
4. Client und Server beenden die Verbindung.

Das Programmbeispiel wird in einzelnen Programmausschnitten vorgestellt, wobei jeweils zwei Programmausschnitte eine Phase des verbindungsorientierten Dienstes erläutern. Ein Programmausschnitt übernimmt dabei die Rolle des Clients und der andere die Rolle des Servers.

Der in den Beispielen des vorliegenden Abschnitts verwendete Programmcode ist vollständig und zusammenhängend dargestellt in Abschnitt „Client im verbindungsorientierten Dienst“ auf Seite 212 und Abschnitt „Server im verbindungsorientierten Dienst“ auf Seite 214.

Lokale Verwaltung am Beispiel des Client-/Server-Modells

Bevor Client und Server eine Kommunikationsverbindung aufbauen können, müssen sie zuerst mit *t_open()* je einen lokalen Kanal zum Transportanbieter einrichten. Danach muss jeder von beiden mit *t_bind()* eine lokale Adresse bekannt geben, unter der er über seinen zugeordneten Transportendpunkt erreichbar ist.
Die verschiedenen Dienste, die die Transportschnittstelle anbietet, erhält der Benutzer mit dem Aufruf von *t_open()*.

Die Dienste sind wie folgt aufgebaut:

Adresse	maximale Größe einer Adresse
Optionen	maximale Anzahl Bytes für protokollspezifische Optionen, die der Benutzer mit dem Transportanbieter austauschen kann
tsdu	maximale Nachrichtengröße, die im verbindungsorientierten oder verbindungslosen Dienst übertragen werden kann
etsdu	maximale Anzahl von Bytes für Vorrangdaten, die über eine Verbindung gesendet werden können
Verbindungsaufbau (connect)	maximale Anzahl von Bytes für Benutzerdaten, die beim Verbindungsaufbau ausgetauscht werden können
Verbindungsabbau (discon)	maximale Anzahl von Bytes der Benutzerdaten, die beim Verbindungsabbau übertragen werden können
Diensttyp	Typ des Dienstes, der vom Transportanbieter unterstützt wird

Drei Diensttypen sind definiert:

T_COTS	Der Transportanbieter unterstützt den verbindungsorientierten Dienst, erlaubt aber keinen geordneten Verbindungsabbau. Die Verbindung kann nur abgebrochen werden.
T_COTS_ORD	Der Transportanbieter unterstützt den verbindungsorientierten Dienst und bietet die Möglichkeit eines geordneten Verbindungsabbaus (Standardfall bei XTI(POSIX) im verbindungsorientierten Dienst).
T_CLTS	Der Transportanbieter unterstützt den verbindungslosen Dienst.

Mit *t_open()* erhält der Benutzer die voreingestellten Leistungsmerkmale des Transportendpunkts. Wenn es sich um dynamische Leistungsmerkmale handelt, können sich diese Merkmale nachträglich noch ändern. Mit *t_getinfo()* kann sich der Benutzer über die aktuellen Leistungsmerkmale des Transportendpunkts informieren.

Wenn ein Benutzer einen Transportendpunkt eingerichtet hat, muss er dem Transportanbieter die Adresse übergeben, unter der er über diesen Transportendpunkt zu erreichen ist. Wie bereits beschrieben, übergibt der Benutzer mit *t_bind()* dem Transportanbieter die Adresse des Transportendpunkts. Bei Server-Stationen sorgt *t_bind()* außerdem dafür, dass ankommende Verbindungsanforderungen vom Transportanbieter bearbeitet und dem Transportendpunkt übergeben werden können.

Während der Einrichtung des Transportendpunkts ist noch eine weitere Funktion verfügbar: Mit *t_optmgmt()* kann der Benutzer Leistungsmerkmale verändern. Von jedem Transportprotokoll wird erwartet, dass es seine eigene Menge von veränderbaren Leistungsmerkmalen bereitstellt. Dies können zum Beispiel Parameter sein, die die Dienstqualität beeinflussen. Auf Grund der protokollspezifischen Natur dieser Parameter werden nur Anwendungen für eine spezielle Protokollumgebung diese Möglichkeit nutzen.

Die lokalen Verwaltungsaufgaben werden nachfolgend jeweils am Beispiel eines Clients und eines Servers gezeigt. Die beiden Beispiele enthalten die Definitionen und die Aufrufe.

Lokale Verwaltung durch den 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() gescheitert");
      exit(1);
   }
   if (t_bind(fd, NULL, NULL) < 0)
   {
      t_error("t_bind() nicht erfolgreich");
      exit(2);
   }
```

Der erste Parameter von *t_open()* ist der Pfadname des Gerätes, das den geforderten Transportdienst bereitstellt. Im vorliegenden Beispiel ist */dev/tcp* eine Gerätedatei; */dev/tcp* stellt ein verbindungsorientiertes Transportprotokoll zur Verfügung. Dieses Transportprotokoll wird durch den zweiten Parameter für Schreib-/Lesezugriffe geöffnet. Den dritten Parameter kann der Benutzer verwenden, um sich über Leistungsmerkmale zu informieren. Diese Information wird für das Erstellen protokollunabhängiger Programme benötigt. Um das Beispiel einfach zu halten, wird auf diese Information nicht zurückgegriffen.

Client und Server nehmen an, dass der Transportanbieter folgende Leistungsmerkmale besitzt:

– Der Transportanbieter unterstützt den Diensttyp T_COTS_ORD; im Beispiel wird T_COTS_ORD für den geordneten Verbindungsabbau verwendet.

– Benutzerdaten können während des Verbindungsaufbaus oder -abbaus nicht ausgetauscht werden.

– Der Transportanbieter unterstützt keine protokollspezifischen Leistungsmerkmale.

Da diese Leistungsmerkmale nicht vom Benutzer benötigt werden, wird beim Aufruf von *t_open()* als dritter Parameter NULL übergeben. Falls der Benutzer einen anderen Diensttyp als T_COTS_ORD benötigt, muss eine andere Gerätedatei geöffnet werden. Ein Beispiel für T_CLTS ist im Abschnitt „Verbindungsloser Dienst am Beispiel eines Auftragssystems“ auf Seite 184 aufgeführt.

Als Rückgabewert liefert *t_open()* einen Integer-Wert, der in allen weiteren Aufrufen des Transportanbieters zur Identifizierung des mit *t_open()* eingerichteten Transportendpunkts benötigt wird. Dieser Integerwert ist ein Dateideskriptor.

Nachdem der Transportendpunkt eingerichtet ist, ruft der Benutzer *t_bind()* auf, um dem Transportendpunkt eine Adresse zuzuweisen. Der erste Parameter von *t_bind()* kennzeichnet den Transportendpunkt, der zweite Parameter beschreibt die Adresse, die an den Transportendpunkt gebunden werden soll. Der dritte Parameter enthält bei der Rückkehr von *t_bind()* die tatsächlich gebundene Adresse.

Im Gegensatz zur Adresse eines Server-Transportendpunkts, die von allen Clients beim Zugriff auf den Server benötigt wird, muss die Adresse eines Clients nicht allgemein bekannt sein. Da kein anderer Prozess versuchen wird, auf die Adresse eines Clients zuzugreifen, kümmert sich ein Client normalerweise nicht um seine Adresse. Dies wird im obigen Beispiel gezeigt, wo beim *t_bind()*-Aufruf als zweiter und dritter Parameter NULL übergeben wird. Wenn der zweite Parameter NULL ist, ordnet der Transportanbieter eine Adresse zu. Der dritte Parameter NULL bedeutet, dass der Client sich nicht für die vom Transportanbieter zugeordnete Adresse „interessiert“.

Wenn entweder *t_open()* oder *t_bind()* nicht erfolgreich ist, wird *t_error()* aufgerufen, um eine entsprechende Fehlermeldung auf *stderr* auszugeben. Wenn irgendeine Transportanbieterfunktion nicht erfolgreich sein sollte, enthält die globale Integer-Variable *t_errno* einen entsprechenden Wert, der den Fehler genauer anzeigt. Eine Reihe solcher Fehlerwerte sowie

die Variable *t_errno* selbst sind in <xti.h> für die Transportanbieter definiert. *t_error()* gibt eine Fehlermeldung entsprechend dem Wert von *t_errno* aus. Diese Funktion arbeitet analog zur Funktion *perror()*, die eine Fehlermeldung entsprechend dem Wert von *errno* ausgibt. Wenn es sich bei dem Fehler im Transportanbieter um einen Systemfehler handelt, erhält *t_errno* den Wert TSYSERR und *errno* wird auf den entsprechenden Systemfehlerwert gesetzt.

Lokale Verwaltung durch den Server

Der Server in diesem Beispiel muss ähnlich vorgehen, bevor mit der Kommunikation begonnen werden kann. Der Server muss einen Transportendpunkt einrichten, der ständig auf Verbindungsanforderungen wartet.

Die notwendigen Definitionen und Aufrufe sehen wie folgt aus:

```
#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; /* Für den Dateideskriptor der Verbindung */

main()
{

   int listen_fd;              /*   Dateideskriptor für
                                *   Verbindungsanforderung
                                */
   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("Aufruf t_open() für listen_fd gescheitert.");
      exit(1);
   }
```

```
if ((bind = (struct t_bind *)t_alloc(listen_fd, T_BIND, T_ALL)) == NULL)
{
   t_error("t_alloc() für Struktur t_bind gescheitert.");
   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() für listen_fd gescheitert.");
   exit(3);
}
```

Analog zum Client ruft auch der Server *t_open()* auf, um eine Verbindung zum gewünschten Transportanbieter aufzubauen, d.h. der Server richtet einen Transportendpunkt (*listen_fd*) ein. Diesen Transportendpunkt *listen_fd* wird der Server später bei Aufruf der Funktion *t_listen()* verwenden, um auf Verbindungsanforderungen zu warten.

Bevor der Server mit der Funktion *t_bind()* eine Adresse an den Transportendpunkt *listen_fd* binden kann, muss der Server diese Adresse bereitstellen. Diese Adresse wird mit dem zweiten Parameter (*bind*) beim Aufruf von *t_bind()* übergeben.

Der Parameter *bind* ist ein Zeiger auf ein Objekt vom Datentyp *struct t_bind*. Alle Strukturen und Konstanten des Transportanbieters sind in <xti.h> deklariert bzw. definiert.

Die Struktur *t_bind* ist in <xti.h> wie folgt deklariert:

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

bind->qlen gibt die maximale Anzahl erlaubter Verbindungsanforderungen an.
Wenn der Wert von *bind->qlen* größer als 0 ist, können ankommende Verbindungsanforderungen mit diesem Transportendpunkt bearbeitet werden. Ankommende Verbindungsanforderungen für die in *bind->addr* bereitgestellte Adresse stellt der Server dann in eine Warteschlange. Außerdem gibt *bind->qlen* die maximale Anzahl von Anforderungen an, die der Server gleichzeitig bearbeiten kann. Der Server muss auf jede Anforderung antworten, indem er sie annimmt oder zurückweist. Eine Verbindungsanforderung heißt anstehend, wenn der Server sie noch nicht beantwortet hat.

Oft wird ein Server erst eine Verbindungsanforderung vollständig bearbeiten und dann die nächste. In diesem Fall ist 1 der richtige Wert für *qlen*. Wenn ein Server mehrere Aufträge gleichzeitig bearbeiten möchte, spezifiziert *bind->qlen* die maximale Anzahl Aufträge, die gleichzeitig bearbeitet werden können.
Da der Server im vorliegenden Beispiel eine Verbindungsanforderung nach der anderen verarbeitet, muss *bind->qlen* der Wert 1 zugewiesen werden. Das Beispiel eines Servers, der mehrere Anforderungen gleichzeitig bearbeitet, wird im Abschnitt „Gleichzeitige Verwaltung mehrerer Verbindungen und ereignisgesteuerter Betrieb“ auf Seite 203 vorgestellt.

addr hat den Datentyp *struct netbuf* und beschreibt die anzubindende Adresse.
Die Struktur *netbuf* ist in <xti.h> wie folgt deklariert:

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

buf ist ein Zeiger auf einen Datenpuffer, *len* gibt die Anzahl Bytes im Puffer an, und *maxlen* gibt die maximale Anzahl Bytes an, die in den Puffer geschrieben werden können. Letztere Angabe wird nur benötigt, wenn Daten vom Transportanbieter zum Benutzer transportiert werden.

Durch den Aufruf von *t_alloc()* wird dynamisch Speicher für ein *t_bind*-Objekt angelegt. Der erste Parameter von *t_alloc()* nennt den Dateideskriptor, der den Transportendpunkt identifiziert. Der zweite Parameter spezifiziert die anzulegende Transportanbieterstruktur, d.h. im vorliegenden Fall *t_bind*. Der dritte Parameter gibt an, welche Komponenten dieser Struktur angelegt werden sollen. T_ALL bedeutet, dass für alle Komponenten der Struktur Speicher angelegt werden soll. Im obigen Beispiel wird dadurch der *addr*-Puffer angelegt. Die Größe dieses Puffers wird vom Transportanbieter bestimmt, der eine maximale Adresslänge festlegt. Diese Länge steht in der Komponente *maxlen* der Struktur *netbuf*.
Die Verwendung von *t_alloc()* stellt die Kompatibilität mit zukünftigen Versionen des Transportanbieters sicher.

Bei Objekten des Typs *struct t_bind* werden die Daten als Adresse interpretiert. Allgemein wird angenommen, dass die Struktur einer Adresse von Protokoll zu Protokoll verschieden ist. Die Struktur *netbuf* ist so aufgebaut, dass jedes Protokoll unterstützt werden kann.

Anschließend wird die Adressinformation dem neu angelegten *t_bind*-Objekt zugewiesen. Im Beispiel wird die Adresse selbst dabei entsprechend der Adressstruktur der Internet-Kommunikationsdomäne strukturiert (siehe *struct sockaddr_in* auf Seite 14).

Die so erzeugte Adresse bindet der Server nun mit der Funktion *t_bind()* an den Transportendpunkt *listen_fd*. Nach erfolgreichem Aufruf von *t_bind()* kann der Server von jedem Client über diese Adresse angesprochen werden. Der Transportanbieter stellt ankommende Verbindungsanforderungen in eine Warteschlange und leitet damit die nächste Phase des Verbindungsaufbauprotokolls ein, den eigentlichen Verbindungsaufbau.

Verbindungsaufbau am Beispiel des Client-/Server-Modells

Der Verbindungsaufbau verdeutlicht den Unterschied zwischen Client und Server. Der Transportanbieter stellt für beide jeweils spezielle Funktionen zur Verfügung. Der Client ruft *t_connect()* auf, um eine Verbindung anzufordern, während der Server mit *t_listen()* auf Verbindungsanforderungen wartet. Der Server kann mit der Funktion *t_accept()* eine Verbindung annehmen oder sie mit *t_snddis()* ablehnen. Der Client wird über die Entscheidung des Transportanbieters informiert, wenn die Funktion *t_connect()* beendet ist.

Verbindungsanforderung durch den Client

Um mit dem Client-/Server-Beispiel fortzufahren, sind aus Sicht des Clients für einen Verbindungsaufbau folgende Schritte notwendig:

```
if ((sndcall = (struct t_call *)t_alloc(fd, T_CALL, T_ADDR)) == NULL) {
   t_error("t_alloc() gescheitert");
   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() für fd gescheitert");
   exit(4);
}
```

Bevor der Client mit *t_connect()* eine Verbindungsanforderung an den Server schicken kann, muss der Client die Adresse des Servers spezifizieren. Diese Adresse wird dann als zweiter Parameter (*sndcall*) beim Aufruf von *t_connect()* übergeben.

Der Parameter *sndcall* ist ein Zeiger auf ein Objekt vom Datentyp *struct t_call*.

Die Struktur *t_call* ist in <xti.h> wie folgt deklariert:

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

t_alloc() wird im Beispiel verwendet, um ein *t_call*-Objekt dynamisch anzulegen. Im Beispiel auf der vorigen Seite werden keine Leistungsmerkmale oder Benutzerdaten angegeben. Nur die Server-Adresse wird verwendet. Als dritter Parameter von *t_alloc()* wird T_ADDR gewählt, um für die Adressinformation einen entsprechenden Puffer anzulegen.

Nach erfolgreicher Ausführung von *t_alloc()* legt der Server die Länge der Server-Adresse sowie die Server-Adresse selbst im von *t_alloc()* reservierten Speicherbereich ab. Die Serveradresse wird dabei entsprechend der Adressstruktur der Internet-Kommunikationsdomäne strukturiert (siehe *struct sockaddr_in* auf Seite 14).

Der Aufruf *t_connect()* schickt eine Verbindungsanforderung zum Server. Der erste Parameter des Aufrufs ist der Transportendpunkt, über den die Verbindung aufgebaut werden soll. Mit dem zweiten Parameter (*sndcall*) wird die Adresse des gewünschten Servers übergeben. Der dritte Parameter ist ebenfalls ein Zeiger auf ein Objekt vom Typ *struct t_call*. Dieser Parameter von *t_connect()* wird benutzt, um Informationen über die errichtete Verbindung zu erhalten. Da diese Information hier nicht benötigt wird, wird im Beispiel als dritter Parameter NULL übergeben. Wenn *t_connect()* erfolgreich ist, wird die Verbindung aufgebaut. Falls der Server die Verbindungsanforderung zurückweist, wird *t_errno* auf den Wert TLOOK gesetzt.

Der Fehler TLOOK hat eine besondere Bedeutung für die Transportschnittstelle: TLOOK informiert den Benutzer, wenn eine Funktion der Schnittstelle durch ein unerwartetes asynchrones Ereignis am gegebenen Transportendpunkt unterbrochen wurde. TLOOK zeigt daher nicht einen Fehler in der Schnittstelle an, sondern nur, dass die aufgerufene Funktion auf Grund des anstehenden Ereignisses nicht ausgeführt wird. Welche Ereignisse der Transportschnittstelle definiert sind, ist auf Seite 191 beschrieben.

Mit der Funktion *t_look()* kann der Benutzer feststellen, welches Ereignis aufgetreten ist, wenn ein Fehler TLOOK gemeldet wird. Wenn im Beispiel auf der vorigen Seite die Verbindungsanforderung abgelehnt wird, erhält der Client eine Nachricht über den Abbruch. Das Programm wird in diesem Fall beendet.

Verbindungsannahme durch den Server

Wenn der Client mit *t_connect()* eine Verbindung anfordert, wird am Transportendpunkt des Servers ein entsprechendes Ereignis gesetzt. Im Folgenden wird gezeigt, welche Schritte für die Behandlung dieses Ereignisses erforderlich sind. Der Server nimmt für jeden Client den Auftrag an und erzeugt einen neuen Prozess, um die Verbindung zu verwalten.

```
if ((call = (struct t_call *)t_alloc(listen_fd, T_CALL, T_ADDR)) == NULL){
   t_error("t_alloc() für t_call Struktur gescheitert");
   exit(5);
}

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

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

Der Server legt mit *t_alloc()* ein Objekt vom Typ *struct t_call* an, das von *t_listen()* benötigt wird. Der dritte Parameter von *t_alloc()*, T_ADDR, bewirkt, dass der Puffer für die Adresse des Clients angelegt wird.
Der Wert von *maxlen* in einem *netbuf*-Objekt gibt die aktuelle Länge des angelegten Puffers an.

Der Server läuft in einer Endlosschleife und bearbeitet pro Schleifendurchlauf eine ankommende Verbindungsanforderung. Dabei geht der Server wie folgt vor:

1. Der Server ruft die Funktion *t_listen()* auf, um auf Verbindungsanforderungen zu warten, die auf dem Transportendpunkt *listen_fd* ankommen. Die Transportadresse des Senders einer Verbindungsanforderung wird von *t_listen()* im *t_call*-Objekt gespeichert, auf das die Zeigervariable *call* zeigt.
   Wenn keine Verbindungsanforderung ansteht, blockiert die Funktion *t_listen()* den Prozess so lange, bis eine Verbindungsanforderung eintrifft.
2. Wenn eine Verbindungsanforderung eintrifft, ruft der Server die benutzerdefinierte Funktion *accept_call()* auf, um die Verbindung zu bestätigen. *accept_call()* nimmt die Verbindungsanforderung auf einem neuen Transportendpunkt entgegen und liefert den zugehörigen Dateideskriptor als Ergebnis. Dieser Dateideskriptor wird in der globalen Variablen *conn_fd* gespeichert. Da die Verbindung auf einem neuen Transportendpunkt aufgebaut wird, kann der Server auf dem alten Transportendpunkt neue Anforderungen erwarten. Die Funktion *accept_call()* ist auf der folgenden Seite näher beschrieben.

3. Wenn die Verbindungsannahme erfolgreich war, erzeugt die Funktion *run_service()* einen neuen Prozess, um die Verbindung zu verwalten. Die benutzerdefinierte Funktion *run_service()* ist auf Seite 177 näher beschrieben.

Die Transportschnittstelle unterstützt einen asynchronen Modus. Der asynchrone Modus wird beschrieben im Kapitel „Weiterführende Konzepte von XTI(POSIX)“ auf Seite 201.

Die Funktion *accept_call()*, die der Server aufruft, um eine Verbindungsanforderung anzunehmen, ist wie folgt definiert:

```
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() Aufruf für accept gescheitert");
      exit(7);
   }
   while (t_accept(listen_fd, resfd, call) < 0) {
      if (t_errno == TLOOK) {
         if (t_look(listen_fd) == T_DISCONNECT) { /* Verbindungsabbruch */
              if (t_rcvdis(listen_fd, NULL) < 0) {
                 t_error("t_rcvdis() gescheitert für listen_fd");
                 exit(9);
              }
              if (t_close(resfd) < 0) {
                 t_error("t_close gescheitert für antwortenden fd");
                 exit(10);
              }
              /* Aufruf beenden und auf weiteren Aufruf warten */
              return(DISCONNECT);
         } else { /* neues T_LISTEN; Ereignis löschen */
              if ((refuse_call =
                 (struct t_call *)t_alloc(listen_fd,T_CALL,0)) == NULL) {
                 t_error("t_alloc() für refuse_call gescheitert");
                 exit(11);
              }

              if (t_listen(listen_fd, refuse_call) < 0) {
                 t_error("t_listen() für refuse_call gescheitert");
                 exit(12);
              }
```

```
                if (t_snddis(listen_fd, refuse_call) < 0) {
                   t_error("t_snddis() für refuse_call gescheitert");
                   exit(13);
                }

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

Der Aufruf von *accept_call()* benötigt zwei Parameter:

– *listen_fd* gibt den Transportendpunkt an, an dem die Verbindungsanforderung angekommen ist.

– *call* ist ein Zeiger auf ein Objekt vom Datentyp *struct t_call*, das alle Informationen für diese Anforderungen enthält.

Die Funktion *t_call()* erzeugt zuerst einen weiteren Transportendpunkt. Der neu erzeugte Transportendpunkt *resfd* wird benutzt, um die Verbindungsanforderung anzunehmen.

Die Funktion *t_accept()* nimmt die Verbindungsanforderung an. Der erste Parameter der Funktion *t_accept()* gibt den Transportendpunkt an, an dem die Anforderung empfangen wurde, der zweite Parameter gibt den Transportendpunkt an, an dem die Anforderung bestätigt werden soll.
Eine Anforderung kann an demselben Transportendpunkt bestätigt werden, an dem sie empfangen wurde. In diesem Fall können andere Clients für die Dauer dieser Verbindung keine Anforderungen stellen.
Der dritte Parameter von *t_accept()* zeigt auf das *t_call*-Objekt der aktuell bearbeiteten Verbindungsanforderung. Dieses Objekt sollte die Adresse des rufenden Clients und die laufende Nummer des *t_listen()*-Aufrufs enthalten. Der Wert von *call->sequence* ist von Bedeutung, falls der Server mehrere Verbindungen verwaltet. Ein entsprechendes Beispiel finden Sie im Abschnitt „Ereignisgesteuerter Server“ auf Seite 220.

Um das vorliegende Beispiel einfach zu halten, beendet der Server das Programm, wenn der Aufruf *t_open()* scheitert. *exit(2)* schließt den Transportendpunkt, der *listen_fd* zugeordnet ist. Der Transportanbieter sendet damit dem Client eine Nachricht, dass die Verbindung abgebrochen wurde und der Verbindungsaufbau nicht erfolgreich war. Der Aufruf *t_connect()* scheitert, und *t_errno* wird auf TLOOK gesetzt.

Die Ausführung von *t_accept()* kann scheitern, falls ein asynchrones Ereignis am empfangenden Transportendpunkt eintrifft, bevor die Verbindung angenommen ist. *t_errno* wird dann auf TLOOK gesetzt. Tabelle 9 auf Seite 192 zeigt, dass genau eines der beiden folgenden Ereignisse eintreffen kann:

- Eine Abbruchbenachrichtigung für die zuvor gemeldete Verbindungsanforderung ist eingetroffen, d.h. der Client, der die Verbindungsanforderung gesendet hat, möchte die Verbindung abbrechen.

  Wenn ein Abbruchwunsch eintrifft, muss der Server sofort durch einen *t_rcvdis()*-Aufruf den Grund des Auftrags analysieren. Die Funktion *t_rcvdis()* hat als Parameter einen Zeiger auf ein Objekt vom Datentyp *t_discon* (siehe Seite 283). Das *t_discon*-Objekt wird benötigt, um die Abbruchbedingung zu speichern. Im vorliegenen Beispiel wird der Grund für den Abbruch nicht abgefragt; daher ist der Parameter auf NULL gesetzt. Nach Empfang der Abbruchbedingung schließt *accept_call()* den Transportendpunkt und liefert ein DISCONNECT als Ergebnis. Dies informiert den Server, dass die Verbindung vom Client geschlossen worden ist.

- Während der Ausführung von *t_accept()* ist eine neue Verbindungsanforderung eingetroffen.

  Im vorliegenden Beispiel weist der Server diese Verbindungsanforderung zurück, um die aktuell bearbeitete Verbindungsanforderung ungestört annehmen zu können. Im Einzelnen verfährt der Server dabei wie folgt:

  1. Mit *t_alloc()* legt der Server ein neues Objekt vom Typ *struct t_call* an.
  2. Anschließend nimmt der Server die neue Verbindungsanforderung mit *t_listen()* entgegen. *t_listen()* liefert im Feld *refuse_call->sequence* ein eindeutiges Kennzeichen für die neue Verbindungsanforderung zurück.
  3. Mit *t_snddis()* weist der Server die neue Verbindungsanforderung zurück.
  4. Nach Freigabe des durch *refuse_call* referenzierten *t_call*-Objekts mit *t_free()* wiederholt der Server den *t_accept()*-Aufruf.

Die Transportverbindung ist mit dem neu erzeugten Transportendpunkt erstellt worden. Der Empfangsendpunkt kann dadurch neue Verbindungsanforderungen behandeln.

Datenübertragung am Beispiel des Client-/Server-Modells

Wenn die Verbindung einmal hergestellt ist, können Client und Server mit dem Datenaustausch beginnen. Hierfür verwenden sie die Funktionen *t_snd()* und *t_rcv()*. Von diesem Zeitpunkt an unterscheidet der Transportanbieter nicht mehr zwischen Client und Server. Jeder Benutzer kann Daten senden, Daten empfangen und die Verbindung beenden. Der Transportanbieter bietet eine gesicherte, die Sendereihenfolge erhaltende Übertragung der Daten über eine bestehende Verbindung.
Im Beispiel überträgt der Server eine Datei zum Client über die bestehende Transportverbindung.

Senden der Daten durch den Server

Der Server organisiert die Datenübertragung, indem er einen neuen Prozess erzeugt, der die Daten zum Client schickt. Der Vaterprozess wartet auf weitere Verbindungsanforderungen, während der Sohnprozess die Daten überträgt.

Die Funktion *run_service()* wird aufgerufen, um diesen Sohnprozess zu erzeugen. Der folgende Ausschnitt aus der Definition von *run_service()* veranschaulicht dieses Vorgehen:

```
run_service(listen_fd)
int listen_fd;
{
   int nbytes;
   FILE *logfp;  /* Zeiger auf die Protokolldatei */
   char buf[1024];

   switch (fork()) {

   case -1:
      perror("fork gescheitert");
      exit(20);
      break;
   default: /* Vaterprozess */

   /* Schliessen von conn_fd und beenden der Funktion */
   if (t_close(conn_fd) < 0) {
      t_error("t_close() gescheitert für conn_fd");
      exit(21);
   }
   return;
```

```
case 0: /* child */

   /* schließen von listen_fd und übertragen der Datei */
   if (t_close(listen_fd) < 0) {
      t_error("t_close() gescheitert für listen_fd");
      exit(22);
   }
   if (t_look(conn_fd) != 0) { /* ist Verbindungsabbruch da? */
      fprintf(stderr, "t_look: nicht erwartetes Ereignis \n");
      exit(25);
   }

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

Nach dem *fork()* kehrt der Vaterprozess wieder zur Hauptschleife zurück und wartet auf neue Verbindungsanforderungen.

Währenddessen verwaltet der Sohnprozess die neu aufgebaute Verbindung. Falls der Aufruf *fork()* scheitert, schließt *exit()* die aufgebaute Verbindung und sendet eine Abbruchmeldung an den Client. Dadurch scheitert dann der Aufruf *t_connect()* des Client.

Der Sohnprozess liest 1024 byte der Protokolldatei und sendet die Daten mit dem *t_snd()*-Aufruf an den Client. *buf* zeigt auf den Anfang des Datenpuffers, und *nbytes* gibt die Anzahl der zu übertragenden Zeichen an.

Wenn der Benutzer dem Transportanbieter zu viele Daten zur Übertragung zur Verfügung stellt, kann der Transportanbieter die Annahme verweigern, um die Flusskontrolle sicherzustellen. In diesem Fall wird der Aufruf *t_snd()* blockiert, bis die Flusskontrolle wieder freigegeben ist und mit der Übertragung fortgefahren werden kann. Der *t_snd()*-Aufruf wird dann nicht beendet, bevor dem Transportanbieter so viele Zeichen übergeben worden sind, wie der Wert der Variablen *nbytes* angibt.

Die Funktion *t_snd()* kontrolliert nicht, ob ein Abbruchwunsch ankam, bevor die Daten an den Transportanbieter übergeben werden. Bedingt durch den Datenverkehr in nur einer Richtung, ist es dem Benutzer außerdem nicht möglich, ankommende Ereignisse zu behandeln. Wenn zum Beispiel die Verbindung unterbrochen wird, sollte der Benutzer informiert werden, dass Daten verloren gehen könnten. Der Benutzer kann *t_look()* aufrufen, um vor jedem *t_snd()*-Aufruf zu prüfen, ob es ankommende Ereignisse gab.

Empfang der Daten durch den Client

Im Beispiel überträgt der Server über die bestehende Transportverbindung eine Datei zum Client. Der Client empfängt die Datei und gibt sie auf der Standardausgabe aus. Um die Daten zu empfangen, verwendet der Client folgendes Programmstück:

```
while ((nbytes = t_rcv(fd, buf, 1024, &flags)) != -1)
   if (fwrite(buf, 1, nbytes, stdout) == 0) {
      fprintf(stderr, "fwrite gescheitert \n");
      exit(5);
   }
}
```

Der Client ruft die Funktion *t_rcv()* auf, um die ankommenden Daten zu empfangen. Wenn keine Daten verfügbar sind, wird der Prozess durch den Aufruf *t_rcv()* solange blockiert, bis Daten verfügbar sind. Dann liefert *t_rcv()* die Anzahl der Bytes zurück, die im Empfangspuffer *buf* bereitstehen (maximal 1024). Der Client schreibt dann die empfangenen Daten auf die Standardausgabe. Die Datenübertragung wird beendet, wenn der Aufruf *t_rcv()* scheitert. Das ist dann der Fall, wenn ein Wunsch nach Verbindungsabbau empfangen wird. Eine Erklärung hierzu finden Sie auf der folgenden Seite.

Falls der Aufruf *fwrite()* scheitert, wird das Programm beendet und der Transportendpunkt geschlossen. Das Schließen eines Transportendpunkts (durch *exit()* oder *t_close()*) in der Datenübertragungsphase bewirkt den Abbruch der Verbindung; der Kommunikationspartner erhält eine Abbruchnachricht.

Verbindungsabbau am Beispiel des Client-/Server-Modells

Wie bereits erwähnt, gibt es zwei unterschiedliche Formen des Verbindungsabbaus, die vom Transportanbieter unterstützt werden können:

– Der Verbindungsabbruch beendet eine Verbindung sofort. Dies kann zu Datenverlust führen, falls noch nicht alle Daten den Empfänger erreicht haben.

  Mit dem Aufruf der Funktion *t_snddis()* kann jeder Benutzer einen solchen Abbruch erreichen. Falls innerhalb des Transportanbieters Probleme auftreten, kann auch der Transportanbieter einen Verbindungsabbruch erzeugen.

  Wenn die Abbruchnachricht den Empfänger erreicht, muss dieser die Funktion *t_rcvdis()* aufrufen, um die Nachricht zu empfangen. *t_rcvdis()* liefert als Ergebnis einen Wert zurück, der den Grund für den Verbindungsabbruch angibt. Dieser Wert ist abhängig vom verwendeten Transportanbieter und sollte bei protokollunabhängigen Programmen nicht interpretiert werden.

– Der *geordnete Verbindungsabbau* beendet eine Verbindung erst dann, wenn alle Daten übertragen worden sind.

Jeder Transportanbieter muss die erste Variante, d.h. den Verbindungsabbruch, unterstützen. Im Beispiel wird unterstellt, dass der Transportanbieter außerdem den geordneten Verbindungsabbau gestattet.

Verbindungsabbau durch den Server

Wenn alle Daten übertragen sind, kann der Server den geordneten Abbau der Verbindung wie folgt einleiten:

```
if (t_sndrel(conn_fd) < 0) {
   t_error("t_sndrel() gescheitert");
   exit(27);
}
```

Die Verbindung wird erst abgebaut, wenn beide Benutzer einen Abbruchwunsch gesendet und von der Gegenseite eine Bestätigung erhalten haben (siehe Seite 161).

Verbindungsabbau durch den Client

Der Verbindungsabbau findet aus der Sicht des Clients in der gleichen Art und Weise statt wie aus der Sicht des Servers. Wie bereits erwähnt, empfängt der Client Daten, bis der Aufruf *t_rcv()* scheitert. Wenn der Server entweder *t_snddis()* oder *t_sndrel()* aufruft, scheitert der Aufruf *t_rcv()*, und *t_errno* wird auf T_LOOK gesetzt. Der Client behandelt diese Situation wie folgt:

```
   if ((t_errno == TLOOK)  &&  (t_look(fd) == T_ORDREL)) {
      if (t_rcvrel(fd) < 0) {
         t_error("t_rcvrel() gescheitert");
         exit(6);
      }
      if (t_sndrel(fd) < 0) {
         t_error("t_sndrel() gescheitert");
         exit(7);
      }
      exit(0);
   }
   t_error("t_rcv() gescheitert");
   exit(8);
}
```

Wenn am Transportendpunkt des Clients ein Ereignis auftritt, überprüft der Client, ob der erwartete Auftrag zum geordneten Verbindungsabbau angekommen ist. Ist dies der Fall, so ruft der Client *t_rcvrel()* auf, um die Anforderung zu erhalten. Danach ruft der Client *t_sndrel()* auf. Dies zeigt dem Server an, dass auch der Client bereit ist, die Verbindung abzubauen. An diesem Punkt wird das Client-Programm beendet, wodurch auch der Transportendpunkt geschlossen wird.

Falls der Transportanbieter den eben beschriebenen geordneten Verbindungsabbau nicht unterstützt, müssen die Benutzer den abbruchartigen Verbindungsabbau verwenden. Dabei müssen die Benutzer selbst dafür sorgen, dass durch den Verbindungsabbau keine Daten verloren gehen. Zum Beispiel kann eine bestimmte Byte-Kombination anzeigen, dass die Verbindung beendet werden soll. Es gibt viele Möglichkeiten, Datenverlusten vorzubeugen. Jede Anwendung und jedes höhere Protokoll muss über einen entsprechenden Mechanismus verfügen, der sich der gegebenen Transportumgebung anpasst.

7.2 Verbindungsloser Dienst

Der verbindungslose Dienst ist paketorientiert und unterstützt die Übertragung von Datagrammen. Datagramme sind vollständig adressierte Einheiten von Daten, die aus Sicht des Transportanbieters keinerlei logische Beziehung zueinander haben.

Verbindungslose Dienste sind interessant für Anwendungen, die

– nur kurzzeitig mit einem Partner kommunizieren,
– dynamisch konfigurierbar sind,
– keine garantierte Auslieferung der Daten in der Sendefolge benötigen.

Verbindungslose Dienste werden somit vorzugsweise für kurze Auftrag-/Antwort-Dialoge eingesetzt, wie sie z.B. für Auftragssysteme typisch sind.

7.2.1 Phasen des verbindungslosen Dienstes

Der verbindungslose Transportdienst besteht aus den beiden Phasen:

– lokale Verwaltung
– Datenübertragung

Lokale Verwaltung

Bei der lokalen Verwaltung werden die gleichen Funktionen benötigt wie beim verbindungsorientierten Dienst (siehe Abschnitt „Verbindungsorientierter Dienst“ auf Seite 158).

Datenübertragung

Die Datenübertragung ermöglicht es dem Benutzer, Datagramme an einen anderen Benutzer zu senden. Jedes Datagramm muss die vollständige Zieladresse enthalten. Dieser auf Nachrichten basierende Datenaustausch wird von den Funktionen *t_sndudata()* und *t_rcvudata()* unterstützt.

Tabelle 6 zeigt die Funktionen für die verbindungslose Datenübertragung.

Funktion	Beschreibung
t_rcvudata()	empfängt eine Nachricht von einem anderen Benutzer.
t_rcvuderr()	empfängt eine Fehlerinformation über eine zuvor gesendete Nachricht.
t_sndudata()	sendet eine Nachricht an einen bestimmten Benutzer.

Tabelle 6: Funktionen für die verbindungslose Datenübertragung

Zusammenspiel der Funktionen des verbindungslosen Dienstes

In Bild 5 ist das Zusammenspiel der XTI-Funktionen veranschaulicht, die die beiden Phasen des verbindungslosen Dienstes realisieren. Ausführlich beschrieben sind die einzelnen XTI-Funktionen im Kapitel „Bibliotheksfunktionen von XTI(POSIX)“ auf Seite 235.

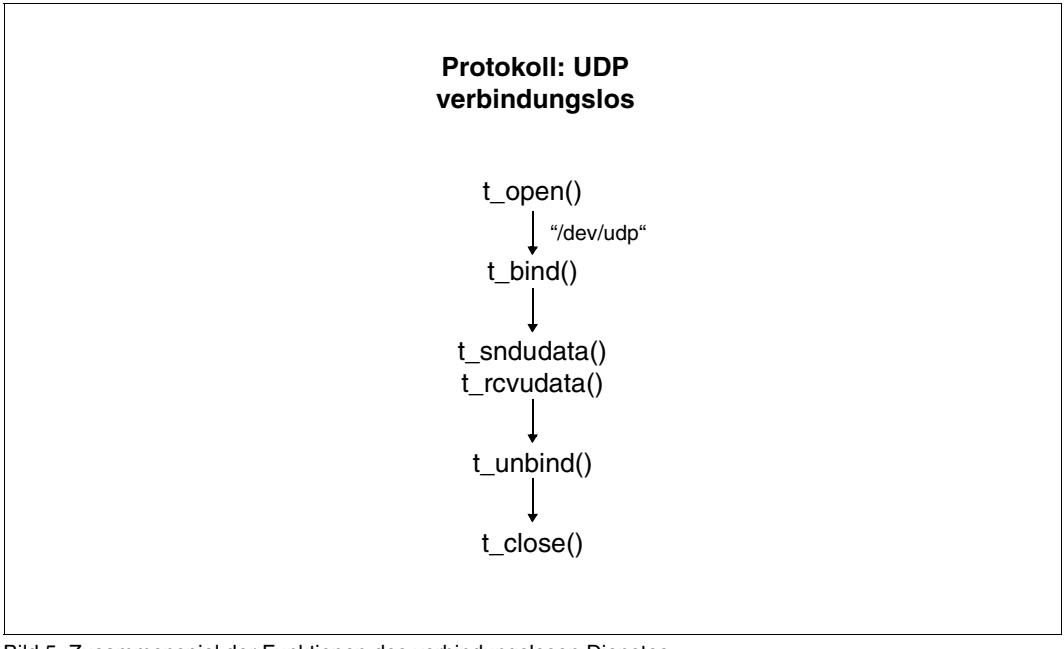

Bild 5: Zusammenspiel der Funktionen des verbindungslosen Dienstes

7.2.2 Verbindungsloser Dienst am Beispiel eines Auftragssystems

Der verbindungslose Dienst wird am Beispiel eines Auftragssystems näher erläutert: Der Server wartet auf ankommende Aufträge, bearbeitet und beantwortet sie.

Lokale Verwaltung am Beispiel eines Auftragssystems

Wie beim verbindungsorientierten Dienst muss der Benutzer vor der Datenübertragung die lokale Verwaltung durchführen. Der Benutzer muss einen entsprechenden verbindungslosen Dienst mit *t_open()* aufrufen und seine Adresse mit *t_bind()* an den Transportendpunkt binden.

Der Benutzer kann die Funktion *t_optmgmt()* verwenden, um die Leistungsmerkmale des Protokolls zu ändern. Wie beim verbindungsorientierten Dienst hat jeder Transportanbieter seine eigenen Leistungsmerkmale. Deshalb macht die Verwendung von *t_optmgmt()* die Programme vom verwendeten Protokoll abhängig.

Mit den folgende Definitionen und Aufrufen führt der **Server** die lokale Verwaltung durch:

```
#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("Der Transportanbieter kann nicht geöffnet werden");
      exit(1);
   }

   if ((bind = (struct t_bind *)t_alloc(fd,T_BIND, T_ADDR)) == NULL) {
      t_error("t_alloc() der t_bind-Struktur gescheitert");
      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() gescheitert");
   exit(3);
}
```

Mit dem Aufruf von *t_open()* erzeugt der Server einen Transportendpunkt.

Mit dem Aufruf von *t_bind()* bindet der Server eine bestimmte Adresse an den Transportendpunkt, sodass mögliche Clients den Server erkennen und auf ihn zugreifen können. Mit *t_alloc()* legt der Server ein Objekt vom Datentyp *t_bind* an und versorgt in der Komponente *addr* des *t_bind*-Objekts die Komponenten *buf* und *len* mit den entsprechenden Werten. Die Adresse selbst wird dabei entsprechend der Adressstruktur der Internet-Kommunikationsdomäne strukturiert.

Ein entscheidender Unterschied zwischen verbindungsorientiertem und verbindungslosem Dienst besteht darin, dass der Inhalt der *t_bind*-Komponente *qlen* im verbindungslosen Dienst keine Bedeutung hat: Es können nämlich alle Benutzer Datagramme empfangen, sobald der Aufruf *t_bind()* beendet ist. Der Transportanbieter definiert beim verbindungs orientierten Dienst während des Verbindungsaufbaus eine Client-/Server-Beziehung. Eine solche Beziehung existiert im verbindungslosen Modus nicht. Bei diesem Beispiel ist es nicht der Transportanbieter, der eine Client-/Server-Beziehung definiert, sondern die Art der Anwendung.

Datenübertragung am Beispiel eines Auftragssystems

Sobald der Benutzer eine Adresse an den Transportendpunkt gebunden hat, kann er Datagramme senden und empfangen. Jede gesendete Nachricht wird von der Adresse des Empfängers begleitet.

Die nachstehende Folge von Aufrufen zeigt den **Server** in der Datenübertragungsphase:

```
if ((ud = (struct t_unitdata *)t_alloc(fd,T_UNITDATA, T_ALL)) == NULL) {
   t_error("t_alloc() der t_unitdata-Struktur gescheitert");
   exit(5);
}
```

```
    if ((uderr = (struct t_uderr *)t_alloc(fd, T_UDERROR, T_ALL)) == NULL) {
       t_error("t_alloc() der t_uderr-Struktur gescheitert");
       exit(6);
    }
    for(;;) {
       if (t_rcvudata(fd, ud, &flags) < 0) {
          if (t_errno == TLOOK) {

                  /*
                   *  Fehler bei einem früher gesendeten Datagramm
                   */

                  if (t_rcvuderr(fd, uderr) < 0) {
                  t_error(„t_rcverr gescheitert“);
                  exit(7);
                  }
                  fprintf(stderr,
                  "fehlerhaftes Datagramm, error = %d \n",
                  uderr->error);
                  continue;
          }
       t_error("t_rcvudata() gescheitert");
       exit(8);
       }

       /*
        * query() bearbeitet den Auftrag und schreibt die
        * Antwort in ud->udata.buf und die Länge in ud->udata.len
        */

       query(ud);

       if (t_sndudata(fd, ud, 0) < 0) {
          t_error("t_sndudata() gescheitert");
          exit(9);
       }
    }
}

query()
{
/* Aus Vereinfachungsgründen nur ein Abschnitt */

}
```

Um Datagramme speichern zu können, muss der Server zunächst mit *t_alloc()* ein Objekt vom Datentyp *struct t_unitdata* anlegen.

Die Struktur *t_unitdata* ist in <xti.h> wie folgt deklariert:

```
struct t_unitdata {
   struct netbuf addr;
   struct netbuf opt;
   struct netbuf udata;
};
```

addr enthält für ankommende Datagramme die Absenderadresse. Für zu sendende Datagramme enthält *addr* die Empfängeradresse. *opt* gibt mögliche Optionen des verwendeten Protokolls an, die auf dieses Datagramm angewendet werden sollen. *udata* enthält die Benutzerdaten. *addr*, *buf* und *udata* müssen mit genügend großen Puffern versehen werden, um ankommende Datagramme zu speichern. Wie im Abschnitt „Verbindungsorientierter Dienst“ auf Seite 158 beschrieben, gewährleistet dies die Angabe T_ALL beim Aufruf von *t_alloc()*. Die Komponente *maxlen* jeder Komponente (vom Typ *struct netbuf*) des erzeugten *t_unitdata*-Objekts wird von *t_alloc()* mit dem entsprechenden Wert versorgt.
Der Server legt auch ein Objekt vom Typ *struct t_uderr* an, um Datagrammfehler bearbeiten zu können (siehe Seite 188).

Der Server läuft in einer Endlosschleife. Er empfängt Aufträge, bearbeitet diese und antwortet den Clients. Zuerst wird *t_rcvudata()* aufgerufen, um den nächsten Auftrag zu empfangen. *t_rcvudata()* empfängt das nächste mögliche Datagramm. Sollte kein Datagramm verfügbar sein, so blockiert *t_rcvudata()* den Prozess, bis ein Datagramm empfangen wird. Der zweite Parameter des Aufrufs *t_rcvudata()* spezifiziert das *t_unitdata*-Objekt, in dem das Datagramm abgespeichert werden soll.

Der dritte Parameter (*flags*) muss ein Zeiger auf einen Integer-Wert sein. Dieser Wert kann beim Beenden von *t_rcvudata()* auf T_MORE gesetzt sein, um anzuzeigen, dass der Puffer *udata* nicht groß genug war für die Aufnahme des gesamten Datagramms. In diesem Fall liefern weitere Aufrufe von *t_rcvudata()* den restlichen Teil des Datagramms.
Da der Puffer in diesem Beispiel mit *t_alloc()* angelegt wurde, kann dieser Fall nicht eintreten und der Server braucht *flags* nicht zu prüfen.

Bei erfolgreichem Empfang eines Datagramms ruft der Server *query()* auf, um den Auftrag zu bearbeiten.

Datagrammfehler

Wenn der Transportanbieter ein mit *t_sndudata()* übergebenes Datagramm nicht bearbeiten kann, wird dem Benutzer ein Fehler T_UDERR gemeldet. Bei diesem Fehler werden Adresse und Optionen des Datagramms sowie ein protokollabhängiger Fehlerwert zurückgeliefert. Die geschilderte Situation kann z.B. eintreten, wenn der Transportanbieter die angegebene Zieladresse nicht findet.

Von jedem Protokoll wird erwartet, dass es alle Ursachen angibt, warum ein Datagramm nicht gesendet wurde.

Die Fehleranzeige gibt keine Auskunft darüber, ob das Datagramm erfolgreich abgeschickt worden ist. Das Transportprotokoll entscheidet, wie die Fehleranzeige benutzt wird. Es sei hier nochmals betont, dass der verbindungslose Dienst keine zuverlässige Zustellung der Daten garantiert.

Der Server wird von dem Fehler unterrichtet, sobald er versucht, ein Datagramm zu empfangen. Der Aufruf *t_rcvudata()* scheitert, und *t_errno* wird auf TLOOK gesetzt. Wenn *t_errno* auf TLOOK gesetzt ist, kann nur ein T_UDERR aufgetreten sein, so dass der Server *t_rcvuderr()* aufruft, um die Fehlerursache festzustellen. Der zweite Parameter beim Aufruf *t_rcvuderr()* ist ein zuvor angelegtes Objekt vom Datentyp *struct t_uderr*. Dieses Objekt wird vom Aufruf *t_rcvuderr()* mit Werten versorgt.

Die Struktur *t_uderr* ist in <xti.h> wie folgt deklariert:

```
struct t_uderr {
   struct netbuf addr;
   struct netbuf opt;
   long error;};
```

addr und *opt* geben die Zieladresse und die für dieses Datagramm gesetzten Optionen an. *error* zeigt einen protokollabhängigen Fehlerwert an, der spezifiziert, warum das Datagramm nicht bearbeitet worden ist. Der Server gibt den Fehlerwert aus und kehrt dann in die normale Schleife zurück.

7.3 Zustände und Zustandsübergänge

In den folgenden Tabellen werden beschrieben:

- Zustände der Transportschnittstelle
- Ereignisse der Transportschnittstelle und *t_look()*
- gesendete Ereignisse
- ankommende Ereignisse
- Zustandsübergänge auf Grund von Aktionen des Transportsystembenutzers
- Zustandsübergänge der Transportschnittstelle
- Ereignisse beim Fehler TLOOK

Zustände der Transportschnittstelle

In der folgenden Tabelle sind die Zustände beschrieben, die verwendet werden, um die Zustandsübergänge der Transportschnittstelle zu beschreiben.

Zustand	Bedeutung	Diensttyp
T_UNINIT	nicht initialisiert; Anfangs- und Endzustand der Schnittstelle	T_COTS, T_COTS_ORD, T_CLTS
T_UNBND	initialisiert, aber nicht eingebunden	T_COTS, T_COTS_ORD, T_CLTS
T_IDLE	keine Verbindung aufgebaut	T_COTS, T_COTS_ORD, T_CLTS
T_OUTCON	gesendete Verbindung wartet auf den Server	T_COTS, T_COTS_ORD
T_INCON	ankommende Verbindung wartet auf den Server	T_COTS, T_COTS_ORD
T_DATAXFER	Datenübertragung	T_COTS, T_COTS_ORD
T_OUTREL	geordneter Verbindungsabbau (wartet auf Bestätigung für den geordneten Verbindungsabbau)	T_COTS_ORD
T_INREL	ankommender geordneter Verbindungsabbau (wartet auf eine Anforderung für den geordneten Verbindungsabbau)	T_COTS_ORD

Tabelle 7: Zustände der Transportschnittstelle

Ereignisse an der Transportschnittstelle und t_look()

Mit Hilfe der Funktion *t_look()* kann der Benutzer feststellen, welches Ereignis aufgetreten ist, wenn ein Fehler TLOOK gemeldet wird. Der Fehler TLOOK hat eine besondere Bedeutung für die Transportschnittstelle. TLOOK informiert den Benutzer, wenn eine Funktion der Schnittstelle durch ein unerwartetes asynchrones Ereignis am gegebenen Transportendpunkt unterbrochen wurde. Ein von TLOOK angezeigter Fehler darf daher nicht als ein Fehler in der Schnittstelle interpretiert werden. Die aufgerufene Funktion wird auf Grund des anstehenden Ereignisses nicht ausgeführt

Folgende Ereignisse der Transportschnittstelle sind definiert:

Ereignis	**Bedeutung**
T_LISTEN	Eine Verbindungsanforderung ist am Transportendpunkt angekommen. T_LISTEN kann nur bei einem Transportendpunkt auftreten, dem eine Adresse mit *qlen* > 0 zugeordnet ist.
T_CONNECT	Die Bestätigung einer vorher gesendeten Verbindungsanforderung ist angekommen. Die Bestätigung kommt, wenn der Server eine Verbindungsanforderung annimmt.
T_DATA	Benutzerdaten sind angekommen.
T_DISCONNECT	Eine Nachricht, dass eine Verbindung abgebrochen oder zurückgewiesen wurde, ist angekommen.
T_ORDREL	Der Auftrag für einen geordneten Verbindungsabbau ist angekommen.
T_UDERR	Die Benachrichtigung über einen Fehler bei einem zuvor gesendeten Datagramm ist angekommen.

Tabelle 8: Ereignisse an der Transportschnittstelle

Ereignisse beim Fehler TLOOK

Tabelle 9 beschreibt die asynchronen Ereignisse, die zum Abbruch einer XTI-Funktion mit dem Fehler TLOOK führen.

XTI-Funktion	Ereignis
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

Tabelle 9: Ereignisse beim Fehler TLOOK

Wenn an einem Transportendpunkt die Ausführung einer XTI-Funktion zu einem Fehler TLOOK führt, liefern nachfolgende Aufrufe derselben oder einer anderen XTI-Funktion, die vom gleichen TLOOK betroffen ist, solange den Fehler TLOOK zurück, bis das auslösende Ereignis behandelt ist. Das den Fehler TLOOK auslösende Ereignis können Sie mit der XTI-Funktion *t_look()* identifizieren und anschließend mit einer geeigneten anderen XTI-Funktion behandeln.

Gesendete Ereignisse

In Tabelle 10 auf Seite 194 sind die gesendeten Ereignisse beschrieben. Sie entsprechen den Rückgabewerten der angegebenen Transportfunktionen, wobei diese Funktionen einen Auftrag oder eine Antwort an den Transportanbieter senden.

In der Tabelle werden einige Ereignisse (z.B. *accept*) nach dem Kontext unterschieden, in dem sie auftreten. Der Kontext hängt von den Werten der folgenden Variablen ab:

- *ocnt*
  Anzahl der anstehenden Verbindungsanforderungen
- *fd*
  Dateideskriptor des aktuellen Transportendpunkts
- *refsd*
  Dateideskriptor des Transportendpunkts, an dem eine Verbindung angenommen wird

Ereignis	Bedeutung	Diensttyp
opened	erfolgreiches Beenden von *t_open()*	T_COTS, T_COTS_ORD, T_CLTS
bind	erfolgreiches Beenden von *t_bind()*	T_COTS, T_COTS_ORD, T_CLTS
optmgmt	erfolgreiches Beenden von *t_optmgmt()*	T_COTS_ORD, T_CLTS
unbind	erfolgreiches Beenden von *t_unbind()*	T_COTS, T_COTS_ORD, T_CLTS
closed	erfolgreiches Beenden von *t_close()*	T_COTS, T_COTS_ORD, T_CLTS
connect1	erfolgreiches Beenden von *t_connect()* im synchronen Betrieb	T_COTS, T_COTS_ORD
connect2	TNODATA-Fehler bei *t_connect()* im asynchronen Betrieb oder TLOOK-Fehler wegen einer am Kommunikationsendpunkt eintreffenden Anforderung zum Verbindungsabbau	T_COTS, T_COTS_ORD
accept1	erfolgreiches Beenden von *t_accept()* mit *ocnt* == 1, *fd* == *resfd*	T_COTS, T_COTS_ORD
accept2	erfolgreiches Beenden von *t_accept()* mit *ocnt* == 1, *fd* != *resfd*	T_COTS, T_COTS_ORD
accept3	erfolgreiches Beenden von *t_accept()* mit *ocnt* >1	T_COTS, T_COTS_ORD
snd	erfolgreiches Beenden von *t_snd()*	T_COTS, T_COTS_ORD
snddis1	erfolgreiches Beenden von *t_snddis()* mit *ocnt* <=1	T_COTS, T_COTS_ORD
snddis2	erfolgreiches Beenden von *t_snddis()* mit *ocnt* >1	T_COTS, T_COTS_ORD
sndrel	erfolgreiches Beenden von *t_sndrel()*	T_COTS_ORD
sndudata	erfolgreiches Beenden von *t_sndudata()*	T_CLTS

Tabelle 10: Gesendete Ereignisse

Ankommende Ereignisse

Die ankommenden Ereignisse entsprechen den erfolgreichen Rückgabewerten der angegebenen Funktion, wobei diese Funktionen Daten oder Informationen über Ereignisse vom Transportanbieter erhalten. Das einzige ankommende Ereignis, das nicht direkt mit dem Rückgabewert einer Funktion zusammenhängt, ist *pass_conn*. Das Ereignis *pass_conn* tritt dann auf, wenn ein Benutzer eine Verbindung an einen anderen Transportendpunkt überträgt. Dieses Ereignis tritt bei einem Transportendpunkt auf, dem die Verbindung übergeben wurde, obwohl keine Funktion der Transportschnittstelle für ihn aufgerufen wurde. Das Ereignis *pass_conn* beschreibt das Verhalten, wenn ein Benutzer eine Verbindung auf einem anderen Transportendpunkt annimmt.

In der folgenden Tabelle werden die Ereignisse *rcvdis* nach dem Kontext unterschieden, in dem sie auftreten. Der Kontext hängt ab vom Wert von *ocnt*. Der Wert von *ocnt* gibt die Anzahl der anstehenden Verbindungsanforderungen auf dem Transportendpunkt an.

Ereignis	Bedeutung	Diensttyp
listen	erfolgreiches Beenden von *t_listen()*	T_COTS, T_COTS_ORD
rcvconnect	erfolgreiches Beenden von *t_rcvconnect()*	T_COTS, T_COTS_ORD
rcv	erfolgreiches Beenden von *t_rcv()*	T_COTS, T_COTS_ORD
rcvdis1	erfolgreiches Beenden von *t_rcvdis()* mit *ocnt* <= 0	T_COTS, T_COTS_ORD
rcvdis2	erfolgreiches Beenden von *t_rcvdis()* mit *ocnt* = = 1	T_COTS, T_COTS_ORD
rcvdis3	erfolgreiches Beenden von *t_rcvdis()* mit *ocnt* > 1	T_COTS, T_COTS_ORD
rcvrel	erfolgreiches Beenden von *t_rcvrel()*	T_COTS_ORD
rcvudata	erfolgreiches Beenden von *t_rcvudata()*	T_CLTS
rcvuderr	erfolgreiches Beenden von *t_rcvuderr()*	T_CLTS
pass_conn	Empfang einer übergebenen Verbindung	T_COTS, T_COTS_ORD

Tabelle 11: Ankommende Ereignisse

Zustandsübergänge auf Grund von Aktionen des Transportbenutzers

In den Zustandstabellen, die unter „„Zustandstabellen“ auf Seite 197“ aufgeführt sind, werden einige Zustandsübergänge von einer Reihe von Aktionen begleitet, die der Transportdienstbenutzer ausführen muss. Diese Aktionen werden mit der Notation „[*n*]“ dargestellt, wobei *n* die Nummer der auszuführenden Aktion ist.

Es handelt sich dabei um die folgenden Aktionen:

1. Setze die Anzahl der anstehenden Verbindungsanforderungen auf 0.
2. Inkrementiere die Anzahl der anstehenden Verbindungsanforderungen.
3. Dekrementiere die Anzahl der anstehenden Verbindungsanforderungen.
4. Übergib eine Verbindung an einen anderen Transportendpunkt, wie in *t_accept()* angegeben.

Zustandstabellen

In den folgenden Tabellen sind die Zustandsübergänge der Transportschnittstelle beschrieben. Zu einem aktuellen Zustand und einem Ereignis wird der Übergang zum nächsten Zustand angezeigt. Außerdem sind alle Aktionen angegeben, die vom Benutzer des Transportsystems auszuführen sind; solche Aktionen sind mit „[*n*]“ gekennzeichnet.

Der Inhalt eines Kästchens gibt jeweils den Folgezustand an. Dieser ist abhängig vom aktuellen Zustand (im Spaltenkopf angegeben) und dem aktuellen empfangenen oder gesendeten Ereignis (links in der betreffenden Zeile angegeben). Ein leeres Kästchen bedeutet, dass die entsprechende Zustands-/Ereignis-Kombination ungültig ist. Zusammen mit dem Folgezustand kann jedes Kästchen eine Aktionsliste enthalten (wie im vorherigen Abschnitt erläutert wurde). Der Transportdienstbenutzer muss die Aktionen in der angegebenen Reihenfolge ausführen.

Beim Lesen der Zustandstabellen sollten Sie folgende Punkte beachten:

– In den Zustandstabellen wird auch die Funktion *t_close()* behandelt (siehe Ereignis *closed* in Tabelle 12). Um einen Transportendpunkt zu schließen, kann *t_close()* jedoch von jedem Zustand aus aufgerufen werden. Wenn die Adresse an einen Transportendpunkt gebunden ist, wird bei Aufruf von *t_close()* die Adresse automatisch freigegeben.

– Der Transportanbieter erkennt, wenn ein Transportdienstbenutzer eine Funktion außerhalb der vorgegebenen Reihenfolge aufruft. In diesem Fall weist der Transportanbieter die Funktion ab und setzt *t_errno* auf TOUTSTATE. Der Zustand ändert sich nicht.

– Wenn ein anderer Transportfehler auftritt, ändert sich der Zustand normalerweise nicht. Eine Ausnahme hiervon ist ein Fehler TLOOK oder TNODATA bei *t_connect()*. Auf weitere Ausnahmen wird bei der Beschreibung der Funktionen im Kapitel „Bibliotheksfunktionen von XTI(POSIX)“ auf Seite 235 explizit hingewiesen. Bei den Zustandstabellen wird die korrekte Verwendung der Transportschnittstelle angenommen.

– Die Funktionen *t_getinfo()*, *t_getstate()*, *t_alloc()*, *t_free()*, *t_sync()*, *t_look()* und *t_error()* sind in den Zustandstabellen nicht aufgeführt, da sie den Zustand nicht beeinflussen.

Jeweils in einer gesonderten Tabelle behandelt werden im Folgenden die Zustandsübergänge in den Phasen:

– lokale Verwaltung (verbindungsorientierter und vebindungsloser Dienst)

– Datenübertragung im verbindungslosen Dienst

– Verbindungsaufbau, Datenübertragung, Verbindungsabbau im verbindungsorientierten Dienst

Ereignis	**Zustand**		
	T_UNINIT	**T_UNBND**	**T_IDLE**
opened	T_UNBND		
bind		T_IDLE [1]	
optmgmt			T_IDLE
unbind			T_UNBND
closed		T_UNINIT	

Tabelle 12: Zustandsübergänge bei der lokalen Verwaltung

Ereignis	**Zustand**
	T_IDLE
sndudata	T_IDLE
rcvudata	T_IDLE
rcvuderr	T_IDLE

Tabelle 13: Zustandsübergänge beim verbindungslosen Dienst

Ereignis	**Zustand**					
	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					

Tabelle 14: Zustandsübergänge beim verbindungsorientierten Dienst

8 Weiterführende Konzepte von XTI(POSIX)

In diesem Kapitel werden einige wichtige weiterführende Konzepte der Transportschnittstelle vorgestellt:

- asynchroner Ausführungsmodus
- gleichzeitige Verwaltung mehrerer Verbindungen durch den Server; ereignisgesteuerter Betrieb mehrerer Verbindungen durch den Server

8.1 Asynchroner Ausführungsmodus

Viele Funktionen der Transportschnittstelle können einen Prozess blockieren, wenn sie auf bestimmte Ereignisse warten oder der Datenfluss den Prozess blockiert. Es gibt aber Situationen, in denen der Benutzer dieses Blockieren verhindern will. So dürfen z.B. zeitkritische Anwendungen niemals blockiert werden. In einem anderen Fall möchte der Prozess weiterarbeiten, während er auf ein Ereignis der Transportschnittstelle wartet.

Daher kann jede Funktion, die den Prozess blockieren könnte, in einem speziellen nicht-blockierenden (asynchronen) Modus ausgeführt werden. Normalerweise blockiert der *t_listen()*-Aufruf den aufrufenden Prozess (Server) solange, bis die Verbindung bestätigt wird. Der Server könnte aber auch periodisch mit dem nicht-blockierenden Aufruf *t_listen()* nachprüfen, ob die Verbindung schon aufgebaut ist. Eingeschaltet wird der asynchrone Modus wird mit dem Parameter O_NONBLOCK für das betreffende Dateikennzeichen. Dies kann mit *t_open()* beim Öffnen des Transportendpunkts erfolgen oder mit einem Aufruf *fcntl()*, bevor eine möglicherweise blockierende Funktion der Transportschnittstelle aufgerufen wird. *fcntl()* kann jederzeit benutzt werden, um den asynchronen Modus an- und auszuschalten.
Alle Programmbeispiele in diesem Kapitel benutzen den voreingestellten synchronen Modus.

8.2 Gleichzeitige Verwaltung mehrerer Verbindungen und ereignisgesteuerter Betrieb

Anhand eines Beispiels werden im Folgenden zwei wichtige Konzepte erläutert:

- Gleichzeitige Verwaltung mehrerer Verbindungen durch den Server:

  Die im Kapitel „Grundlagen von XTI(POSIX)“ auf Seite 157 vorgestellte Server-Anwendung kann immer nur eine Verbindungsanforderung nach der anderen bearbeiten. Die Transportschnittstelle erlaubt jedoch auch die gleichzeitige Bearbeitung mehrerer Verbindungen. Dies ist z.B in folgenden Fällen sinnvoll:

  – Der Server möchte jedem Client eine Priorität zuordnen.

  – Mehrere Clients möchten eine Verbindung zu einem Server aufbauen, der gerade eine Verbindungsanforderung bearbeitet. Wenn der Server immer nur eine Verbindungsanforderung gleichzeitig bearbeiten kann, finden die Clients den Server besetzt vor. Kann der Server jedoch mehrere Verbindungsanforderungen gleichzeitig bearbeiten, finden die Clients den Server nur dann besetzt vor, wenn bereits die maximal möglichen Client-Anforderungen vom Server bearbeitet werden.

- Programmierung eines ereignisgesteuerten Betriebs:

  Der Programmierer kann mit Hilfe der Transportschnittstelle ereignisgesteuerte Programme schreiben. Beim ereignisgesteuerten Server fragt der Prozess einenTrans-portendpunkt permanent ab, ob von der Transportschnittstelle Ereignisse gemeldet werden. Je nach gemeldetem Ereignis ruft der Server dann die entsprechende Schnittstellenfunktion auf.

Das nachfolgende Programmbeispiel verwendet für die lokale Verwaltung die gleichen Definitionen und Funktionsaufrufe wie das Server-Beispiel im Abschnitt „Verbindungsorientierter Dienst“ auf Seite 158. Der im Beispiel verwendete Programmcode ist vollständig und zusammenhängend dargestellt im Abschnitt „Ereignisgesteuerter Server“ auf Seite 220.

```
#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;/* Transportendpunkt des Servers */

/* Zum Speichern der Verbindungen */
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;

   /*
    * Öffnen eines Transportendpunkts und Binden der
    * Adresse. Es werden aber mehrere Endpunkte unterstützt.
    */

   if ((pollfds[0].fd = t_open("/dev/tcp", O_RDWR, NULL)) < 0) {
   t_error("t_open() gescheitert");
   exit(1);
   }

   if ((bind = (struct t_bind *)t_alloc(pollfds[0].fd,
                                        T_BIND, T_ALL)) == NULL) {
      t_error("t_alloc() von t_bind-Struktur gescheitert");
      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() gescheitert");
   exit(3);
}
```

Das von *t_open()* gelieferte Dateikennzeichen wird im ersten Element des *struct pollfd*-Vektors *pollfds* gespeichert (siehe Struktur *pollfd* auf Seite 146). Der Vektor *pollfds* wird später bei einem Aufruf der POSIX-Funktion *poll()* verwendet, um ankommende Ereignisse zu bearbeiten. *poll()* ist eine allgemeine C-Bibliotheksfunktion, die auf Seite 146 beschrieben ist. Zu beachten ist, dass in diesem Beispiel nur ein Transportendpunkt eingerichtet wird. Da der restliche Teil des Beispiels jedoch mehrere Verbindungen vorsieht, sind nur geringfügige Änderungen nötig, um mit diesem Programm mehrere Kommunikationsverbindungen verwalten zu können.

Wesentlich für das vorliegende Beispiel ist, dass der Server *bind->qlen* auf einen Wert >1 setzt und beim Aufruf von *t_bind()* entsprechend übergibt. Dies ermöglicht dem Server, an einem Transportendpunkt mehrere Verbindungsanforderungen zu empfangen. Bei den Beispielen im Kapitel „Grundlagen von XTI(POSIX)“ auf Seite 157 wird vom Server gleichzeitig immer nur eine Verbindung angenommen und bearbeitet. Im vorliegenden Beispiel kann der Server dagegen bis zu MAX_CONN_IND Anforderungen gleichzeitig annehmen. Allerdings kann der Transportanbieter den Wert von *bind->qlen* herabsetzen, wenn er die vom Server gewünschte Anzahl Verbindungen nicht bearbeiten kann.

Nachdem der Server seine Adresse bekannt gegeben hat, verfährt er wie folgt:

```
pollfds[0].events = POLLIN;

for(;;) {
   if (poll(pollfds, NUM_FDS, -1) < 0) {
      perror("poll() gescheitert");
      exit(5);
   }

   for (i = 0; i < NUM_FDS; i++) {

      switch (pollfds[i].revents) {

      default:
              perror("poll gibt Fehlermeldung zurück");
              exit(6);
              break;

      case 0:
              continue;
```

```
            case POLLIN:
                    do_event(i, pollfds[i].fd);
                    service_conn_ind(i, pollfds[i].fd);
            }
        }
    }
}
```

Die Komponente *events* des ersten Elements des *pollfd*-Vektors *pollfds* setzt der Server auf POLLIN, damit er über alle an der Transportschnittstelle ankommenden Ereignisse informiert wird. Der Server geht dann in eine Endlosschleife. In der Endlosschleife wartet der Server mit *poll()* auf Ereignisse an den Transportendpunkten und bearbeitet diese Ereignisse dann entsprechend.

Der Aufruf *poll()* blockiert den Prozess, bis ein Ereignis eintrifft. Nach dem Ende des Aufrufs prüft der Server bei jedem Eintrag (entsprechend einem Transportendpunkt), ob dort ein Ereignis aufgetreten ist. Wenn *revents* auf 0 gesetzt ist, gab es an diesem Transportendpunkt kein Ereignis. Hat *revents* den Wert POLLIN, so ist an diesem Punkt ein Ereignis aufgetreten. In diesem Fall ruft der Server *do_event()* auf, um das Ereignis zu bearbeiten. Hat *revents* einen anderen Wert als POLLIN, so ist ein Fehler an diesem Punkt aufgetreten und das Programm wird beendet.

In jedem Schleifendurchlauf, bei dem ein Ereignis gefunden wird, ruft der Server *service_conn_ind()* (siehe Seite 209) auf, um mögliche anstehende Anforderungen zu bearbeiten. Falls beim Bearbeiten einer Anforderung eine weitere Anforderung ansteht, wird *service_conn_ind* sofort verlassen; die aktuelle Anforderung wird dabei gespeichert, um sie später mit der Funktion *do_event()* zu bearbeiten.

Die Funktion *do_event()* wird bei einem ankommenden Ereignis aufgerufen, um das Ereignis zu bearbeiten. Die Funktion *do_event()* ist wie folgt definiert:

```
do_event(slot, fd)
{
   struct t_discon *discon;
   int i;

   switch (t_look(fd)) {

   default:
      fprintf(stderr,"t_look: unerwartetes Ereignis \n");
      exit(7);
      break;

   case -1:
      t_error("t_look() gescheitert");
      exit(9);
      break;
```

```
    case 0:
       /* wenn POLLIN gemeldet wird, sollte dies nie geschehen */
       fprintf(stderr,"t_look() meldet kein Ereignis\n");
       exit(10);

    case T_LISTEN:

       /*
        * Suchen eines freien Elements im calls-Feld
        */
       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() von t_call-Struktur gescheitert");
          exit(11);
       }

       if (t_listen(fd, calls[slot][i]) < 0) {
          t_error("t_listen() gescheitert");
          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() gescheitert");
          exit(13);
       }
       /*
        * Im calls-Feld Anforderung finden und löschen
        */
       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;
    }
}
```

Die Funktion *do_event()* hat zwei Parameter: eine Nummer *slot* und ein Dateikennzeichen *fd*. *slot* indiziert die Vektoren (Untermatrizen) der globalen Matrix *calls*, deren Elemente Zeiger auf *t_call*-Objekte sind. Jeder abzufragende Transportendpunkt ist durch einen Vektor der Matrix *calls* repräsentiert. Der Wert von *slot* gibt somit den zu bearbeitenden Transportendpunkt an. Die Vektorelemente zeigen auf die *t_call*-Objekte, in denen die ankommenden Anforderungen des zugehörigen Transportendpunkts abgespeichert werden.

Der Aufruf *t_look()* ermittelt das Ereignis, das an dem durch *fd* identifizierten Transportendpunkt aufgetreten ist. Falls ein Verbindungswunsch (T_LISTEN) oder ein Abbruchwunsch (T_DISCONNECT) angekommen ist, wird er entsprechend bearbeitet. Bei anderen Ereignissen wird eine entsprechende Fehlermeldung ausgegeben und das Programm beendet.

Bei einer Verbindungsanforderung sucht *do_event()* einen freien Eintrag im Feld *calls*. Für diesen Eintrag wird nun ein *t_call*-Objekt angefordert. Mit *t_listen()* wird die Anforderung empfangen. In dem Feld muss immer mindestens ein freier Eintrag sein, da *t_bind()* beim Anlegen des Feldes den maximalen Wert von gleichzeitig bearbeitbaren Anforderungen angegeben hat. Die Bearbeitung der Anforderung erfolgt später.

Ein ankommender Abbruchwunsch muss zu einer früher angekommenen Verbindungsanforderung gehören. Dieser Fall tritt ein, falls ein Client eine Verbindungsanforderung schickt und diese sofort wieder abbricht. *do_event()* legt eine *t_discon*-Struktur an, um die Informationen für den Abbruch zu erhalten.

Die Struktur *t_discon* ist in <xti.h> wie folgt deklariert:

```
struct t_discon {
   struct netbuf udata;
   int reason;
   int sequence;
};
```

reason gibt die protokollspezifische Ursache für den Verbindungsabbau an. *sequence* gibt die Nummer der Verbindungsanforderung an, die abgebrochen werden soll.

Die Funktion *t_rcvdis()* wird aufgerufen, um die genannten Informationen zu erhalten. **calls* des Programms, in dem die Anforderungen verwaltet werden, wird dann nach der Anforderung abgesucht, die in der Komponente *sequence* angegeben ist. Wenn die Anforderung gefunden ist, wird der Speicher freigegeben und der Eintrag auf NULL gesetzt.

Falls irgendein Ereignis am Transportendpunkt aufgetreten ist, wird die Funktion *service_conn_ind()* aufgerufen, um alle an diesem Endpunkt anstehenden Anforderungen wie folgt zu bearbeiten:

```
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() gescheitert");
         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() gescheitert");
         exit(16);
      }
      t_free(calls[slot][i], T_CALL);
      calls[slot][i] = NULL;

      run_service(fd);
   }
}
```

Für den angegebenen Endpunkt (*slot*) wird im Feld nach Anforderungen gesucht. Für jede Anforderung öffnet der Server einen Transportendpunkt und nimmt die Anforderung an. Sollte inzwischen ein anderes Ereignis (Verbindungsanforderung oder Verbindungsabbruch) eingetroffen sein, so scheitert der *t_accept()*-Aufruf, und *t_errno* wird auf TLOOK gesetzt.

Ein Benutzer kann keine Anforderung annehmen, falls andere Verbindungsanforderungen oder Abbruchanforderungen an diesem Transportendpunkt anstehen.

Wenn dieser Fehler auftritt, wird *conn_fd* sofort geschlossen und die Funktion verlassen. Die Anforderung bleibt im Feld erhalten und kann daher zu einem späteren Zeitpunkt bearbeitet werden. Damit befindet sich der Server-Prozess wieder in der Hauptschleife, und das Ereignis kann mit dem nächsten Aufruf von *poll()* behandelt werden. Auf diese Weise lassen sich mehrere Aufträge gleichzeitig bearbeiten.

Wenn alle Ereignisse bearbeitet sind, kann die Funktion *service_conn_ind()* die Verbindungen herstellen und die Funktion *run_service()* für die Datenübertragung aufrufen. Die Funktion *run_service()* ist beschrieben im Abschnitt „Datenübertragung am Beispiel des Client-/Server-Modells“ auf Seite 177.

9 Beispiele zu XTI(POSIX)

In Kapitel „Grundlagen von XTI(POSIX)“ auf Seite 157 und Kapitel „Weiterführende Konzepte von XTI(POSIX)“ auf Seite 201“ werden Beispielprogramme abschnittsweise dargestellt und erläutert. Im vorliegenden Kapitel sind diese Beispielprogramme noch einmal vollständig und zusammenhängend abgedruckt.

9.1 Client im verbindungsorientierten Dienst

Das folgende Client-Programm im verbindungsorientierten Dienst ist im Abschnitt „Verbindungsorientiertes Client-/Server-Modell“ auf Seite 164 näher erläutert. Der Client baut eine Transportverbindung mit einem Server auf, erhält dann Daten vom Server und schreibt die Daten auf seine Standardausgabe. Die Verbindung wird mit dem geordneten Verbindungsabbau der Transportschnittstelle aufgelöst. Der Client kann mit jedem in den Beispielen dieses Kapitels vorgestellten verbindungsorientierten Server kommunizieren.

```
#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() gescheitert");
      exit(1);
   }

   if (t_bind(fd, NULL, NULL) < 0) {
      t_error("t_bind() gescheitert");
      exit(2);
   }

   if ((sndcall = (struct t_call *)t_alloc(fd,T_CALL, T_ADDR)) == NULL) {
      t_error("t_alloc() gescheitert");
      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() gescheitert für fd");
        exit(4);
    }

    while ((nbytes = t_rcv(fd, buf, 1024, &flags)) != -1) {
        if (fwrite(buf, 1, nbytes, stdout) == 0) {
            fprintf(stderr, "fwrite() gescheitert\n");
            exit(5);
        }
    }

    if ((t_errno == TLOOK)  &&  (t_look(fd) == T_ORDREL)) {
        if (t_rcvrel(fd) < 0) {
            t_error("t_rcvrel() gescheitert");
            exit(6);
        }
        if (t_sndrel(fd) < 0) {
            t_error("t_sndrel() gescheitert");
            exit(7);
        }
        exit(0);
    }
    t_error("t_rcv() gescheitert");
    exit(8);
}
```

9.2 Server im verbindungsorientierten Dienst

Das folgende Server-Programm für den verbindungsorientierten Dienst ist im Abschnitt „Verbindungsorientiertes Client-/Server-Modell“ auf Seite 164 näher erläutert. Der Server baut eine Transportverbindung mit einem Client auf und übergibt dann eine Protokolldatei an diesen Client. Die Verbindung wird mit dem geordneten Verbindungsabbau der Transportschnittstelle aufgelöst. Der zuvor dargestellte Client im verbindungsorientierten Dienst kann mit dem hier beschriebenen Server kommunizieren.

```
#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;  /* Verbindungsaufbau */

main()
{
   int listen_fd;  /* Transportendpunkt überwachen */
   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() gescheitert für listen_fd");
      exit(1);
   }

   if ((bind = (struct t_bind *)t_alloc(listen_fd,T_BIND, T_ALL)) == NULL) {
      t_error("t_alloc() der Struktur t_bind gescheitert");
      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() für listen_fd gescheitert");
       exit(3);
    }

    if ((call = (struct t_call *)t_alloc(listen_fd,T_CALL, T_ALL)) == NULL) {
       t_error("t_alloc() von t_call-Struktur gescheitert");
       exit(5);
    }

    for(;;) {
       if (t_listen(listen_fd, call) < 0) {
          t_error("t_listen() für listen_fd gescheitert");
          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() für antwortenden fd gescheitert");
       exit(7);
    }

    while (t_accept(listen_fd, resfd, call) < 0) {
       if (t_errno == TLOOK) {
          if (t_look(listen_fd) == T_DISCONNECT) { /* Verbindungsabbruch */
               if (t_rcvdis(listen_fd, NULL) < 0) {
                  t_error("t_rcvdis() gescheitert für listen_fd");
                  exit(9);
               }
               if (t_close(resfd) < 0) {
                  t_error("t_close gescheitert für antwortenden fd");
                  exit(10);
               }
               /* Aufruf beenden und auf weiteren Aufruf warten */
               return(DISCONNECT);
          } else { /* neues T_LISTEN; Ereignis löschen */
```

```
                if ((refuse_call =
                   (struct t_call*)t_alloc(listen_fd,T_CALL,0)) == NULL) {
                   t_error("t_alloc() für refuse_call gescheitert");
                   exit(11);
                }

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

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

                if (t_free((char *)refuse_call, T_CALL) < 0) {
                   t_error("t_free() für refuse_call gescheitert");
                   exit(14);
                }
             }
          } else {
             t_error("t_accept() gescheitert");
             exit(15);
          }
       }

       return(resfd);
    }

    run_server(listen_fd)
    int listen_fd;
    {
       int nbytes;
       FILE *logfp; /* Dateizeiger auf Log-Datei */
       char buf[1024];

       switch (fork()) {

       case -1:
          perror("fork gescheitert");
          exit(20);
          break;

       default: /* Elternprozess */
          /* conn_fd schließen und wieder als Monitor aktiv sein */
          if (t_close(conn_fd) < 0) {
             t_error("t_close() für conn_fd gescheitert");
```

```
            exit(21);
         }
         return;

      case 0:  /* Kind */

         /* listen_fd schließen und Dienst ausführen */
         if (t_close(listen_fd) < 0) {
            t_error("t_close() für listen_fd gescheitert");
            exit(22);
         }
         if ((logfp = fopen(FILENAME, "r")) == NULL) {
            perror("Logdatei kann nicht geöffnet werden");
            exit(23);
         }

         if (t_look(conn_fd) != 0) { /* gab es eine Unterbrechung? */
            fprintf(stderr, "t_look: unerwartetes Ereignis\n");
            exit(25);
         }

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

         if (t_sndrel(conn_fd) < 0) {
            t_error("t_sndrel() gescheitert");
            exit(27);
         }
         while(t_look(conn_fd) == 0) {
            sleep(1);
         }
         if(t_look(conn_fd) == T_DISCONNECT) {
            fprintf(stderr, "Verbindung abgebrochen\n");
            exit(12);
         }
         exit(0);
      }
   }
```

9.3 Datagramm-orientierter Transaktionsserver

Das folgende Programm für ein Auftragssystem im verbindungslosen Modus ist im Abschnitt „Verbindungsloser Dienst am Beispiel eines Auftragssystems“ auf Seite 184 näher erläutert. Der Server wartet auf eingehende Aufträge für Datenpakete, behandelt dann jeden Auftrag und sendet eine Antwort.

```
#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("Öffnen von /dev/udp nicht möglich");
      exit(1);
   }

   if ((bind = (struct t_bind *)t_alloc(fd,
      T_BIND, T_ADDR)) == NULL) {
      t_error("t_alloc() der t_bind-Struktur gescheitert");
      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() gescheitert");
      exit(3);
   }
```

```
    if ((ud = (struct t_unitdata *)t_alloc(fd,
       T_UNITDATA, T_ALL)) == NULL) {
       t_error("t_alloc() von t_unitdata-Structur gescheitert");
       exit(5);
    }

    if ((uderr = (struct t_uderr *)t_alloc(fd,
       T_UDERROR, T_ALL)) == NULL) {
       t_error("t_alloc() von t_uderr-Structur gescheitert");
       exit(6);
    }

    for(;;) {
       if (t_rcvudata(fd, ud, &flags) < 0) {
          if (t_errno == TLOOK) {
                   /*
                    * Fehler wegen vorherigem Datagramm
                    */

                   if (t_rcvuderr(fd, uderr) < 0) {
                      t_error("t_rcvuderr() gescheitert");
                      exit(7);
                   }
                   fprintf(stderr,
                   "Datagramm-Fehler, error = %d\n",
                   uderr->error);
                   continue;
          }
          t_error("t_rcvudata() gescheitert");
          exit(8);
       }
       /*
        * query() bearbeitet den Auftrag und schreibt die
        * Antwort in ud->udata.buf und die Länge in ud->udata.len
        */
       query(ud);

       if (t_sndudata(fd, ud, 0) < 0) {
          t_error("t_sndudata() gescheitert");
          exit(9);
       }
    }
}

query()
{
/* aus Vereinfachungsgründen nur ein Abschnitt */
}
```

9.4 Ereignisgesteuerter Server

Das folgende Serverprogramm für den verbindungsorientierten Dienst ist ab Seite 203 im Kapitel „Weiterführende Konzepte von XTI(POSIX)“ näher erläutert. Der Server verwaltet mehrere Verbindungsanforderungen auf ereignisgesteuerte Art. Jeder der zuvor in diesem Kapitel vorgestellten verbindungsorientierten Clients kann mit diesem Server kommunizieren.

```
#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; /* Verbindung zum Server */

/* Zum Speichern der Verbindungen */
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;

   /*
    * Nur Öffnen und Binden eines Transportendpunkts,
    * obwohl das auch für mehr möglich wäre
    */
   if ((pollfds[0].fd = t_open("/dev/tcp", O_RDWR, NULL)) < 0) {
      t_error("t_open() gescheitert");
      exit(1);
   }

   if ((bind = (struct t_bind *)t_alloc(pollfds[0].fd,
                                        T_BIND, T_ALL)) == NULL) {
      t_error("t_alloc() der t_bind-Structur gescheitert");
      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() gescheitert");
       exit(3);
    }

    pollfds[0].events = POLLIN;

    for(;;) {
       if (poll(pollfds, NUM_FDS, -1) < 0) {
          perror("poll() gescheitert");
          exit(5);
       }

       for (i = 0; i < NUM_FDS; i++) {

          switch (pollfds[i].revents) {

          default:
                  perror(
                  "poll gibt Fehlerereignis zurück");
                  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: unerwartetes Ereignis\n");
   exit(7);
   break;

case -1:
   t_error("t_look() gescheitert");
   exit(9);
   break;

case 0:
   /* Wenn POLLIN zurückgeliefert wird, sollte dies nicht vorkommen */
   fprintf(stderr,"Von t_look() kein Ereignis zurückgegeben\n");
   exit(10);

case T_LISTEN:
   /*
    * freies Element im Aufruf-Bereich finden
    */
   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() von t_call-Struktur gescheitert");
      exit(11);
   }

   if (t_listen(fd, calls[slot][i]) < 0) {
      t_error("t_listen() gescheitert");
      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() gescheitert");
      exit(13);
   }
```

```
        /*
         * call ind im Bereich finden und löschen
         */
        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() gescheitert");
           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() gescheitert");
           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; /* Zeiger auf Log-Datei */
   char buf[1024];
   switch (fork()) {

   case -1:
      perror("fork() gescheitert");
      exit(20);
      break;

   default: /* Vaterprozess */

      /* conn_fd schliessen und wieder überwachen */
      if (t_close(conn_fd) < 0) {
         t_error("t_close() gescheitert für conn_fd");
         exit(21);
      }
      return;

   case 0:  /* Kindprozess */

      /* listen_fd schliessen und Dienst ausführen */
      if (t_close(listen_fd) < 0) {
         t_error("t_close() gescheitert für listen_fd");
         exit(22);
      }
      if ((logfp = fopen(FILENAME, "r")) == NULL) {
         perror("Logfile kann nicht geöffnet werden");
         exit(23);
      }

      if (t_look(conn_fd) != 0) { /* gibt es schon disconnect? */
         fprintf(stderr, "t_look: unerwartetes Ereignis\n");
         exit(25);
      }

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

```
            if (t_sndrel(conn_fd) < 0) {
               t_error("t_sndrel() gescheitert");
               exit(27);
            }
            while(t_look(conn_fd) == 0) {
               sleep(1);
            }

            if(t_look(conn_fd) == T_DISCONNECT) {
               fprintf(stderr, "Verbindung abgebrochen\n");
               exit(12);
            }
            exit(0);
         }
      }
```

10 XTI-Trace

Mit dem XTI-Trace können Sie Trace-Informationen über die einzelnen XTI-Aufrufe eines Kommunikationsprozesses erzeugen.

Den XTI-Trace können Sie mit Hilfe der Umgebungsvariablen XTITRACE steuern.
Die Variable XTITRACE können Sie verwenden,

- um den XTI-Trace einzuschalten,
- um festzulegen, welche Informationen gesammelt werden sollen.

Alternativ können Sie den XTI-Trace zur Laufzeit des Programms mit der XTI-Funktion *t_optmgmt()* aktivieren. Die Funktion *t_optmgmt()* ist beschrieben im Abschnitt „t_optmgmt() - Transportendpunkt-Optionen verwalten“ auf Seite 272.

Im Standardfall werden die protokollierten Trace-Informationen im Verzeichnis für temporäre Dateien gesichert. Mit dem Programm *xtitrace* können Sie diese Dateien auswerten und sich die Trace-Informationen ausgeben lassen. Den Umfang der Auswertungen können Sie durch Angabe spezieller Optionen beim Aufruf von *xtitrace* bestimmen.

In den nachfolgenden Abschnitten ist beschrieben,

- wie Sie die Umgebungsvariable XTITRACE parametrisieren, um die gewünschten Trace-Informationen zu protokollieren.
- wie Sie mit dem Programm *xtitrace* die protokollierten Trace-Informationen ausgeben.

10.1 Umgebungsvariable XTITRACE parametrisieren

Der erste XTI-Aufruf eines Prozesses wertet die Umgebungsvariable XTITRACE aus und schaltet ggf. den XTI-Trace ein. Nach dem Einschalten des Trace wird im gewünschten Verzeichnis (Option -f *dir*, siehe nächste Seite) die temporäre Trace-Datei XTIF*pid* geöffnet, sofern sie nicht bereits geöffnet ist (*pid* spezifiziert die Prozessnummer). In diese Datei werden die Trace-Daten geschrieben.

Wenn die Datei XTIF*pid* keine weiteren Trace-Daten mehr aufnehmen kann, werden nachfolgende Trace-Daten in die Datei XTIS*pid* geschrieben. Diese Datei hat dieselbe Funktion wie die Datei XTIF*pid*. Wenn auch die Datei XTIS*pid* voll geschrieben ist, wird XTIF*pid* bereinigt und anschließend mit neuen Trace-Daten beschrieben. Der Trace-Mechanismus kann ggf. mehrmals zwischen den Dateien XTIF*pid* und XTIS*pid* hin-und herschalten. Bei jedem Dateiwechsel sollte der Prozess jeweils den Inhalt der soeben mit Trace-Daten beschriebenen temporären Datei in eine permanente Datei sichern. Auf diese Weise werden die protokollierten Trace-Daten nicht überschrieben und können später mit dem Programm *xtitrace* ausgegeben werden.

Für die Trace-Dateien XTIF*pid* und XTIS*pid* werden die Zugriffsrechte rw------- (0600) vergeben und können unter der Benutzerkennung des Prozesses eingesehen werden. Für die Pufferung der Trace-Daten wird dynamisch Speicherplatz zugeordnet. Dieser Speicherplatz und die Dateien XTIF*pid* und XTIS*pid* bleiben für die Dauer des Prozesses zugeordnet.

Die für XTITRACE spezifizierten Optionen kontrollieren den Trace-Mechanismus:

– Die Optionen *s* und *S* legen den Umfang der zu protokollierenden Informationen fest.
– Die Option *r* steuert das zyklische Überschreiben der Dateien XTIF*pid* und XTIS*pid.*
– Die Option *f* kontrolliert den Speicherplatz für die Dateien XTIF*pid* und XTIS*pid.*

Die Umgebungsvariable XTITRACE parametrisieren Sie mit folgenden Anweisungen:

```
XTITRACE="-option [ -r wrap][ -f dir]";
export XTITRACE;
```

-option

option bestimmt den Trace-Typ. Die Aktivierung eines Trace erfordert die Angabe eines Option-Wertes für *option*.

Die folgenden beiden Werte können Sie für *option* angeben:

– s
Protokolliert werden die Funktionsnamen der XTI-Aufrufe sowie die zugehörigen Parameter- und Returnwerte. Im Fehlerfall werden die Werte von *t_errno* und *errno* sowie die Fehlerposition *errpos* protokolliert.

– S
Protokolliert werden alle Informationen, die auch bei Angabe von *s* protokolliert werden. Falls Parameter vorkommen, die als Zeiger übergeben werden, werden zusätzlich die Werte der durch die Zeiger adressierten Objekte protokolliert.
Die Angabe *S* ist der Angabe *s* vorzuziehen.

-r *wrap*
Für *wrap* geben Sie eine Dezimalzahl an.
wrap legt fest, dass die Trace-Datei nach *wrap* * BUFSIZ Bytes gewechselt wird: Nach jeweils *wrap* * BUFSIZ protokollierten Bytes schaltet der Trace-Mechanismus von der Datei XTIF*pid* um auf die Datei XTIS*pid* und umgekehrt. Dabei wird der Inhalt der jeweils neu zugeschalteten Datei überschrieben. Die Konstante BUFSIZ ist definiert in <stdio.h>.

Standardwert für *wrap*: 512

-f *dir*
Mit *dir* spezifizieren Sie das Dateiverzeichnis, in dem die Trace-Dateien XTIF*pid* und XTIS*pid* angelegt werden.

Standardwert für *dir*: Standard-Dateiverzeichnis */usr/tmp*

10.2 Trace-Informationen mit dem Programm xtitrace ausgeben

Das Programm *xtitrace* liest die vom XTI-Trace erzeugten Trace-Informationen aus einer oder mehreren Dateien. *xtitrace* verarbeitet diese Trace-Informationen gemäß den für *xtitrace* spezifizierten Optionen und gibt das Ergebnis auf die Standardausgabe aus.

Nach erfolgreicher Ausführung von *xtitraxce* ist der Status 0, andernfalls ungleich 0.

Programm *xtitrace* aufrufen

Das Programm *xtitrace* rufen Sie wie folgt auf:

```
xtitrace[ -option] file ...
```

-option

Mit *option* legen Sie fest, welche Informationen der durch *file* ... spezifizierte(n) Trace-Datei(en) auszugeben sind. Pro *xtitrace*-Aufruf können Sie für *option* mehrere der nachfolgend beschriebenen Werte angeben.

- c
  Ausgegeben werden die Trace-Informationen zu XTI-Aufrufen für folgende Aktionen:
  - Installieren/Deinstallieren einer Kommunikationsanwendung
  - Einrichten/Abbauen einer Verbindung

  Betroffen sind die XTI-Funktionen *t_accept()*, *t_bind()*, *t_close()*, *t_connect()*, *t_listen()*, *t_open()*, *t_rcvconnect()*, *t_rcvdis()*, *t_rcvrel()*, *t_snddis()*, *t_sndrel()* und *t_unbind()*.
- d
  Ausgegeben werden die Trace-Informationen zu XTI-Aufrufen für den Datenaustausch.
  Betroffen sind die XTI-Funktionen *t_rcv()*, *t_rcvudata()*, *t_rcvuderr()*, *t_snd()* und *t_sndudata()*.
- m
  Ausgegeben werden die Trace-Informationen zu XTI-Aufrufen, die bei den Optionen *c* und *d* nicht berücksichtigt sind.
  Betroffen sind die XTI-Funktionen *t_alloc()*, *t_error()*, *t_free()*, *t_getinfo()*, *t_getstate()*, *t_look()*, *t_optmgmt()* und *t_sync()*.

– v
  Ausgegeben werden die Trace-Informationen zu allen XTI-Aufrufen sowie die Werte der zugehörigen Parameter und Optionen. Falls Parameter vorkommen, die als Zeiger übergeben werden, werden die Werte der durch die Zeiger adressierten Objekte ebenfalls ausgegeben. Letzeres setzt allerdings voraus, dass die entsprechenden Daten bei der Trace-Erstellung aufgezeichnet wurden (siehe vorhergehender Abschnitt „Umgebungsvariable XTITRACE parametrisieren“ auf Seite 228). Wenn XTITRACE mit Option *S* parametrisiert wurde, sollten Sie für *option* den Wert *v* spezifizieren. Die Angabe *v* hat dieselbe Bedeutung wie die Angabe *cdmv*.

file ...
  Mit *file* spezifizieren Sie den Namen einer Datei, die binäre Trace-Daten enthält. Sie können auch mehrere Dateinamen angeben.

Ausgabeformat des XTI-Trace

Das Programm *xtitrace* beginnt die Ausgabe immer mit einem Header. Dahinter schreibt *xtitrace* die Trace-Informationen zu den einzelnen XTI-Aufrufen. Je nach Parametrisierung der Umgebungsvariablen XTITRACE und des Programms *xtitrace* gibt *xtitrace* für jeden protokollierten XTI-Aufruf entweder eine einzelne Zeile oder mehrere Zeilen unterschiedlichen Formats aus.

Format des Headers

Der Header enthält folgende Informationen:

– Versionsnummer der XTI-Bibliothek
– Datum und Uhrzeit des Trace-Beginns
– spezifizierte Werte der Ausgabeoption für *xtitrace*
– Name(n) der Trace-Datei(en), deren Inhalt *xtitrace* ausgibt

Beispiel

```
XTI    TRACE (Vx.x)              Mon Aug 11 15:13:34 1997
OPTIONS 'cdmv' , TRACE FILE 'XTIF00963'
```

Format der ersten Ausgabezeile für einen protokollierten XTI-Aufruf

Die Trace-Information für einen XTI-Aufruf beginnt immer mit einer Zeile, die wie folgt aufgebaut ist:

- Am Anfang der Zeile steht der Zeitstempel:

  minutes:*seconds*.*milliseconds* (z.B. 24:16.324)

  Die Genauigkeit in Millisekunden hängt von der verwendeten Hardware ab.

- An den Zeitstempel schließt sich der aufgezeichnete XTI-Aufruf (z.B. *t_bind()*) an. Dahinter folgt eine in runde Klammern eingeschlossene Liste, in der die Parameter und zugehörigen Parameterwerte des betreffenden XTI-Aufrufs aufgeführt sind (in der von XTI geforderten Reihenfolge). Die Parameterwerte sind in dezimaler (%d), sedezimaler (0x%x) oder symbolischer Form (%s) dargestellt. Ein sedezimal dargestellter Parameterwert beginnt immer mit 0x.

  Im Einzelnen gilt für die Darstellung von Parametern und zugehörigen Werten:

  – Werte von Zeigern sind sedezimal dargestellt.

  – Bei Parametern vom Typ Integer (z.B. *fd*) kann der korrespondierende Wert in sedezimaler, dezimaler oder symbolischer Form dargestellt sein. Parameter und zugehöriger Wert sind durch ein Leerzeichen getrennt.

- Bei XTI-Funktionen, deren Ausführung vom Zustand des Transportendpunkts abhängt, informiert das Trace-Protokoll, ob der Aufruf blockiert (Standardfall) oder nicht (Spezifikation: O_NDELAY bzw. O_NONBLOCK) .

Format weiterer Ausgabezeilen für einen protokollierten XTI-Aufruf

Die nachfolgend erläuterten Trace-Information gibt *xtitrace* nur aus, wenn die beiden folgenden Bedingungen erfüllt sind:

– Für die Trace-Erstellung wurde die Variable XTITRACE mit Option *S* parametrisiert.
– Das Programm *xtitrace* wurde mit Option *v* parametrisiert.

Für Parameter, die als Zeiger übergeben werden, gibt *xtitrace* dann Namen und Werte der von diesen Zeigern adressierten Datenobjekte aus. Die Werte dieser Datenobjekte (z.B. Strukturkomponenten) werden sedezimal ausgegeben. Die Namenskonventionen für Parameter und Strukturkomponenten entsprechen den im Kapitel „Bibliotheksfunktionen von XTI(POSIX)“ auf Seite 235 verwendeten Namenskonventionen.

Die Trace-Informationen zu Strukturkomponenten enthalten einige Sonderzeichen mit folgender Bedeutung:

> Der betreffenden Komponente muss vor Aufruf der protokollierten XTI-Funktion von der aufrufenden Kommunikationsanwendung ein Wert zugewiesen werden.

< In der betreffenden Komponente wird von der protokollierten XTI-Funktion ein Wert zurückgeliefert, falls die XTI-Funktion ordnungsgemäß ausgeführt wird.

\- Der Wert der betreffenden Komponente ist für die protokollierte XTI-Funktion ohne Bedeutung.

Wenn „---“ an Stelle eines Komponentenwerts ausgegeben wird, ist der betreffenden Komponente kein Wert zugeordnet.

Format der letzten Ausgabezeile für einen protokollierten XTI-Aufruf

In der letzten Zeile für einen protokollierten XTI-Aufruf wird immer der Returnwert der betreffenen XTI-Funktion ausgegeben. Im Fehlerfall werden *t_errno*, evtl. *errno* sowie Angaben zur Fehlerposition (*errpos*) ausgegeben.

Beispiel für eine ausführliche Protokollierung eines XTI-Aufrufs

```
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 Bibliotheksfunktionen von XTI(POSIX)

In diesem Kapitel sind die Bibliotheksfunktionen von XTI(POSIX) beschrieben.

Zunächst wird das Format vorgestellt, in dem die einzelnen XTI-Funktionen beschrieben sind. Die darauf folgende Übersicht fasst jeweils mehrere XTI-Funktionen unter aufgabenorientierten Gesichtspunkten zusammen. Im Anschluss daran sind alle XTI-Funktionen in alphabetischer Reihenfolge beschrieben.

11.1 Beschreibungsformat

Die XTI-Funktionen sind in einem einheitlichen Format beschrieben. Die Beschreibung der Funktionen ist wie folgt aufgebaut:

Funktionsname - Kurzbeschreibung der Funktionalität

```
#include < ... >
Syntax der Funktion
```

Beschreibung

Ausführliche Beschreibung der Funktionalität und Erklärung der Parameter.

Returnwert

Auflistung und Beschreibung möglicher Returnwerte der Funktion.

Fehler

Aufzählung und Beschreibung der Fehlercodes, die bei einem fehlerhaften Aufruf oder Ablauf der Funktion in *t_errno* abgelegt werden.

Hinweis

Begriffserklärungen oder Informationen über das Zusammenwirken mit anderen Funktionen oder Tipps für die Anwendung. Dieser Abschnitt kann fehlen.

Siehe auch

Querverweise auf die Beschreibung anderer Funktionen.

11.2 Übersicht über die Funktionen

Die folgende Übersicht über die Bibliotheksfunktionen von XTI fasst jeweils mehrere Funktionen unter aufgabenorientierten Gesichtspunkten zusammen.

Verbindungen über Transportendpunkte aufbauen und beenden

Funktion	Beschreibung	siehe
t_open()	Transportendpunkt einrichten	Seite 268
t_close()	Transportendpunkt schließen	Seite 249
t_bind()	einem Transportendpunkt eine Adresse zuordnen	Seite 246
t_unbind()	Transportendpunkt deaktivieren	Seite 304
t_connect()	Verbindung über einen Transportendpunkt initiieren (z.B. durch einen Client)	Seite 250
t_rcvconnect()	Status einer zuvor abgeschickten Verbindungsanforderung abfragen	Seite 280
t_listen()	Transportendpunkt auf anstehende Verbindungsanforderungen überprüfen (z.B. durch einen Server)	Seite 264
t_accept()	Verbindung über einen Transportendpunkt annehmen (z.B. durch einen Server)	Seite 241
t_rcvrel()	Empfang eines Wunsches nach geordnetem Verbindungsabbau bestätigen	Seite 285
t_rcvdis()	Ursache eines Verbindungsabbaus abfragen	Seite 283
t_sndrel()	geordneten Verbindungsabbau einleiten	Seite 296
t_snddis()	Verbindungsanforderung zurückweisen oder sofortigen Abbruch einer bereits bestehenden Verbindung einleiten	Seite 294

Daten zwischen Transportendpunkten übertragen

Funktion	Beschreibung	siehe
t_rcv()	Daten über einen Transportendpunkt empfangen (verbindungsorientiert)	Seite 278
t_rcvudata()	Datagramme über einen Transportendpunkt empfangen (verbindungslos)	Seite 287
t_rcvuderr()	Fehlerinformationen über ein gesendetes Datagramm empfangen (verbindungslos)	Seite 290
t_snd()	Daten über einen Transportendpunkt senden (verbindungsorientiert)	Seite 292
t_sndudata()	Datagramme über einen Transportendpunkt senden (verbindungslos)	Seite 298

Informationen über Transportendpunkte erhalten

Funktion	Beschreibung	siehe
t_getinfo()	protokollspezifische Informationen abfragen	Seite 257
t_getstate()	aktuellen Zustand des Transportanbieters abfragen	Seite 262
t_getprotaddr()	Protokolladressen abfragen	Seite 260
t_look()	vom Transportanbieter gemeldetes aktuelles Ereignis auf dem Transportendpunkt abfragen	Seite 266

Optionen eines Transportendpunkts verwalten

Funktion	Beschreibung	siehe
t_optmgmt()	Optionen eines Transportendpunkts verwalten	Seite 272

Datenstrukturen der Transportbibliothek verwenden

Funktion	Beschreibung	siehe
t_alloc()	für in der Transportbibliothek <xti.h> deklarierte Datenstrukturen dynamisch Speicher reservieren	Seite 244
t_free()	für in der Transportbibliothek <xti.h> deklarierte Datenstrukturen reservierten Speicher freigeben	Seite 255
t_sync()	Datenstrukturen der Transportbibliothek <xti.h> synchronisieren	Seite 302

Fehlermeldungen generieren

Funktion	Beschreibung	siehe
t_error()	Fehlermeldung auf Standardausgabe ausgeben	Seite 253
t_strerror()	Fehlermeldungstext ausgeben	Seite 301

11.3 Beschreibung der Funktionen

In diesem Abschnitt sind die Bibliotheksfunktionen von XTI(POSIX) in alphabetischer Reihenfolge beschrieben.

Um die XTI-Funktionen ausführen zu können, muss die Anwendung die X/Open-konforme Include-Datei <xti.h> einbinden. Die Datei <xti.h> wird bei der Installation von SOCKETS(POSIX) in das Verzeichnis */usr/include* kopiert (siehe auch Abschnitt „Include-Dateien“ auf Seite 9).

Liefert eine XTI-Funktion den Fehler TSYSERR, so wird die Fehlervariable *errno* gesetzt. Die Werte für *errno* sind in <errno.h> definiert.

t_accept() - Verbindung annehmen

```
#include <xti.h>
int t_accept(int fd, int resfd, struct t_call *call);
```

Beschreibung

Mit der Funktion *t_accept()* nimmt der Transportbenutzer über einen Transportendpunkt eine Verbindung an, die von einem anderen Transportbenutzer mit der Funktion *t_connect()* angefordert wurde.

Der Parameter *fd* bezeichnet den lokalen Transportendpunkt, auf dem eine Verbindungsanforderung eingetroffen ist. Der Parameter *resfd* spezifiziert den lokalen Transportendpunkt, über den die Verbindung hergestellt werden soll.

Beim Transportendpunkt *resfd*, auf dem die Verbindung angenommen werden soll, sind zwei Fälle zu unterscheiden:

– *resfd* == *fd*
  In diesem Fall dürfen auf *fd* keine weiteren Verbindungsanforderungen anstehen, d.h. der Transportbenutzer muss alle zuvor auf *fd* eingegangenen Verbindungsanforderungen mit *t_accept()* oder *t_snddis()* bearbeitet haben. Andernfalls beendet sich *t_accept()* mit Fehler und setzt *t_errno* auf TINDOUT.

– *resfd* != *fd*
  In diesem Fall muss sich *resfd* beim Aufruf von *t_accept()* im Zustand T_UNBND oder T_IDLE befinden (siehe Abschnitt „t_getstate() - Aktuellen Zustand abfragen“ auf Seite 262).

Mit dem Parameter *call* übergibt der Benutzer Informationen, die der Transportanbieter für die Einrichtung der Verbindung benötigt. *call* ist ein Zeiger auf ein Objekt vom Typ *struct t_call*.

Die Struktur *t_call* ist in <xti.h> wie folgt deklariert:

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

call->addr enthält die Protokolladresse des Transportbenutzers, der die Verbindungsanforderung gesendet hat.

call->opt zeigt alle die Verbindung betreffenden Optionen an. Werte und Syntax dieser Optionen sind protokollspezifisch.
Das Senden von Benutzerdaten (Parameter *call->udata*) wird nicht unterstützt.
call->sequence enthält den zuvor von *t_listen()* zurückgelieferten Wert, der die auf dem Transportendpunkt *fd* anstehende Verbindungsanforderung eindeutig identifiziert.

Wenn für den durch *fd* übergebenen Transportendpunkt noch weitere Ereignisse anstehen (Verbindungsanforderung oder Verbindungsabbruch-Wunsch), beendet sich *t_accept()* mit Fehler und setzt *t_errno* auf TLOOK.

Returnwert

0:
bei Erfolg.

-1:
bei Fehler. *t_ errno* wird gesetzt, um den Fehler anzuzeigen.

Fehler

TBADF
Der angegebene Deskriptor verweist nicht auf einen Transportendpunkt.

TOUTSTATE
Innerhalb einer Sequenz von XTI-Funktionsaufrufen für den Transportendpunkt *fd* wurde *t_accept()* an der falschen Stelle aufgerufen, oder der durch *resfd* übergebene Transportendpunkt befindet sich nicht im Zustand T_IDLE oder T_UNBND.

TACCES
Der Benutzer hat keine Erlaubnis, eine Verbindung auf dem antwortenden Transportendpunkt anzunehmen oder die angegebenen Optionen zu verwenden.

TBADDATA
Das Senden von Benutzerdaten wird nicht unterstützt.

TBADOPT
Die spezifizierten Optionen hatten ein falsches Format oder enthielten ungültige Informationen.

TBADSEQ
Es wurde eine ungültige Sequenznummer angegeben.

TINDOUT
Die Funktion wurde mit *fd == resfd* aufgerufen, und für den durch *fd* übergebenen Transportendpunkt stehen noch weitere Verbindungsanforderungen an. Diese zuvor eingegangenen Verbindungsanforderungen müssen zuerst mit *t_accept()* oder *t_snddis()* bearbeitet werden.

TLOOK
Auf dem durch *fd* übergebenen Transportendpunkt ist ein asynchrones Ereignis eingetreten und erfordert sofortige Bearbeitung.

TNOTSUPPORT
Die Funktion wird vom darunter liegenden Transportdienst nicht unterstützt.

TRESQLEN
Dem durch *resfd* übergebenen Transportendpunkt (mit *resfd* != *fd*) ist eine Protokolladresse zugeordnet, für die *qlen* > 0 gilt.

TSYSERR
Während der Ausführung dieser Funktion ist ein Systemfehler aufgetreten.

Siehe auch

t_connect(), t_getstate(), t_listen(), t_open(), t_rcvconnect()

t_alloc() - Speicher für Bibliotheksstruktur anlegen

```
#include <xti.h>
char *t_alloc(int fd, int struct_type, int fields);
```

Beschreibung

Mit der Funktion *t_alloc()* reserviert der Transportbenutzer dynamisch Speicher für verschiedene Typen von Strukturen. *t_alloc()* liefert einen Zeiger auf das neu angelegte Strukturobjekt zurück. Jedes mit *t_alloc()* erzeugte Strukturobjekt kann beim Aufruf bestimmter XTI-Funktionen als aktueller Parameter übergeben werden.

Als aktuellen Parameter für *fd* muss der Benutzer den Transportendpunkt spezifizieren, über den das mit *t_alloc()* erzeugte Strukturobjekt beim Aufruf einer XTI-Funktion (z.B. *t_bind()*) übergeben wird. Auf diese Weise kann *t_alloc()* auf die geeignete Größeninformation zugreifen. Die Größe des angelegten Puffers resultiert nämlich aus derselben Information, die der Benutzer mit *t_open()* und *t_getinfo()* für den betreffenden Transportendpunkt erhält.

Der Parameter *struct_type* spezifiziert den Strukturtyp. *t_alloc()* reserviert dann Speicher für die betreffende Struktur sowie für Puffer, auf die diese Struktur verweist.
Für *struct_type* kann der Benutzer beim Aufruf von *t_alloc()* folgende Werte angeben:

– T_BIND (für *struct t_bind*)
– T_CALL (für *struct t_call*)
– T_OPTMGMT (für *struct t_optmgmt*)
– T_DIS (für *struct t_discon*)
– T_UNITDATA (für *struct t_unitdata*)
– T_UDERROR (für *struct t_uderr*)
– T_INFO (für *struct t_info*)

Mit Ausnahme der Struktur *t_info* enthält jede der oben genannten Strukturen zumindest eine Komponente vom Typ *struct netbuf*.

Die Struktur *netbuf* ist in <xti.h> wie folgt deklariert:

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

Mit dem Parameter *fields* spezifiziert der Benutzer für jede *netbuf*-Struktur in der durch *struct_type* spezifizierten Struktur, ob für den Puffer dieser *netbuf*-Struktur ebenfalls Speicher reserviert werden soll. *fields* wird gebildet, indem beliebige der nachfolgend beschriebenen Werte durch bitweises inklusives ODER verknüpft werden:

– T_ADDR: Komponente *addr* der Strukturen *t_bind*, *t_call*, *t_unitdata* oder *t_uderr*

– T_OPT: Komponente *opt* der Strukturen *t_optmgmt*, *t_call*, *t_unitdata* oder *t_uderr*

– T_UDATA: Komponente *udata* der Strukturen *t_call*, *t_discon* oder *t_unitdata*

– T_ALL: alle relevanten Komponenten der durch *struct_type* spezifizierten Struktur

Für jede durch den Parameter *fields* spezifizierte *netbuf*-Struktur reserviert *t_alloc()* Speicher für den dieser Struktur zugeordneten Puffer. Außerdem initialisiert *t_alloc()* in den einzelnen *netbuf*-Strukturen jeweils den Zeiger *buf* und den Wert von *maxlen* entsprechend.

Wenn der Wert von *maxlen* in irgendeiner durch *fields* spezifizierten *netbuf*-Struktur den Wert -1 oder -2 hat (siehe *t_open()* oder *t_getinfo()*), kann *t_alloc()* die Größe des Puffers nicht ermitteln und beendet sich mit Fehler. Dabei wird *t_errno* auf TSYSERR und *errno* auf EINVAL gesetzt. Für jede nicht in *fields* spezifizierte *netbuf*-Struktur wird *buf* auf NULL und *maxlen* auf 0 gesetzt.

Returnwert

Bei erfolgreicher Ausführung liefert *t_alloc()* einen Zeiger auf die neu angelegte Struktur zurück.
Bei Fehler wird der Null-Zeiger zurückgeliefert. *t_ errno* wird gesetzt, um den Fehler anzuzeigen.

Fehler

TBADF
: Der angegebene Dateideskriptor verweist nicht auf einen Transportendpunkt.

TSYSERR
: Während der Ausführung dieser Funktion ist ein Systemfehler aufgetreten.

Siehe auch

t_free(), t_getinfo(), t_open()

t_bind() - Einem Transportendpunkt eine Adresse zuordnen

```
#include <xti.h>
int t_bind(int fd, struct t_bind *req, struct t_bind *ret);
```

Beschreibung

Mit der Funktion *t_bind()* ordnet der Benutzer dem durch den Parameter *fd* spezifizierten Transportendpunkt eine Protokolladresse zu und aktiviert diesen Transportendpunkt.

Nach erfolgreicher Ausführung von *t_bind()* hat der Benutzer folgende Möglichkeiten:

– Im verbindungsorientierten Modus kann der Benutzer den durch *fd* spezifizierten Transportendpunkt mit *t_listen()* auf anstehende Verbindungsanforderungen überprüfen; anschließend kann der Benutzer auf *fd* ggf. Verbindungen mit *t_accept()* annehmen. Ebenso kann der Benutzer über den Transportendpunkt *fd* mit *t_connect()* Verbindungsanforderungen an andere Transportendpunkte schicken.

– Im verbindungslosen Modus kann der Benutzer über den durch *fd* spezifizierten Transportendpunkt Datagramme versenden oder empfangen.

Die Parameter *req* und *ret* zeigen jeweils auf ein Objekt vom Typ *struct t_bind*.

Die Struktur *t_bind* ist in <xti.h> wie folgt deklariert:

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

In *req->addr* spezifiziert der Benutzer die Protokolladresse, die dem Transportendpunkt zugeordnet werden soll. In *req->addr.len* spezifiziert der Benutzer die Länge (in Bytes) dieser Adresse. *req->addr.buf* zeigt auf den Adresspuffer. *req->addr.maxlen* ist ohne Bedeutung.

In *ret->addr.buf* übergibt der Transportbenutzer einen Zeiger auf einen Puffer; die maximale Länge dieses Puffers spezifiziert der Benutzer in *ret->addr.maxlen*. Nach erfolgreicher Ausführung liefert *t_bind()* in *ret-> addr.buf* die dem Transportendpunkt *fd* zugeordnete Adresse zurück. Die tatsächliche Länge dieser Adresse liefert *t_bind()* in *ret->addr.len* zurück.

Wenn die in *ret->addr.maxlen* angegebene Länge zu klein ist für die Aufnahme der von *t_bind()* zurückgelieferten Adresse, liefert *t_bind()* den Fehlercode TBUFOVFLW zurück. Der Zustand des Transportendpunkts ändert sich jedoch auf T_IDLE.

req->qlen und *ret->qlen* sind nur von Bedeutung, wenn *fd* im verbindungsorientierten Modus betrieben wird. Dort wird durch *req->qlen* und *ret->qlen* die Maximalzahl anstehender Verbindungsanforderungen festgelegt, die der Transportanbieter für den Transportendpunkt *fd* unterstützt. Eine anstehende Verbindungsanforderung ist eine vom Transportsystem an einen Transportendpunkt des Benutzers übergebene Verbindungsanforderung, die von diesem Benutzer bislang weder akzeptiert (*t_accept()*) noch zurückgewiesen (*t_snddis()*) wurde. Die Anzahl der vom Transportanbieter unterstützten Verbindungsanforderungen für den Transportendpunkt *fd* ergibt sich wie folgt:

– Vor Aufruf von *t_bind()* spezifiziert der Benutzer in *req->qlen* die Anzahl anstehender Verbindungsanforderungen, die der Transportanbieter auf dem Transportendpunkt *fd* untersützen soll. *req->qlen* > 0 ist nur bei einem Transportendpunkt sinnvoll, den der Benutzer später mit *t_listen()* passiv auf anstehende Verbindungsanforderungen „abhört“.

– *req->qlen* wird vom Transportanbieter ausgewertet. Falls der Transportanbieter die in *req->qlen* spezifizierte Anzahl anstehender Verbindungsanforderungen nicht unterstützen kann, setzt er den in *req->qlen* übergebenen Wert entsprechend herab. Niemals setzt der Transportanbieter jedoch einen *req->qlen*-Wert > 0 auf 0. Derzeit kann der Transportanbieter maximal 8 anstehende Verbindungsanforderungen unterstützen.

– In *ret->qlen* liefert *t_bind()* die Anzahl anstehender Verbindungsanforderungen zurück, die der Transportanbieter für den Transportendpunkt *fd* tatsächlich unterstützt.

Wenn der Benutzer die zu bindende, also dem Transportendpunkt *fd* zuzuordnende Adresse nicht selbst spezifizieren möchte, übergibt er als aktuellen Parameter für *req* den Null-Zeiger. In diesem Fall wählt der Transportanbieter die zu bindende Adresse aus, wobei er für *req->qlen* implizit den Wert 0 voraussetzt.
Ebenso kann der Benutzer als aktuellen Parameter für *ret* den Null-Zeiger übergeben, wenn es ihm gleichgültig ist, welche Adresse bei *t_bind()* vom Transportanbieter an *fd* gebunden wird und welchen Wert *qlen* hat.
Es ist zulässig, im selben *t_bind()*-Aufruf sowohl für *req* als auch für *ret* jeweils den Null-Zeiger zu übergeben. Dann wählt der Transportanbieter die Adresse aus, die an *fd* gebunden wird; *t_bind()* liefert diese Information jedoch nicht an den Benutzer zurück.

Returnwert

0:
bei Erfolg.

-1:
bei Fehler. *t_ errno* wird gesetzt, um den Fehler anzuzeigen.

Fehler

TACCES
: Der Benutzer hat keine Erlaubnis, die spezifizierte Adresse zu benutzen.

TADDRBUSY
: Die spezifizierte Protokolladresse wird schon benutzt.

TBADADDR
: Die spezifizierte Protokolladresse hatte ein falsches Format oder enthielt falsche Informationen.

TBADF
: Der angegebene Dateideskriptor verweist nicht auf einen Transportendpunkt.

TBUFOVFLW
: Die erlaubte Anzahl Bytes für einen Ergebnisparameter ist zu klein, um den Wert des Parameters zu speichern. Der Zustand des Transportanbieters wird in T_IDLE umgeändert, und die Information, die in **ret* zurückgeliefert werden soll, wird gelöscht.

TNOADDR
: Der Transportanbieter konnte keine Adresse reservieren (siehe auch Abschnitt „Abhängigkeiten vom BS2000/OSD-Transportsystem BCAM“ auf Seite 316).

TOUTSTATE
: Innerhalb einer Sequenz von XTI-Funktionsaufrufen für den Transportendpunkt *fd* wurde die Funktion an der falschen Stelle aufgerufen.

TSYSERR
: Während der Ausführung dieser Funktion ist ein Systemfehler aufgetreten.

Siehe auch

t_open(), t_optmgmt(), t_unbind()

t_close() - Transportendpunkt schließen

```
#include <xti.h>
int t_close(int fd);
```

Beschreibung

Mit der Funktion *t_close()* informiert der Benutzer den Transportanbieter, dass er den durch *fd* spezifizierten Transportendpunkt nicht mehr benötigt. *t_close()* gibt alle für *fd* reservierten lokalen Bibliotheks-Betriebsmittel frei.

t_close() sollte im Zustand T_UNBND aufgerufen werden (siehe Abschnitt „t_getstate() - Aktuellen Zustand abfragen“ auf Seite 262). Da *t_close()* keine Zustandsinformationen überprüft, kann *t_close()* aber auch in jedem anderen Zustand aufgerufen werden, um einen Transportendpunkt zu schließen.

Wenn es im aufrufenden oder einem anderen Prozess keinen weiteren Deskriptor für den Transportendpunkt *fd* gibt, wird der Tranportendpunkt vollständig abgebaut, d.h. die System-Ressourcen werden freigegeben. Noch bestehende Verbindungen werden abgebrochen. Dabei gehen Daten verloren, die noch nicht gesendet oder vom Empfänger abgeholt wurden.

Returnwert

0:
bei Erfolg.

-1:
bei Fehler. *t_ errno* wird gesetzt, um den Fehler anzuzeigen.

Fehler

TBADF
Der angegebene Dateideskriptor verweist nicht auf einen Transportendpunkt.

Siehe auch

t_getstate(), t_open(), t_unbind()

t_connect() - Verbindung anfordern

```
#include <xti.h>
int t_connect(int fd, struct t_call *sndcall, struct t_call *rcvcall);
```

Beschreibung

Mit der Funktion *t_connect()* sendet der Transportbenutzer über den lokalen Transportendpunkt *fd* eine Verbindungsanforderung an einen anderen Transportbenutzer, der durch die im Parameter *sndcall* übergebene Protokolladresse spezifiziert ist.

Die Parameter *sndcall* und *rcvcall* zeigen auf jeweils ein Objekt vom Typ *struct t_call*.

Die Struktur *t_call* ist in <xti.h> wie folgt deklariert:

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

In *sndcall* übergibt der Aufrufer von *t_connect()* Informationen, die der Transportanbieter benötigt, um eine Verbindungsanforderung zu senden:

– *sndcall->addr* enthält die Protokolladresse des Transportendpunkts, an den die Verbindungsanforderung gesendet werden soll.

– *sndcall->opt* enthält protokollspezifische Informationen, die der Transportanbieter benötigt. *sndcall->opt* spezifiziert aber **nicht** die Struktur der Optionen, da der Transportanbieter die Struktur aller an ihn übergebenen Optionen selbst festlegt. Diese Optionen sind spezifisch für das darunter liegende Protokoll des Transportanbieters.
Wenn der Benutzer in *sndcall->opt.len* den Wert 0 übergibt, wählt der Transportanbieter Standardoptionen aus und der Benutzer muss die Optionen nicht mit dem Transportanbieter aushandeln.

Da das Senden von Benutzerdaten nicht unterstützt wird, ist *sndcall->udata* für *t_connect()* ohne Bedeutung. *sndcall->sequence* ist ebenfalls ohne Bedeutung für *t_connect()*.

In *rcvcall* liefert *t_connect()* bei erfolgreicher Ausführung Informationen über die soeben eingerichtete Verbindung zurück:

– *rcvcall->addr* enthält die Protokolladresse des Transportendpunkts, der die Verbindungsanforderung mit *t_accept()* angenommen hat.
Vor Aufruf von *t_connect()* muss der Benutzer in *rcval->addr.maxlen* die maximale Länge des Ergebnispuffers (*rcval->addr.buf*) bekannt geben.

– *rcval->opt* enthält protokollspezifische Informationen, die die neu eingerichtete Verbindung betreffen.
Vor Aufruf von *t_connect()* muss der Benutzer in *rcval->opt.maxlen* die maximale Länge des Ergebnispuffers (*rcval->opt.buf*) bekannt geben.

Da das Empfangen von Benutzerdaten nicht unterstützt wird, ist *rcvcall->udata* für *t_connect()* ohne Bedeutung. *rcvcall->sequence* ist ebenfalls ohne Bedeutung für *t_connect()*.

Im Standardfall arbeitet *t_connect()* im synchronen Modus. Im synchronen Modus wartet (blockiert) *t_connect()* solange, bis eine Antwort des Zielbenutzers, d.h. des Transportbenutzers, an den die Verbindungsanforderung gesendet wurde, eintrifft. Erst nach Empfang der Antwort gibt *t_connect()* die Kontrolle wieder an den aufrufenden Transportbenutzer zurück. Eine erfolgreiche Durchführung (Returnwert 0) von *t_connect()* zeigt an, dass die angeforderte Verbindung eingerichtet wurde.

Wenn jedoch zuvor mit *t_open()* oder der POSIX-Funktion *fcntl()* für den durch *fd* spezifizierten Transportendpunkt O_NDELAY oder O_NONBLOCK gesetzt wurde, wird *t_connect()* im asynchronen Modus ausgeführt. Im asynchronen Modus wartet *t_connect()* nicht auf die Antwort des Zielbenutzers, sondern gibt die Kontrolle nach Abfrage des Status der Verbindungsanforderung sofort wieder an den aufrufenden Benutzer zurück. Falls die angeforderte Verbindung noch nicht eingerichtet ist, liefert *t_connect()* den Returnwert -1 zurück und setzt *t_errno* auf TNODATA.
Im asynchronen Modus leitet *t_connect()* den Verbindungsaufbau also einfach durch Senden einer Verbindungsanforderung an den Zielbenutzer ein. Mit der Funktion *t_rcvconnect()* kann der lokale Benutzer den Status der angeforderten Verbindung abfragen.

Returnwert

0:
bei Erfolg.

-1:
bei Fehler. *t_ errno* wird gesetzt, um den Fehler anzuzeigen.

Fehler

TACCES
: Der Benutzer hat keine Erlaubnis, die angegebene Adresse oder die angegebenen Optionen zu verwenden.

TBADADDR
: Die angegebene Protokolladresse hatte ein falsches Format oder enthielt falsche Informationen.

TBADDATA
: Das Senden von Benutzerdaten wird nicht unterstützt.

TBADF
: Der angegebene Dateideskriptor verweist nicht auf einen Transportendpunkt.

TBADOPT
: Die angegebenen Protokolloptionen hatten ein falsches Format oder enthielten falsche Informationen.

TBUFOVFLW
: Die Anzahl der Bytes, die für einen Ergebnisparameter reserviert wurden, reichen nicht aus, um den Wert des Parameters zu speichern. Falls im synchronen Modus gearbeitet wird, wird der Zustand des Transportanbieters aus Benutzersicht auf den Zustand T_DATAXFER gesetzt, und die Information für die Verbindungsanforderung, die in **rcvcall* zurückgeliefert werden soll, wird entfernt.

TLOOK
: Auf dem durch *fd* übergebenen Transportendpunkt ist ein asynchrones Ereignis eingetreten und erfordert sofortige Bearbeitung.

TNODATA
: O_NDELAY oder O_NONBLOCK wurde gesetzt, sodass die Funktion den Vorgang des Verbindungsaufbaus erfolgreich einleiten konnte, jedoch nicht auf eine Antwort vom fernen Benutzer wartet.

TNOTSUPPORT
: Diese Funktion wird vom darunter liegenden Transportanbieter nicht unterstützt.

TOUTSTATE
: Innerhalb einer Sequenz von XTI-Funktionsaufrufen für den Transportendpunkt *fd* wurde die Funktion an der falschen Stelle aufgerufen.

TSYSERR
: Während der Ausführung dieser Funktion ist ein Systemfehler aufgetreten.

Siehe auch

t_accept(), t_getinfo(), t_listen(), t_open(), t_optmgmt(), t_rcvconnect(), fcntl()

t_error() - Fehlermeldung auf Standardausgabe ausgeben

```
#include <xti.h>

int t_error(char *errmsg);
extern int t_errno;
extern char *t_errlist[];
extern int t_nerr;
```

Beschreibung

Mit der Funktion *t_error()* schreibt der Benutzer eine selbst formulierte Meldung auf die Standard-Fehlerausgabe, wobei die Meldung den zuletzt aufgetretenen Fehler beim Aufruf einer XTI-Funktion beschreibt. Diese Meldung, die den Fehler im Kontext beschreibt, wird im Parameter *errmsg* übergeben.

t_errlist ist ein Vektor von Meldungen, die jeweils als Character-String dargestellt sind und die Formatierung der Benutzermeldungen ermöglichen. *t_errno* kann als Index für diesen Vektor verwendet werden, um eine bestimmte Fehlermeldung in String-Darstellung zu erhalten (ohne Zeilenendeabschluss). *t_nerr* ist der maximale Indexwert für den Vektor *t_errlist*.
t_errno wird gesetzt, wenn ein Fehler auftritt; jedoch wird *t_errno* bei nachfolgenden erfolgreichen Aufrufen nicht gelöscht.

Die Ausgabe von *t_error()* besteht aus der vom Benutzer übergebenen Fehlermeldung, gefolgt von einem Doppelpunkt und der Standard-Fehlerausgabe der XTI-Funktion für den aktuellen Wert in *t_errno*. Wenn *t_errno* den Wert TSYSERR besitzt, gibt *t_error()* die Standardfehlermeldung auch für den aktuellen Wert in *errno* aus.

Returnwert

Stets 0

Fehler

Für *t_error()* sind keine Fehlercodes definiert.

Beispiel

Wenn sich die Funktion *t_connect()* auf dem Transportendpunkt *fd2* mit Fehler beendet, weil eine ungültige Adresse angegeben wurde, kann dem Fehler der folgende Aufruf folgen:

```
t_error("t_connect failed");
```

Folgende Meldung wird ausgeben:

```
t_connect failed: incorrect addr format
```

"t_connect failed" teilt dem Benutzer mit, welche Funktion fehlgeschlagen ist. "incorrect addr format" zeigt den spezifischen Fehler an, der aufgetreten ist.

t_free() - Speicher für Bibliotheksstruktur freigeben

```
#include <xti.h>
int t_free(char *ptr, int struct_type);
```

Beschreibung

Mit der Funktion *t_free()* gibt der Benutzer Speicherplatz frei, den er zuvor mit der Funktion *t_alloc()* angelegt hat. *t_free()* gibt den Speicherplatz für das Objekt vom Typ *struct_type* frei, auf das der Zeiger *ptr* zeigt.

struct_type spezifiziert einen der sechs Strukturtypen, die bei *t_alloc()* beschrieben sind:

– T_BIND (für *struct t_bind*)

– T_CALL (für *struct t_call*)

– T_OPTMGMT (für *struct t_optmgmt*)

– T_DIS (für *struct t_discon*)

– T_UNITDATA (für *struct t_unitdata*)

– T_UDERROR (für *struct t_uderr*)

– T_INFO (für *struct t_info*)

Die Funktion *t_free()* überprüft im *struct_type*-Objekt **ptr* die Komponenten *ptr->addr*, *ptr->opt* und *ptr->udata* vom Typ *struct netbuf* und gibt die Puffer frei, auf die die Komponenten *buf* der einzelnen *netbuf*-Strukturen zeigen. Wenn ein Zeiger *buf* der Null-Zeiger ist, versucht *t_free()* nicht, den zugehörigen Speicher freizugeben. Sobald alle *buf*-Puffer freigegeben sind, gibt *t_free()* die Struktur frei, auf die *ptr* zeigt.

t_free() führt zu undefinierten Ergebnissen, wenn *ptr* oder irgendein Zeiger *buf* auf einen Speicherbereich zeigt, der zuvor nicht mit *t_alloc()* angelegt worden ist.

Returnwert

0:
bei Erfolg.

-1:
bei Fehler. *t_ errno* wird gesetzt, um den Fehler anzuzeigen.

Fehler

TSYSERR
Während der Ausführung dieser Funktion ist ein Systemfehler aufgetreten.

Siehe auch

t_alloc()

t_getinfo() - Protokollspezifische Informationen abfragen

```
#include <xti.h>
int t_getinfo(int fd, struct t_info *info);
```

Beschreibung

Die Funktion *t_getinfo()* liefert dem Benutzer Informationen über die aktuelle Charakteristik des darunter liegenden Transportprotokolls, das mit dem Transportendpunkt (Dateideskriptor) *fd* verbundenen ist. In der *t_info*-Struktur, auf die der Parameter *info* zeigt, liefert *t_getinfo()* die gleichen Informationen zurück, die bereits von *t_open()* beim Einrichten des Transportendpunkts *fd* zurückgeliefert wurden. Somit kann der Transportbenutzer mit *t_getinfo()* jederzeit auf die bei *t_open()* bereitgestellten Informationen zugreifen.

Die Struktur *t_info*, auf die der Parameter *info* zeigt, ist in <xti.h> wie folgt deklariert:

```
struct t_info {
long addr;     /* max. Länge der Transportprotokolladresse */
long options;  /* max. Anzahl Bytes der protokollspez. Optionen */
long tsdu;     /* max. Größe eines Datenpakets (TSDU) */
long etsdu;    /* max. Größe eines Pakets für Vorrangdaten (ETSDU) */
long connect;  /* max. erlaubte Datenmenge bei Verbindungsaufbau-Funkt. */
long discon;   /* max. erlaubte Datenmenge bei den Funktionen t_snddis() und
                  t_rcvdis() */
long servtype; /* von dem Transportanbieter angebotener Diensttyp */
long flags;    /* andere Information des Transportanbieters /*
};
```

Im Einzelnen haben die Werte der *t_info*-Komponenten folgende Bedeutung:

addr
: Ein Wert ≥ 0 gibt die maximale Länge einer Transportprotokoll-Adresse an. Der Wert -1 zeigt an, dass die Adresslänge nicht begrenzt ist. Der Wert -2 zeigt an, dass der Transportanbieter Benutzerzugriffe auf die Transportprotokolladresse nicht unterstützt.

options
: Ein Wert ≥ 0 gibt die maximale Länge (in Bytes) an, die der Transportanbieter für protokollspezifische Optionen unterstützt. Der Wert -1 zeigt an, dass die Länge der Optionen nicht begrenzt ist. Der Wert -2 zeigt an, dass der Transportanbieter Optionen, die vom Benutzer beeinflusst werden können, nicht unterstützt.

tsdu
Ein Wert > 0 gibt die maximale Länge einer Transportdienst-Dateneinheit (TSDU) an. Der Wert 0 zeigt an, dass der Transportanbieter das Konzept der TSDU nicht unterstützt, obwohl er das Senden eines Datenstroms ohne Einhaltung logischer Blockgrenzen über die Verbindung anbietet. Der Wert -1 zeigt an, dass die Länge einer TSDU nicht begrenzt ist. Der Wert -2 zeigt an, dass der Transportanbieter die Übertragung normaler Daten nicht unterstützt.

etsdu
Ein Wert > 0 gibt die maximale Länge einer vorrangigen Transportdienst-Dateneinheit (ETSDU) an. Der Wert 0 zeigt an, dass der Transportanbieter das Konzept der ETSDU nicht unterstützt, obwohl er das Senden eines Datenstroms ohne Einhaltung logischer Blockgrenzen über die Verbindung anbietet. Der Wert -1 zeigt an, dass die Länge einer ETSDU nicht begrenzt ist. Der Wert -2 zeigt an, dass der Transportanbieter die Übertragung vorrangiger Daten nicht unterstützt.

connect
Ein Wert ≥ 0 gibt die maximale Anzahl Daten an, die mit Funktionen zum Verbindungsaufbau gesendet werden können. Der Wert -1 zeigt an, dass die Menge der Daten, die während des Verbindungsaufbaus gesendet werden können, unbegrenzt ist. Der Wert -2 zeigt an, dass der Transportanbieter das Senden von Daten mit Funktionen zum Verbindungsaufbau nicht unterstützt.

discon
Ein Wert ≥ 0 gibt die maximale Anzahl Daten an, die mit den Funktionen *t_snddis()* und *t_rcvdis()* gesendet werden können. Der Wert -1 zeigt an, dass die Menge der Daten, die mit Funktionen für den Verbindungsabbau gesendet werden können, unbegrenzt ist. Der Wert -2 zeigt an, dass der Transportanbieter das Senden von Daten mit Funktionen zum Verbindungsabbau nicht unterstützt.

servtype
Diese Komponente spezifiziert den Diensttyp, der vom Transportanbieter unterstützt wird (siehe folgende Seite).

flags
Dieses Feld spezifiziert andere Informationen des Transportanbieters (derzeit werden keine Informationen geliefert).

Wenn der Transportdienstbenutzer protokollunabhängig sein möchte, kann er anhand der oben genannten Werte ermitteln, wie groß die Puffer zur Speicherung der einzelnen Informationen sein müssen. Alternativ kann der Benutzer die Funktion *t_alloc()* verwenden, um Speicher für diese Puffer anzulegen. Ein Fehler tritt auf, wenn ein Benutzer die zulässigen Grenzwerte beim Aufruf einer XTI-Funktion überschreitet.
Die in den einzelnen *t_info*-Komponenten gespeicherten Werte können sich infolge einer Option-Aushandlung (mit *t_optmgmt()*) verändern. Mit der Funktion *t_getinfo()* kann sich der Benutzer über die jeweils aktuellen Charakteristiken informieren.

Die Komponente *info->servtype* enthält nach Ausführung von *t_getinfo()* einen der folgenden Werte:

T_COTS_ORD
: Der Transportanbieter unterstützt einen verbindungsorientierten Dienst mit einem optionalen geordneten Verbindungsabbau. Für diesen Diensttyp liefert *t_getinfo()* den Wert -2 für *etsdu*, *connect* und *discon* zurück.

T_CLTS
: Der Transportanbieter unterstützt einen verbindungslosen Dienst. Für diesen Diensttyp liefert *t_getinfo()* den Wert -2 für *etsdu*, *connect* und *discon* zurück.

Returnwert

0:
: bei Erfolg.

-1:
: bei Fehler. *t_ errno* wird gesetzt, um den Fehler anzuzeigen.

Fehler

TBADF
: Der angegebene Dateideskriptor verweist nicht auf einen Transportendpunkt.

TSYSERR
: Während der Ausführung dieser Funktion ist ein Systemfehler aufgetreten.

Siehe auch

t_open()

t_getprotaddr() - Protokolladressen abfragen

```
#include <xti.h>
int t_getprotaddr(int fd, struct t_bind *boundaddr, struct t_bind *peeraddr);
```

Beschreibung

Die Funktion *getprotaddr()* liefert die lokale und die entfernte Protokolladresse zurück, die dem Transportendpunkt *fd* aktuell zugeordnet sind. Die Parameter *boundaddr* und *peeraddr* zeigen auf Objekte vom Typ *struct t_bind*.

Die Struktur *t_bind* ist in <xti.h> wie folgt deklariert:

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

Vor Aufruf von *t_getprotaddr()* spezifiziert der Benutzer in *boundaddr->maxlen* und *peeraddr->maxlen* die maximale Größe der Adresspuffer. Außerdem spezifiziert der Benutzer mit *boundaddr->addr.buf* und *peeraddr->addr.buf* jeweils einen Zeiger auf den Puffer, in dem *t_getprotaddr()* die betreffende Adresse zurückliefern soll.

Nach Ausführung von *t_getprotaddr()* zeigt *boundaddr->addr.buf* auf die Adresse (sofern vorhanden), die dem Transportendpunkt *fd* zugeordnet ist. *boundaddr->addr.len* enthält die Länge dieser Adresse. Falls sich der Transportendpunkt *fd* im Zustand T_UNBND befindet, liefert *t_getprotaddr()* in der Komponente *boundaddr->addr.len* den Wert 0 zurück.
peeraddr->addr.buf zeigt nach Ausführung von *t_getprotaddr()* auf die Adresse des Kommunikationspartners (sofern vorhanden) von *fd*.
peeraddr->addr.len enthält die Länge dieser Adresse. Falls sich der Transportendpunkt *fd* nicht im Zustand T_DATAXFER befindet, liefert *t_getprotaddr()* in der Komponente *peeraddr->addr.len* den Wert 0 zurück.

Returnwert

0:
bei Erfolg.

-1:
bei Fehler. *t_ errno* wird gesetzt, um den Fehler anzuzeigen.

Fehler

TBADF
: Der angegebene Dateideskriptor verweist nicht auf einen Transportendpunkt.

TBUFOVFLW
: Die Anzahl der Bytes, die für einen Ergebnisparameter (mit *maxlen*) reserviert wurden, ist größer als 0, aber nicht ausreichend, um den Wert dieses Parameters zu speichern.

TPROTO
: Dieser Fehler zeigt an, dass ein Kommunikationsproblem zwischen XTI und dem Transportsystem entdeckt wurde, für das es keine andere passende Fehlerbeschreibung gibt.

TSYSERR
: Während der Ausführung dieser Funktion ist ein Systemfehler aufgetreten.

Siehe auch

t_bind()

t_getstate() - Aktuellen Zustand abfragen

```
#include <xti.h>
int t_getstate(int fd);
```

Beschreibung

Die Funktion *t_getstate()* liefert den aktuellen Zustand des Transportendpunkts zurück.

Returnwert

Im Erfolgsfall wird der aktuelle Zustand des Transportendpunkts zurückgeliefert.
Im Fehlerfall liefert *t_getstate()* den Wert -1 zurück.

Der aktuelle Zustand des Transportendpunkts kann folgende Werte annehmen:

T_UNBND
: Der Transportendpunkt ist nicht an den Transportdienst gebunden.

T_IDLE
: Der Transportendpunkt ist an das Transportsystem gebunden.

T_OUTCON
: Eine abgeschickte Verbindungsanforderung wurde noch nicht bearbeitet.

T_INCON
: Eine eingetroffene Verbindungsanforderung wurde noch nicht bearbeitet.

T_DATAXFER
: Datentransfer-Phase

T_OUTREL
: Wunsch nach geordnetem Verbindungsabbau wurde abgeschickt (Warten auf Anzeige eines geordneten Verbindungsabbaus).

T_INREL
: Warten auf eine Anforderung zum geordneten Verbindungsabbau.

Wenn sich der Transportanbieter genau zum Zeitpunkt des *t_getstate()*-Aufrufs in einem Zustandsübergang befindet, beendet sich *t_getstate()* mit Fehler.

Fehler

TBADF
Der angegebene Dateideskriptor verweist nicht auf einen Transportendpunkt.

TSTATECHNG
Der Transportanbieter wechselt gerade seinen Zustand.

TSYSERR
Während der Ausführung dieser Funktion ist ein Systemfehler aufgetreten.

Siehe auch

t_open()

t_listen() - Auf Verbindungsanforderungen warten

```
#include <xti.h>

int t_listen(int fd, struct t_call *call);
```

Beschreibung

Mit der Funktion *t_listen()* hört der Benutzer den Transportendpunkt *fd* passiv nach Verbindungsanforderungen ab, die andere Transportendpunkte mit *t_connect()* an *fd* senden. Nach Ausführung von *t_listen()* zeigt der Parameter *call* auf ein Objekt vom Typ *struct t_call*, das Informationen über ankommende Verbindungsanforderungen enthält.

Die Struktur *t_call* ist in <xti.h> wie folgt deklariert:

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

In *call->addr.buf* liefert *t_listen()* die Protokolladresse des Transportdienstbenutzers zurück, der die Verbindungsanforderung gesendet hat. Vor Aufruf von *t_listen()* muss der Benutzer in *call->addr.maxlen* die maximale Länge des Ergebnispuffers *call->addr.buf* spezifizieren. Die Rückgabe von protokollspezifischen Parametern in *call->opt* und die Rückgabe von Benutzerdaten in *call->udata* werden nicht unterstützt.
Der Wert von *call->sequence* identifiziert nach Ausführung von *t_listen()* eindeutig die eingegangene Verbindungsanforderung und ermöglicht es so dem Benutzer, mehrere Verbindungsanforderungen abzuhören, bevor er auf eine dieser Anforderungen antwortet.

Im Standardfall arbeitet *t_listen()* im synchronen Modus. Im synchronen Modus wartet (blockiert) *t_listen()*, wenn keine Verbindungsanforderung vorliegt, und gibt erst nach Eintreffen einer Verbindungsanforderung die Kontrolle an den Benutzer zurück.
Wenn jedoch der Benutzer zuvor mit *t_open()* oder der POSIX-Funktion *fcntl()* O_NDELAY oder O_NONBLOCK gesetzt hat, arbeitet *t_listen()* im asynchronen Modus. *t_listen()* fragt dann nur noch auf anstehende Verbindungsanforderungen ab (*poll()*), wartet aber nicht. Wenn keine Verbindungsanforderungen vorhanden sind, liefert *t_listen()* den Wert -1 zurück und setzt *t_errno* auf TNODATA.

Returnwert

0:
bei Erfolg.

-1:
bei Fehler. *t_ errno* wird gesetzt, um den Fehler anzuzeigen.

Fehler

TBADF
Der angegebene Dateideskriptor verweist nicht auf einen Transportendpunkt.

TBADQLEN
Der Wert von *qlen* des Transportendpunkts, auf den *fd* verweist, ist 0.

TBUFOVFLW
Die Anzahl der (mit *maxlen*) für einen Ergebnisparameter reservierten Bytes reicht nicht aus, um den Wert des Parameters abzuspeichern. Der Zustand des Transportanbieters ändert sich aus der Sicht des Benutzers auf T_INCON. Die Information über die Verbindungsanforderung, die in **call* zurückgeliefert werden soll, wird gelöscht.

TLOOK
Auf dem durch *fd* übergebenen Transportendpunkt ist ein asynchrones Ereignis eingetreten und erfordert sofortige Bearbeitung.

TNODATA
O_NDELAY oder O_NONBLOCK ist gesetzt worden, aber es befindet sich keine Verbindungsanforderung in der Warteschlange.

TNOTSUPPORT
Diese Funktion wird vom darunter liegenden Transportanbieter nicht unterstützt.

TPROTO
Die Verbindung zum Transportsystem BCAM wurde beendet.

TSYSERR
Während der Ausführung dieser Funktion ist ein Systemfehler aufgetreten.

Siehe auch

t_accept(), t_bind(), t_connect(), t_open(), t_rcvconnect(), fcntl()

t_look() - Aktuelles Ereignis abfragen

```
#include <xti.h>
int t_look(int fd);
```

Beschreibung

Mit der Funktion *t_look()* fragt der Benutzer das aktuelle Ereignis auf dem durch den Parameter *fd* spezifizierten Transportendpunkt ab.
t_look() ermöglicht es dem Transportanbieter, dem Benutzer ein asynchrones Ereignis zu melden, wenn der Benutzer Funktionen im synchronen Modus ausführt. Bestimmte Ereignisse erfordern eine sofortige Meldung an den Benutzer und werden durch einen speziellen Fehlercode (TLOOK) bei der aktuellen oder als nächstes ausgeführten Funktion angezeigt. Mit *t_look()* kann der Benutzer außerdem einen Transportendpunkt periodisch auf asynchrone Ereignisse abzufragen (*poll()*).

Returnwert

Bei erfolgreicher Ausführung liefert *t_look()* einen Wert zurück, der das aufgetretene Ereignis anzeigt. Falls kein Ereignis aufgetreten ist, liefert *t_look()* den Wert 0 zurück.
Im Fehlerfall wird -1 zurückgeliefert und *t_errno* gesetzt, um den Fehler anzuzeigen.

Folgende Ereignisse können von *t_look()* zurückgeliefert werden:

T_LISTEN
: Verbindungsanzeige wurde empfangen.

T_CONNECT
: Verbindungsbestätigung wurde empfangen.

T_DATA
: Daten wurden empfangen.

T_DISCONNECT
: Anzeige des Verbindungsabbaus wurde empfangen.

T_UDERR
: Datagramm-Fehleranzeige wurde empfangen.

T_ORDREL
: Anzeige eines geordneten Verbindungsabbaus wurde empfangen.

Fehler

TBADF
Der angegebene Dateideskriptor verweist nicht auf einen Transportendpunkt.

TSYSERR
Während der Ausführung dieser Funktion ist ein Systemfehler aufgetreten.

Siehe auch

t_open()

t_open() - Transportendpunkt einrichten

```
#include <xti.h>
#include <fcntl.h>

int t_open(char *path, int oflag, struct t_info *info);
```

Beschreibung

Mit der Funktion *t_open()* richtet der Benutzer einen Transportendpunkt durch Öffnen einer Datei eines UNIX-Systems ein, die einen bestimmten Transportanbieter (d.h. das Transportprotokoll) kennzeichnet. Der Aufruf von *t_open()* ist der erste Schritt bei der Initialisierung eines Transportendpunkts.
t_open() liefert einen Dateideskriptor auf einen Transportendpunkt dieses Typs zurück.

Unterstützt werden auf Basis des TCP/IP-Protokolls:

– */dev/tcp* für das Eröffnen eines verbindungsorientierten Transportendpunkts
– */dev/udp* für das Eröffnen eines verbindungslosen Transportendpunkts

Mit dem Parameter *path* übergibt der Benutzer einen Zeiger auf den Pfadnamen der zu öffnenden Datei. *oflag* kann mittels bitweiser inklusiver ODER-Verknüpfung von O_NDELAY bzw. O_NONBLOCK mit O_RDWR gebildet werden. Diese Optionen sind in der Include-Datei <fcntl.h> deklariert.

Der mit *t_open()* eingerichtete Transportendpunkt wird in nachfolgenden Aufrufen von XTI-Funktionen durch den von *t_open()* zurückgelieferten Dateideskriptor identifiziert.

Der Parameter *info* zeigt auf ein Objekt vom Typ *struct t_info*, in dem *t_open()* die Characteristik des darunter liegenden Transportprotokolls zurückliefert.
Wenn beim Aufruf von *t_open()* als aktueller Parameter für *info* der Null-Zeiger übergeben wird, liefert *t_open()* keine Protokollinformation zurück.

Die Struktur *t_info* ist in <xti.h> wie folgt deklariert:

```
struct t_info {
long addr;     /* max. Länge der Transportprotokolladresse */
long options;  /* max. Anzahl Bytes der protokollspez. Optionen */
long tsdu;     /* max. Größe eines Datenpakets (TSDU) */
long etsdu;    /* max. Größe eines Pakets für Vorrangdaten (ETSDU) */
long connect;  /* max. erlaubte Datenmenge beiVerbindungsaufbau-Funktionen */
long discon;   /* max. erlaubte Datenmenge bei den Funktionen t_snddis() und
                  t_rcvdis() */
long servtype; /* von dem Transportanbieter angebotener Diensttyp */
long flags;    /* andere Information des Transportanbieters /*
};
```

Im Einzelnen haben die Werte der *t_info*-Komponenten folgende Bedeutung:

addr
: Ein Wert ≥ 0 gibt die maximale Länge einer Transportprotokoll-Adresse an. Der Wert -1 zeigt an, dass die Adresslänge nicht begrenzt ist. Der Wert -2 zeigt an, dass der Transportanbieter keinen Benutzerzugriff auf die Transportprotokolladresse unterstützt.

options
: Ein Wert ≥ 0 gibt an, welche maximale Länge (in Bytes) für protokollspezifische Optionen der Transportanbieter unterstützt. Der Wert -1 zeigt an, dass die Länge der Optionen nicht begrenzt ist. Der Wert -2 zeigt an, dass der Transportanbieter keine Optionen unterstützt, die vom Benutzer beeinflusst werden können.

tsdu
: Ein Wert > 0 gibt die maximale Länge einer Transportdienst-Dateneinheit (TSDU) an. Der Wert 0 zeigt an, dass der Transportanbieter das Konzept der TSDU nicht unterstützt, obwohl er das Senden eines Datenstroms ohne Einhaltung logischer Blockgrenzen über die Verbindung anbietet. Der Wert -1 zeigt an, dass die Länge einer TSDU nicht begrenzt ist. Der Wert -2 zeigt an, dass der Transportanbieter die Übertragung normaler Daten nicht unterstützt.

etsdu
: Ein Wert > 0 gibt die maximale Länge einer vorrangigen Transportdienst-Dateneinheit (ETSDU) an. Der Wert 0 zeigt an, dass der Transportanbieter das Konzept der ETSDU nicht unterstützt, obwohl er das Senden eines Datenstroms ohne Einhaltung logischer Blockgrenzen über die Verbindung anbietet. Der Wert -1 zeigt an, dass die Länge einer ETSDU nicht begrenzt ist. Der Wert -2 zeigt an, dass der Transportanbieter die Übertragung von Vorrangdaten nicht unterstützt.

connect
Ein Wert ≥ 0 gibt die maximale Anzahl Daten an, die mit Funktionen zum Verbindungsaufbau gesendet werden können. Der Wert -1 zeigt an, dass die Menge der Daten, die während des Verbindungsaufbaus gesendet werden können, unbegrenzt ist. Der Wert -2 zeigt an, dass der Transportanbieter das Senden von Daten mit Funktionen zum Verbindungsaufbau nicht unterstützt.

discon
Ein Wert ≥ 0 gibt die maximale Anzahl Daten an, die mit Funktionen *t_snddis()* und *t_rcvdis()* gesendet werden können. Der Wert -1 zeigt an, dass die Menge der Daten, die mit Funktionen für den Verbindungsabbau gesendet werden können, unbegrenzt ist. Der Wert -2 zeigt an, dass der Transportanbieter das Senden von Daten mit Funktionen zum Verbindungsabbau nicht unterstützt.

servtype
Diese Komponente spezifiziert den Diensttyp, der vom Transportanbieter unterstützt wird (siehe unten).

flags
Dieses Feld spezifiziert andere Informationen des Transportanbieters (derzeit werden keine Informationen geliefert).

Wenn der Transportdienstbenutzer protokollunabhängig sein möchte, kann er anhand der oben genannten Werte ermitteln, wie groß die Puffer zur Speicherung der einzelnen Informationen sein müssen. Alternativ kann der Benutzer die Funktion *t_alloc()* verwenden, um Speicher für diese Puffer anzulegen. Ein Fehler tritt auf, wenn ein Benutzer die zulässigen Grenzwerte beim Aufruf einer XTI-Funktion überschreitet.

Die Komponente *info->servtype* enthält nach Ausführung von *t_open()* einen der folgenden Werte:

T_COTS_ORD
Der Transportanbieter unterstützt einen verbindungsorientierten Dienst mit einem optionalen geordneten Verbindungsabbau. Für diesen Diensttyp liefert *t_open()* den Wert -2 für *etsdu*, *connect* und *discon* zurück.

T_CLTS
Der Transportanbieter unterstützt einen verbindungslosen Dienst. Für diesen Diensttyp liefert *t_open()* den Wert -2 für *etsdu*, *connect* und *discon* zurück.

Zu einem bestimmten Zeitpunkt kann ein Transportendpunkt nur einen der genannten Dienste unterstützen.

Returnwert

Bei erfolgreicher Ausführung liefert *t_open()* einen gültigen Dateideskriptor zurück.
Im Fehlerfall wird -1 zurückgeliefert und *t_errno* gesetzt, um den Fehler anzuzeigen.

Fehler

TSYSERR
Während der Ausführung dieser Funktion ist ein Systemfehler aufgetreten.

TBADFLAG
Eine ungültige Option wurde angegeben.

TBADNAME
Der in *path* angegebene Name ist ungültig.

TPROTO
Eine Verbindung zum Transportsystem konnte nicht aufgebaut werden.

Siehe auch

open()

t_optmgmt() - Transportendpunkt-Optionen verwalten

```
#include <xti.h>

int t_optmgmt(int fd, struct t_optmgmt *req, struct t_optmgmt *ret);
```

Beschreibung

Mit der Funktion *t_optmgmt()* kann ein Transportdienstbenutzer Protokoll-Optionen abfragen, verifizieren oder mit dem Transportanbieter aushandeln.

Der Parameter *fd* spezifiziert einen Transportendpunkt. Die Parameter *req* und *ret* zeigen jeweils auf ein Objekt vom Typ *struct t_optmgmt*.

Die Struktur *t_optmgmt* ist in <xti.h> wie folgt deklariert:

```
struct t_optmgmt {
   struct netbuf opt;
   long flags;
};
```

Die Komponente *opt* spezifiziert die Protokoll-Optionen. Die Komponente *flags* spezifiziert die Aktion, die mit diesen Optionen durchgeführt werden soll. Die Optionen werden durch eine *netbuf*-Struktur dargestellt, ähnlich wie die Adressen bei *t_bind()*.

Der Transportbenutzer verwendet den Parameter *req*, um mit *t_optmgmt()* eine bestimmte Aktion des Transportanbieters anzufordern und um Optionen an den Transportanbieter zu senden. In *req->opt.len* spezifiziert der Benutzer die Länge (in Bytes) des Puffers, in dem die Optionen an den Transportanbieter übergeben werden. *req->opt.buf* zeigt auf diesen Puffer. *req->opt.maxlen* ist ohne Bedeutung.

Jede Option ist im Option-Puffer *req->opt.buf* als *t_opthdr*-Struktur abgelegt und muss innerhalb des Puffers auf Wortgrenze ausgerichtet sein. Wenn der Benutzer mehrere Optionen angibt, müssen alle Optionen der gleichen Protokollebene angehören.

Die Struktur *t_opthdr* ist in <xti.h> wie folgt deklariert:

```
struct t_opthdr {
   unsigned long len;    /* gibt die gesamte Länge der Option an     */
                         /* und errechnet sich als Summe der Länge   */
                         /* der Struktur t_opthdr und der Länge      */
                         /* eines evtl. nachfolgenden Option-Werts   */
   unsigned long level;  /* spezifiziert die Protokollebene          */
   unsigned long name;   /* spezifiziert die Option                  */
   unsigned long status; /* liefert Informationen, ob Setzen/Rück-   */
                         /* setzen dieser Option durchgeführt werden */
                         /* konnte                                   */
};
```

Mit dem in <xti.h> definierten Makro OPT_NEXTHDR (*pbuf*, *buflen*, *poption*) kann der Benutzer aus dem Option-Puffer lesen und in den Option-Puffer schreiben. Der Parameter *pbuf* ist ein Zeiger auf den Anfang des Option-Puffers, *buflen* gibt die Länge des Option-Puffers an und *poption* ist ein Zeiger auf die aktuelle Option innerhalb des Puffers. Als Rückgabewert liefert der Makro OPT_NEXTHDR einen Zeiger auf die nächste Option. Falls das Ende des Option-Puffers erreicht ist, liefert der Makro den Null-Zeiger zurück.

Vor Aufruf von *t_optmgmt()* muss der Benutzer in *ret->opt.maxlen* die maximale Länge (in Bytes) des Ergebnispuffers spezifizieren, in dem *t_optmgmt()* diese Informationen zurückliefert. *req->opt.buf* zeigt auf diesen Puffer. Nach Ausführung von *t_optmgmt()* enthält *ret->opt.len* die tatsächliche Länge der zurückgelieferten Optionen und Flag-Werte.

In *req->flags* muss der Benutzer eine der folgenden Aktionen spezifizieren:

T_NEGOTIATE
: Mit dieser Aktion handelt der Benutzer mit dem Transportanbieter die Werte der Optionen aus, die er in *req->opt.buf* spezifiziert hat. Der Transportanbieter liefert die ausgehandelten Werte im Ergebnispuffer *ret->opt.buf* zurück.

  Für jede Option zeigt das Status-Feld der zugehörigen *t_opthdr*-Struktur das Ergebnis der Operation an.
  Das Statusfeld kann folgende Werte annehmen:

  – T_SUCCESS, wenn die Option erfolgreich verändert werden konnte.
  – T_PARTSUCCESS, wenn ein kleinerer Option-Wert als der angegebene gesetzt werden konnte.
  – T_FAILURE, wenn die Änderung nicht durchgeführt werden konnte.
  – T_READONLY, wenn die Option nur gelesen, aber nicht geändert werden darf.
  – T_NOTSUPPORT, wenn der Transportanbieter die Option nicht unterstützt.

In *ret->flags* liefert *t_optmgmt()* das kleinste gemeinsame Ergebnis aller option-spezifischen Ergebnisse. Dabei hat T_NOTSUPPORT die höchste Wertigkeit; die Wertigkeit der Ergebnisse nimmt ab in der Reihenfolge T_NOTSUPPORT, T_READONLY, T_FAILURE, T_PARTSUCCESS, T_SUCCESS.

Auf jeder Protokollebene kann der Benutzer mit der Option T_ALLOPT alle in der betreffenden Protokollebene unterstützten Optionen auf ihren ursprünglichen Wert zurücksetzen. Im Ergebnispuffer *ret->opt.buf* liefert *t_optmgmt()* dann alle Optionen mit den zugehörigen Werten zurück.

T_CHECK

Mit dieser Aktion kann der Benutzer prüfen, ob der Transportanbieter die in *req->opt.buf* spezifizierten Optionen unterstützt.

Wenn eine Option ohne Option-Wert spezifiziert ist, setzt *t_optmgmt()* für die betreffende Option im Ergebnispuffer *ret->opt.buf* nur das Status-Feld.
Das Statusfeld enthält einen der folgenden Werte:

- T_SUCCESS, falls der Transportanbieter die Option unterstützt.
- T_NOTSUPPORT, falls der Transportanbieter die Option nicht unterstützt.
- T_READONLY, falls die Option nur gelesen werden darf.

Wenn eine Option mit Option-Wert spezifiziert ist, liefert *t_optmgmt()* das Ergebnis in der oben unter „T_NEGOTIATE“ beschriebenen Weise. In *ret->flags* liefert *t_optmgmt()* dann das kleinste gemeinsame Ergebnis aller option-spezifischen Ergebnisse.

T_CURRENT

Mit dieser Aktion kann der Benutzer die Werte der in *req->opt.buf* spezifizierten Optionen ermitteln. Nach Ausführung von *t_optmgmt()* enthält *ret->opt.buf* die Optionen und ihre aktuellen Werte.
ret->opt.flags zeigt das Ergebnis an:

- T_SUCCESS, falls der Transportanbieter die Option unterstützt.
- T_NOTSUPPORT, falls der Transportanbieter die Option nicht unterstützt.
- T_READONLY, falls die Option nur gelesen werden darf.

Auf jeder Protokollebene kann der Benutzer mit der Option T_ALLOPT veranlassen, dass *t_optmgmt()* alle unterstützten Optionen mit den zugehörigen Werten zurückliefert.

T_DEFAULT
Mit dieser Aktion kann der Benutzer sich die Standardwerte der in **req* spezifizierten Optionen in **ret* zurückliefern lassen.
Das Statusfeld einer Option in *ret->opt.buf* hat dann folgenden Wert:

- T_SUCCESS, falls der Transportanbieter die Option unterstützt.
- T_NOTSUPPORT, falls der Transportanbieter die Option nicht unterstützt.
- T_READONLY, falls die Option nur gelesen werden darf.

In *ret->flags* liefert *t_optmgmt()* dann das kleinste gemeinsame Ergebnis aller optionspezifischen Ergebnisse.
Auf jeder Protokollebene kann sich der Benutzer mit der Option T_ALLOPT alle in der betreffenden Protokollebene unterstützten Optionen mit ihren ursprünglichen Werten zurückliefern lassen.

Protokollebenen und Optionen

Die Optionen verteilen sich auf verschiedene Protokollebenen.

Tabelle 15 gibt eine Übersicht über Protokollebenen und Optionen:

Protokollebene	Name der Option	Typ des Option-Werts	Option-Werte
XTI_GENERIC	XTI_DEBUG	unsigned long	0 oder XTI_GENERIC
INET_TCP	TCP_KEEPALIVE	struct t_kpalive	(siehe Text)
	TCP_NODELAY	unsigned long	T_YES oder T_NO
INET_IP	IP_BROADCAST	unsigned int	T_YES oder T_NO

Tabelle 15: Protokollebenen und Optionen

Optionen der Protokollebene XTI_GENERIC

XTI_DEBUG
Mit dieser Option legt der Benutzer fest, ob Diagnose-Informationen erstellt werden:

- XTI_GENERIC als Option-Wert angegeben: Diagnose-Informationen werden erstellt.
- kein Option-Wert angegeben: Diagnose-Informationen werden nicht erstellt.

Zu Diagnose-Informationen siehe auch Kapitel „XTI-Trace“ auf Seite 227.

Optionen der Protokollebene INET_TCP

TCP_KEEPALIVE

Das Setzen dieser Option aktiviert einen Mechanismus, der eine Verbindung periodisch daraufhin abprüft, ob sie noch besteht. Der Option-Wert ist gespeichert in einem Objekt vom Typ *struct t_kpalive*.

Die Struktur *t_kpalive* ist in <xti.h> wie folgt deklariert:

```
struct t_kpalive {
   long kp_onoff;     /* Option ein-/ausschalten */
   long kp_timeout;   /* keep-alive-Timeout in Minuten */
};
```

Mit dem Parameter *kp_onoff* kann die „Lebendüberwachung“ von Verbindungen ein- oder ausgeschaltet werden. *kp_onoff* kann den Wert T_YES oder T_NO annehmen. Der Parameter *kp_timeout* ist ohne Bedeutung, da das Transportsystem selbst die Zeitabstände für die Überwachung der Verbindung festlegt.

TCP_NODELAY

Mit dieser Option kann der Benutzer das Zeitverhalten beim Senden von Daten beeinflussen.
Im Standardfall werden Daten sofort gesendet. Treten jedoch Verzögerungen beim Senden einzelner Daten auf, dann sammelt das Transportsystem kleinere Datenmengen und sendet diese Daten gemeinsam zu einem späteren Zeitpunkt. Dadurch wird die Netzlast verringert. Wenn die Option TCP_NODELAY gesetzt ist (Option-Wert T_YES) ist der geschilderte Mechanismus nicht wirksam, d.h. die Daten werden sofort gesendet.

Option der Protokollebene INET_IP

IP_BROADCAST

Mit dieser Option steuert der Benutzer das Senden von Broadcast-Nachrichten.
Da Broadcast-Nachrichten im BS2000/OSD immer gesendet werden dürfen, hat die Option IP_BROADCAST keine funktionelle Bedeutung. Zu beachten ist jedoch, dass der Empfang von Broadcast-Nachrichten nicht zugelassen sein kann.

Returnwert

0:

bei Erfolg.

-1:

bei Fehler. *t_ errno* wird gesetzt, um den Fehler anzuzeigen.

Fehler

TACCES
Der Benutzer hat nicht die Berechtigung, die angegebenen Optionen auszuhandeln.

TBADF
Der angegebene Dateideskriptor verweist nicht auf einen Transportendpunkt.

TBADFLAG
Es wurde ein ungültiges Flag angegeben.

TBADOPT
Die angegebenen Protokoll-Optionen hatten ein falsches Format oder enthielten ungültige Informationen.

TBUFOVFLW
Die Anzahl der (mit *maxlen*) für einen Ergebnisparameter reservierten Bytes reicht nicht aus, um den Wert des Parameters abzuspeichern. Die in **ret* zurückzuliefernde Information wird gelöscht.

TOUTSTATE
Innerhalb einer Sequenz von XTI-Funktionsaufrufen für den Transportendpunkt *fd* wurde die Funktion an der falschen Stelle aufgerufen.

TSYSERR
Während der Ausführung dieser Funktion ist ein Systemfehler aufgetreten.

Siehe auch

t_getinfo(), t_open()

t_rcv() - Daten über eine Verbindung empfangen

```
#include <xti.h>

int t_rcv(int fd, char *buf, unsigned nbytes, int *flags);
```

Beschreibung

Mit der Funktion *t_rcv()* kann der Transportbenutzer über eine bestehende Verbindung Daten empfangen.
Der Parameter *fd* identifiziert den lokalen Transportendpunkt, über den die Daten empfangen werden. *buf* zeigt auf einen Empfangspuffer, in dem *t_rcv()* die eintreffenden Benutzerdaten ablegt. Mit *nbytes* spezifiziert der Benutzer die Länge dieses Empfangspuffers.
Der Parameter *flags* wird nicht unterstützt.

Im Standardfall arbeitet *t_rcv()* im synchronen Modus, d.h. *t_rcv()* wartet auf die Ankunft weiterer Daten und blockiert, falls im Augenblick keine Daten verfügbar sind.
Wenn jedoch zuvor mit *t_open()* oder der POSIX-Funktion *fcntl()* für den durch *fd* spezifizierten Transportendpunkt O_NDELAY oder O_NONBLOCK gesetzt wurde, arbeitet *t_rcv()* im asynchronen Modus und beendet sich mit Fehler, falls keine Daten vorhanden sind. Dabei liefert *t_rcv()* den Returnwert -1 zurück und setzt *t_errno* auf TNODATA.

Returnwert

Nach erfolgreicher Ausführung liefert *t_rcv()* die Anzahl empfangener Bytes zurück.
Im Fehlerfall wird -1 zurückgegeben und *t_errno* gesetzt, um den Fehler anzuzeigen.

Fehler

TBADF
: Der angegebene Dateideskriptor verweist nicht auf einen Transportendpunkt.

TLOOK
: Auf dem durch *fd* übergebenen Transportendpunkt ist ein asynchrones Ereignis eingetreten und erfordert sofortige Bearbeitung.

TNODATA
: O_NDELAY oder O_NONBLOCK ist gesetzt worden, aber es sind zurzeit keine Daten vom Transportanbieter verfügbar.

TNOTSUPPORT
Diese Funktion wird vom darunter liegenden Transportanbieter nicht unterstützt.

TOUTSTATE
Innerhalb einer Sequenz von XTI-Funktionsaufrufen für den Transportendpunkt *fd* wurde die Funktion an der falschen Stelle aufgerufen.

TSYSERR
Während der Ausführung dieser Funktion ist ein Systemfehler aufgetreten.

Siehe auch

t_open(), t_snd(), fcntl()

t_rcvconnect() - Status einer Verbindungsanforderung abfragen

```
#include <xti.h>

int t_rcvconnect(int fd, struct t_call *call);
```

Beschreibung

Mit der Funktion *t_rcvconnect()* kann der Transportbenutzer den Status einer Verbindung ermitteln, die er zuvor mit *t_connect()* im asynchronen Modus angefordert hat. Im asynchronen Modus wird *t_rcvconnect()* in Verbindung mit *t_connect()* verwendet, um eine Verbindung einzurichten. Nach erfolgreicher Ausführung von *t_rcvconnect()* ist die Verbindung eingerichtet.

Der Parameter *fd* spezifiziert den lokalen Transportendpunkt, auf dem die zuvor mit *t_connect()* angeforderte Verbindung eingerichtet werden soll. Der Parameter *call* zeigt auf ein Objekt vom Typ *struct t_call*, in dem *t_rcvconnect()* Informationen über die zuvor mit *t_connect()* angeforderte Verbindung zurückliefert.

Die Struktur *t_call* ist in <xti.h> wie folgt deklariert:

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

In *call->addr* liefert *t_rcvconnect()* die Protokolladresse des antwortenden Transportendpunkts. Die Rückgabe von protokollspezifischen Informationen in *call->udata* sowie von Benutzerdaten in *call->udata* wird vom Transportanbieter nicht unterstützt. *call->sequence* ist für die Funktion *t_rcvconnect()* ohne Bedeutung.
Vor Aufruf von *t_rcvconnect()* muss der Benutzer in den einzelnen *netbuf*-Strukturen von **call* die Komponente *maxlen* mit dem Wert für die jeweils maximale Puffergröße versorgen. Als aktueller Parameter für *call* kann auch der Null-Zeiger übergeben werden. In diesem Fall liefert *t_rcvconnect()* keine Informationen an den Benutzer zurück.

Im Standardfall arbeitet *t_rcvconnect()* im synchronen Modus. Im synchronen Modus wartet *t_rcvconnect()* auf eine Bestätigung der zuvor mit *t_connect()* angeforderten Verbindung und gibt erst nach Empfang der Bestätigung die Kontrolle wieder an den aufrufenden Transportbenutzer zurück. Nach Ausführung von *t_rcvconnect()* stehen in *call->addr* die gültigen Informationen über die soeben eingerichtete Verbindung.

Wenn jedoch zuvor mit *t_open()* oder der POSIX-Funktion *fcntl()* für den durch *fd* spezifizierten Transportendpunkt O_NDELAY oder O_NONBLOCK gesetzt wurde, wird *t_rcvconnect()* im asynchronen Modus ausgeführt. Im asynchronen Modus wartet *t_rcvconnect()* nicht auf eine Verbindungsbestätigung, sondern gibt sofort nach Abfrage des Status der Verbindungsanforderung die Kontrolle wieder an den aufrufenden Benutzer zurück. Falls die angeforderte Verbindung noch nicht eingerichtet ist, liefert *t_rcvconnect()* den Returnwert -1 zurück und setzt *t_errno* auf TNODATA. In diesem Fall muss der Benutzer *t_rcvconnect()* zu einem späteren Zeitpunkt erneut aufrufen, um die Verbindungsaufbau-Phase zu beenden und die zugehörigen Informationen in *call->addr* zu erhalten.

Returnwert

0:
bei Erfolg.

-1:
bei Fehler. *t_ errno* wird gesetzt, um den Fehler anzuzeigen.

Fehler

TBADF
Der angegebene Dateideskriptor verweist nicht auf einen Transportendpunkt.

TBUFOVFLW
Die Anzahl der für einen Ergebnisparameter reservierten Bytes reicht nicht aus, um den Wert des Parameters zu speichern. Der Zustand des Transportanbieters wird aus Benutzersicht auf den Zustand T_DATAXFER gesetzt, und die Information für die Verbindungsanforderung, die in **call* zurückgeliefert werden soll, wird entfernt.

TLOOK
Auf dem durch *fd* übergebenen Transportendpunkt ist ein asynchrones Ereignis eingetreten und verlangt sofortige Bearbeitung.

TNODATA
O_NDELAY oder O_NONBLOCK wurde gesetzt, aber es ist noch keine Verbindungsbestätigung angekommen.

TNOTSUPPORT
Diese Funktion wird vom darunter liegenden Transportanbieter nicht unterstützt.

TOUTSTATE
Innerhalb einer Sequenz von XTI-Funktionsaufrufen für den Transportendpunkt *fd* wurde die Funktion an der falschen Stelle aufgerufen.

TSYSERR
Während der Ausführung dieser Funktion ist ein Systemfehler aufgetreten.

Siehe auch

t_accept(), t_bind(), t_connect(), t_listen(), t_open(), fcntl()

t_rcvdis() - Ursache eines Verbindungsabbaus abfragen

```
#include <xti.h>

int t_rcvdis(int fd, struct t_discon *discon);
```

Beschreibung

Mit der Funktion *t_rcvdis()* kann der Benutzer die Ursache eines Verbindungsabbaus abfragen.
Der Parameter *fd* spezifiziert den lokalen Transportendpunkt der aufgelösten Verbindung.
Der Parameter *discon* zeigt auf ein Objekt vom Typ *struct t_discon*.

Die Struktur *t_discon* ist in <xti.h> wie folgt deklariert:

```
struct t_discon {
   struct netbuf udata;
   int reason;
   int sequence;
};
```

Nach Ausführung von *t_rcvdis()* steht in *discon->reason* ein protokollabhängiger Code, der die Ursache des Verbindungsabbaus angibt. Dieser Code entspricht einem der möglichen Werte für die Fehlervariable *errno* (definiert in <errno.h>). Folgende Codes sind derzeit möglich:

ECONNREFUSED
: Der Verbindungswunsch wurde vom Partner zurückgewiesen.

ECONNRESET
: Die Verbindung wurde vom Partner abgebrochen.

ENETDOWN
: Die Verbindung wurde vom Transportsystem abgebrochen. In diesem Fall sollte der Benutzer den Transportendpunkt mit *t_close()* schließen.

ETIMEDOUT
: Die Verbindung konnte nicht innerhalb einer bestimmten Zeitspanne aufgebaut werden.

Der in *discon->sequence* zurückgelieferte Wert identifiziert eine anstehende Verbindungsanforderung, die mit dem Verbindungsabbau in Zusammenhang steht. *discon->sequence* ist nur von Bedeutung, wenn der die Funktion *t_rcvdis()* aufrufende Transportbenutzer zuvor den Socket *fd* einmal oder mehrmals mit *t_listen()* auf anstehende Verbindungsanforderungen abgehört hat und nun diese Verbindungsanforderungen bearbeitet. Bei Eintreffen eines Verbindungsabbau-Wunsches kann der Benutzer mit dem Wert von *discon->sequence* feststellen, welche der anstehenden Verbindungsanforderungen betroffen ist.
Die Rückgabe von Benutzerdaten in *discon->udata* wird vom Transportanbieter nicht unterstützt.

Wenn der Transportbenutzer nicht an den zurückgelieferten Werten von *discon->reason* und *discon->sequence* interessiert ist, kann er beim Aufruf von *t_rcvdis()* den Null-Zeiger als aktuellen Parameter für *discon* angeben.

Returnwert

0:
bei Erfolg.

-1:
bei Fehler. *t_ errno* wird gesetzt, um den Fehler anzuzeigen.

Fehler

TBADF
Der angegebene Dateideskriptor verweist nicht auf einen Transportendpunkt.

TNODIS
Auf dem angegebenen Transportendpunkt ist zurzeit keine Anforderung zum Verbindungsabbau vorhanden.

TNOTSUPPORT
Diese Funktion wird vom darunter liegenden Transportanbieter nicht unterstützt.

TOUTSTATE
Innerhalb einer Sequenz von XTI-Funktionsaufrufen für den Transportendpunkt *fd* wurde die Funktion an der falschen Stelle aufgerufen.

TSYSERR
Während der Ausführung dieser Funktion ist ein Systemfehler aufgetreten.

Siehe auch

t_connect(), t_listen(), t_open(), t_snddis()

t_rcvrel() - Verbindungsabbau-Wunsch bestätigen

```
#include <xti.h>
int t_rcvrel(int fd);
```

Beschreibung

Mit der Funktion *t_rcvrel()* kann der Benutzer den Empfang einer Anforderung zum geordneten Verbindungsabbau bestätigen. Der Parameter *fd* spezifiziert den lokalen Transportendpunkt, der zur Verbindung gehört.

Nach Empfang der Anforderung sollte der Benutzer auf keinen Fall ein permanentes Blockieren verursachen. Der Benutzer kann jedoch weiter Daten über die Verbindung senden, sofern er noch nicht die Funktion *t_sndrel()* aufgerufen hat.

Returnwert

0:
bei Erfolg.

-1:
bei Fehler. *t_ errno* wird gesetzt, um den Fehler anzuzeigen.

Fehler

TBADF
Der angegebene Dateideskriptor verweist nicht auf einen Transportendpunkt.

TLOOK
Auf dem durch *fd* übergebenen Transportendpunkt ist ein asynchrones Ereignis eingetreten und erfordert sofortige Bearbeitung.

TNOREL
Auf dem angegebenen Transportendpunkt ist zurzeit keine Anzeige für einen geordneten Verbindungsabbau vorhanden.

TNOTSUPPORT
Diese Funktion wird vom darunter liegenden Transportanbieter nicht unterstützt.

TOUTSTATE
Innerhalb einer Sequenz von XTI-Funktionsaufrufen für den Transportendpunkt *fd* wurde die Funktion an der falschen Stelle aufgerufen.

TSYSERR
Während der Ausführung dieser Funktion ist ein Systemfehler aufgetreten.

Siehe auch

t_open(), t_sndrel()

t_rcvudata() - Datagramme empfangen

```
#include <xti.h>
int t_rcvudata(int fd, struct t_unitdata *unitdata, int *flags);
```

Beschreibung

Mit der Funktion *t_rcvudata()* kann der Benutzer im verbindungslosen Modus ein Datagramm von einem anderen Benutzer empfangen.
Der Parameter *fd* spezifiziert den lokalen Transportendpunkt, über den das Datagramm empfangen wird. *flags* informiert den Benutzer nach Ausführung von *t_rcvudata()*, ob das Datagramm vollständig empfangen wurde. *unitdata* ist ein Zeiger auf ein Objekt vom Typ *struct t_unitdata*, in dem *t_rcvudata()* Informationen über das empfangene Datagramm zurückliefert.

Die Struktur *t_unitdata* ist in <xti.h> wie folgt deklariert:

```
struct t_unitdata {
   struct netbuf addr;
   struct netbuf opt;
   struct netbuf udata;
};
```

Vor Aufruf von *t_rcvudata()* muss der Benutzer in den einzelnen *netbuf*-Strukturen von **unitdata* die Komponente *maxlen* mit dem Wert für die jeweils maximale Puffergröße versorgen.

Nach Ausführung von *t_rcvudata()* enthält *unitdata->addr* die Protokolladresse des Senders, *unitdata->opt* enthält protokollspezifische Optionen, die das empfangene Datagramm betreffen und *unitdata->udata* enthält die empfangenen Benutzerdaten.

Im Standardfall arbeitet *t_rcvudata()* im synchronen Modus, d.h. *t_rcvudata()* wartet auf die Ankunft eines Datagramms und blockiert, falls momentan keine Datagramme vorhanden sind.
Wenn jedoch zuvor mit *t_open()* oder der POSIX-Funktion *fcntl()* für den durch *fd* spezifizierten Transportendpunkt O_NDELAY oder O_NONBLOCK gesetzt wurde, arbeitet *t_rcvudata()* im asynchronen Modus und beendet sich mit Fehler, falls keine Datagramme vorhanden sind. Dabei liefert *t_rcvudata()* den Returnwert -1 zurück und setzt *t_errno* auf TNODATA.

Wenn der Puffer in *unitdata->udata* für die Aufnahme des Datagramms nicht groß genug ist, legt *t_rcvudata()* das Datagramm so weit wie möglich im Puffer ab und setzt das Flag T_MORE. Das Flag T_MORE zeigt an, dass ein weiterer *t_rcvudata()*-Aufruf erforderlich ist, um den Rest des Datagramms zu empfangen. Solange das Datagramm nicht vollständig empfangen ist, liefern nachfolgende *t_rcvudata()*-Aufrufe als Länge von Protokolladresse und Optionen jeweils den Wert 0 zurück.

Returnwert

0:
bei Erfolg.

-1:
bei Fehler. *t_ errno* wird gesetzt, um den Fehler anzuzeigen.

Fehler

TBADF
Der angegebene Dateideskriptor verweist nicht auf einen Transportendpunkt.

TBUFOVFLW
Die Anzahl der Bytes, die für die zurückzuliefernde Protokolladresse oder die Optionen angelegt wurde, ist zu klein, um diese Informationen zu speichern. Die Informationen, die in **unitdata* zurückzuliefern sind, werden gelöscht.

TLOOK
Auf dem durch *fd* übergebenen Transportendpunkt ist ein asynchrones Ereignis eingetreten und erfordert sofortige Bearbeitung.

TNODATA
O_NDELAY oder O_NONBLOCK wurde gesetzt, aber es sind zurzeit keine Datagramme vom Transportanbieter verfügbar.

TNOTSUPPORT
Diese Funktion wird vom darunter liegenden Transportanbieter nicht unterstützt.

TOUTSTATE
Innerhalb einer Sequenz von XTI-Funktionsaufrufen für den Transportendpunkt *fd* wurde die Funktion an der falschen Stelle aufgerufen.

TSYSERR
Während der Ausführung dieser Funktion ist ein Systemfehler aufgetreten.
Im vorliegenden Fall enthält die Fehlervariable *errno* genauere Informationen:

EFAULT Der Bereich, der in *unitdata->addr*, *unitdata->opt* oder *unitdata->udata* angegeben ist, liegt außerhalb des Prozess-Adressbereichs.

ETIME Das Datagramm wurde gelöscht, da ein transportsystem-abhängiges Zeitlimit überschritten wurde (siehe auch „Relevante Einstellungen beim Transportsystem BCAM“ auf Seite 317).

EINTR Der Aufruf wurde durch ein Signal unterbrochen.

Siehe auch

t_rcvuderr(), t_sndudata()

t_rcvuderr() - Fehlerinformation über gesendetes Datagramm abfragen

```
#include <xti.h>

int t_rcvuderr(int fd, struct t_uderr *uderr);
```

Beschreibung

Mit der Funktion *t_rcvuderr()* kann der Benutzer im verbindungslosen Modus Informationen abfragen, die sich auf einen Fehler bei einem zuvor abgeschickten oder empfangenen Datagramm beziehen. *t_rcvuderr()* sollte nur nach einer Fehleranzeige aufgerufen werden.

Der Parameter *fd* spezifiziert den lokalen Transportendpunkt, über den die Fehlermeldung empfangen wird. Der Parameter *uderr* ist ein Zeiger auf ein Objekt vom Typ *struct t_uderr*.

Die Struktur *t_uderr* ist in <xti.h> wie folgt deklariert:

```
struct t_uderr {
   struct netbuf addr;
   struct netbuf opt;
   long error;
};
```

Vor Aufruf von *t_rcvuderr()* muss der Benutzer in *uderr->addr* die Komponente *maxlen* mit dem Wert für die jeweils maximale Puffergröße versorgen.
Die Rückgabe protokollspezifischer Optionen in *uderr->opt* wird vom Transportanbieter nicht unterstützt.

In *uderr->error* liefert *t_uderr()* einen protokollspezifischen Fehlercode zurück. Dieser Fehlercode entspricht einem der möglichen Werte für die Fehlervariable *errno* (definiert in <errno.h>). Folgende Codes sind derzeit möglich:

EADDRNOTAVAIL
: Der Partner, an den das zuletzt mit *t_sndudata()* verschickte Datagramm gesendet werden sollte, ist nicht erreichbar.

ENETDOWN
: Der Transportendpunkt wurde durch das Transportsystem von diesem getrennt. In diesem Fall sollte der Benutzer den Transportendpunkt mit *t_close()* schließen.

Wenn der Benutzer das fehlerhafte Datagramm nicht ermitteln will, kann er beim Aufruf von *t_rcvuderr()* den Null-Zeiger als aktuellen Parameter für *uderr* angeben.

Returnwert

0:
bei Erfolg.

-1:
bei Fehler. *t_ errno* wird gesetzt, um den Fehler anzuzeigen.

Fehler

TBADF
Der angegebene Dateideskriptor verweist nicht auf einen Transportendpunkt.

TBUFOVFLW
Die Anzahl der Bytes, die für die zurückzuliefernde Protokolladresse oder die Optionen angelegt wurde, ist zu klein, um diese Informationen zu speichern. Die Informationen, die in **uderr* zurückzuliefern sind, werden nicht berücksichtigt.

TNOTSUPPORT
Diese Funktion wird vom darunter liegenden Transportanbieter nicht unterstützt.

TNOUDERR
Zurzeit liegt auf dem angegebenen Transportendpunkt keine Fehlermeldung zu einem Datagramm vor.

TSYSERR
Während der Ausführung dieser Funktion ist ein Systemfehler aufgetreten.

Siehe auch

t_rcvdata(), t_sndudata()

t_snd() - Daten über eine Verbindung senden

```
#include <xti.h>

int t_snd(int fd, char *buf, unsigned nbytes, int flags);
```

Beschreibung

Mit der Funktion *t_snd()* sendet der Benutzer Daten.
Der Parameter *fd* spezifiziert den lokalen Transportendpunkt, über den die Daten gesendet werden sollen. *buf* ist ein Zeiger auf die zu sendenden Benutzerdaten. Mit *nbytes* spezifiziert der Benutzer die Länge (in Bytes) der zu versendenden Benutzerdaten. *flags* wird vom Transportanbieter nicht unterstützt. Deshalb muss beim Aufruf von *t_snd()* für *flags* der Wert 0 übergeben werden.

Im Standardfall arbeitet *t_snd()* im synchronen Modus. Im synchronen Modus wartet (blockiert) *t_snd()*, falls Flusskontroll-Beschränkungen verhindern, dass zum Zeitpunkt des *t_snd()*-Aufrufs alle Daten vom Transportanbieter übernommen werden können.
Wenn jedoch zuvor mit *t_open()* oder der POSIX-Funktion *fcntl()* für den durch *fd* spezifizierten Transportendpunkt O_NDELAY oder O_NONBLOCK gesetzt wurde, wird *t_snd()* im asynchronen Modus ausgeführt und beendet sich mit Fehler, falls Beschränkungen der Flusskontrolle bestehen.

Bei erfolgreicher Ausführung gibt der Rückgabewert von *t_snd()* an, wie viele Daten-Bytes vom Transportanbieter angenommen wurden. Normalerweise wird diese Zahl dem im Parameter *nbytes* übergebenen Wert entsprechen. Im asynchronen Modus wird jedoch möglicherweise nur ein Teil der zu sendenden Daten vom Transportanbieter angenommen.
In diesem Fall liefert *t_snd()* einen Wert kleiner als *nbytes* zurück.

Returnwert

Bei erfolgreicher Ausführung liefert *t_snd()* die Anzahl Bytes zurück, die vom Transportanbieter angenommen wurden.
Im Fehlerfall wird *t_errno* auf -1 gesetzt, um den Fehler anzuzeigen.

Fehler

TBADDATA
: Der Parameter *nbytes* hat den Wert 0, aber das Versenden von Null-Bytes wird vom darunter liegenden Transportanbieter nicht unterstützt.

TBADF
: Der angegebene Dateideskriptor verweist nicht auf einen Transportendpunkt.

TFLOW
: O_NDELAY oder O_NONBLOCK wurde gesetzt, aber die Flusskontrolle hat nicht erlaubt, dass der Transportanbieter zu diesem Zeitpunkt Daten annimmt.

TLOOK
: Auf dem durch *fd* übergebenen Transportendpunkt ist ein asynchrones Ereignis eingetreten und erfordert sofortige Bearbeitung.

TNOTSUPPORT
: Diese Funktion wird vom darunter liegenden Transportanbieter nicht unterstützt.

TOUTSTATE
: Innerhalb einer Sequenz von XTI-Funktionsaufrufen für den Transportendpunkt *fd* wurde die Funktion an der falschen Stelle aufgerufen.

TSYSERR
: Während der Ausführung dieser Funktion ist ein Systemfehler aufgetreten.

Siehe auch

t_open(), t_rcv(), fcntl()

t_snddis() - Verbindung zurückweisen oder abbrechen

```
#include <xti.h>
int t_snddis(int fd, struct t_call *call);
```

Beschreibung

Mit der Funktion *t_snddis()* kann der Benutzer folgende Aktionen veranlassen:

– Verbindungsanforderung zurückweisen
– sofortigen Abbau (abortive release) einer bestehenden Verbindung einleiten

Der Parameter *fd* spezifiziert den lokalen Transportendpunkt der abzubauenden bzw. angeforderten Verbindung. Der Parameter *call* zeigt auf ein Objekt vom Typ *struct t_call*.

Die Struktur *t_call* ist in <xti.h> wie folgt deklariert:

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

Je nachdem, ob mit *t_snddis()* eine Verbindungsanforderung zurückgewiesen oder eine Verbindung abgebaut werden soll, wird der Parameter *call* unterschiedlich verwendet:

– Soll eine Verbindungsanforderung zurückgewiesen werden, darf beim Aufruf von *t_snddis()* für *call* nicht der Null-Zeiger übergeben werden. In *call->sequence* muss der Benutzer einen Wert spezifizieren, der die abgelehnte Verbindungsanforderung gegenüber dem Transportanbieter identifiziert. Die Inhalte von *call->addr*, *call->opt* und *call->udata* werden von *t_snddis()* ignoriert.
– Soll eine Verbindung abgebaut werden, darf *call* der Null-Zeiger sein.

Returnwert

0:
bei Erfolg.

-1:
bei Fehler. *t_ errno* wird gesetzt, um den Fehler anzuzeigen.

Fehler

TBADF
: Der angegebene Dateideskriptor verweist nicht auf einen Transportendpunkt.

TBADSEQ
: Es wurde eine ungültige Folgenummer angegeben, oder bei Ablehnung einer Verbindungsanforderung wurde für *call* der Null-Zeiger angegeben. Die abgehende Warteschlange des Transportanbieters wird gelöscht, was Datenverlust zur Folge haben kann.

TLOOK
: Auf dem durch *fd* übergebenen Transportendpunkt ist ein asynchrones Ereignis eingetreten und erfordert sofortige Bearbeitung.

TNOTSUPPORT
: Diese Funktion wird vom darunter liegenden Transportanbieter nicht unterstützt.

TOUTSTATE
: Innerhalb einer Sequenz von XTI-Funktionsaufrufen für den Transportendpunkt *fd* wurde die Funktion an der falschen Stelle aufgerufen. Die abgehende Warteschlange des Transportanbieters kann gelöscht werden, was Datenverlust zur Folge haben kann.

TSYSERR
: Während der Ausführung dieser Funktion ist ein Systemfehler aufgetreten.

Siehe auch

t_connect(), t_getinfo(), t_listen(), t_open()

t_sndrel() - Geordneten Verbindungsabbau einleiten

```
#include <xti.h>
int t_sndrel(int fd);
```

Beschreibung

Mit der Funktion *t_sndrel()* leitet der Benutzer den geordneten Abbau einer Transportverbindung ein. Außerdem informiert *t_sndrel()* den Transportanbieter, dass der Benutzer keine weiteren Daten sendet.
Der Parameter *fd* spezifiziert den lokalen Transportendpunkt der Verbindung, die abgebaut werden soll.

Nach Ausführung von *t_sndrel()* darf der Benutzer keine Daten mehr über die Verbindung senden. Er darf jedoch weiter Daten über die Verbindung empfangen, solange er selbst noch keine Anforderung zum geordneten Verbindungsabbau erhalten hat.

Returnwert

0:
bei Erfolg.

-1:
bei Fehler. *t_ errno* wird gesetzt, um den Fehler anzuzeigen.

Fehler

TBADF
Der angegebene Dateideskriptor verweist nicht auf einen Transportendpunkt.

TFLOW
O_NDELAY oder O_NONBLOCK wurde gesetzt, aber die Flusskontrolle hat dem Transportanbieter nicht erlaubt, die Funktion zu diesem Zeitpunkt zu akzeptieren.

TLOOK
Auf dem durch *fd* übergebenen Transportendpunkt ist ein asynchrones Ereignis eingetreten und erfordert sofortige Bearbeitung.

TNOTSUPPORT
Diese Funktion wird vom darunter liegenden Transportanbieter nicht unterstützt.

TOUTSTATE
Innerhalb einer Sequenz von XTI-Funktionsaufrufen für den Transportendpunkt *fd* wurde die Funktion an der falschen Stelle aufgerufen.

TSYSERR
Während der Ausführung dieser Funktion ist ein Systemfehler aufgetreten.

Siehe auch

t_open(), t_rcvrel()

t_sndudata() - Datagramme versenden

```
#include <xti.h>
int t_sndudata(int fd, struct t_unitdata *unitdata);
```

Beschreibung

Mit der Funktion *t_sndudata()* sendet der Benutzer im verbindungslosen Modus ein Datagramm an einen anderen Transportbenutzer.
Der Parameter *fd* spezifiziert den lokalen Transportendpunkt, über den das Datagramm gesendet wird. Der Parameter *unitdata* ist ein Zeiger auf ein Objekt vom Typ *struct t_unitdata*.

Die Struktur *t_unitdata* ist in <xti.h> wie folgt deklariert:

```
struct t_unitdata {
   struct netbuf addr;
   struct netbuf opt;
   struct netbuf udata;
};
```

Vor Aufruf von *t_rcvudata()* spezifiziert der Benutzer in *unitdata->addr* die Ziel-Protokolladresse und in *unitdata->udata* die zu übertragenden Daten. Das Setzen von protokollspezifischen Optionen in *unitdata->opt* wird vom Transportanbieter nicht unterstützt.
Falls der Benutzer in *unitdata->addr.len* den Wert 0 spezifiziert hat und der Transportanbieter das Versenden von Null-Bytes nicht unterstützt, liefert *t_sndudata()* den Wert -1 zurück und setzt *t_errno* auf TBADDATA.

Im Standardfall arbeitet *t_sndudata()* im synchronen Modus, d.h. *t_sndudata()* wartet (blockiert), falls Flusskontroll-Beschränkungen verhindern, dass der Transportanbieter das Datagramm zum Zeitpunkt des Aufrufs von *t_sndudata()* akzeptiert.
Wenn jedoch zuvor mit *t_open()* oder der POSIX-Funktion *fcntl()* für den durch *fd* spezifizierten Transportendpunkt O_NDELAY oder O_NONBLOCK gesetzt wurde, arbeitet *t_sndudata()* im asynchronen Modus und beendet sich mit Fehler, falls der Transportanbieter das Datagramm nicht sofort akzeptiert.

Wenn *t_sndudata()* in einem ungültigen Zustand aufgerufen wurde oder wenn die in *unitdata->udata.len* spezifizierte Datagrammlänge größer ist als die TSDU-Länge, generiert der Transportanbieter einen EPROTO-Protokollfehler (siehe Fehler TSYSERR). Wenn der EPROTO-Fehler auf Grund eines ungültigen Zustands generiert wurde, wird der Fehler erst dann gemeldet, wenn der Transportendpunkt *fd* referenziert wird. Die Länge der TSDU (Transportdienst-Dateneinheit) wird von den Funktionen *t_open()* und *t_getinfo()* zurückgeliefert.

Returnwert

0:
bei Erfolg.

-1:
bei Fehler. *t_ errno* wird gesetzt, um den Fehler anzuzeigen.

Fehler

TBADADDR
Die angegebene Protokolladresse hatte ein falsches Format oder enthielt falsche Informationen.

TBADDATA
Der Parameter *nbytes* hat den Wert 0, aber das Versenden von Null-Bytes wird vom darunter liegenden Transportanbieter nicht unterstützt, oder die Nachricht war zu groß, um auf einmal gesendet zu werden.

TBADF
Der angegebene Dateideskriptor verweist nicht auf einen Transportendpunkt.

TFLOW
O_NDELAY oder O_NONBLOCK wurde gesetzt, aber die Flusskontrolle hat nicht erlaubt, dass der Transportanbieter zu diesem Zeitpunkt Daten annimmt.

TLOOK
Auf dem durch *fd* übergebenen Transportendpunkt ist ein asynchrones Ereignis eingetreten und erfordert sofortige Bearbeitung.

TNOTSUPPORT
Diese Funktion wird vom darunter liegenden Transportanbieter nicht unterstützt.

TSYSERR
Während der Ausführung dieser Funktion ist ein Systemfehler aufgetreten.
Im vorliegenden Fall enthält die Fehlervariable *errno* genauere Informationen:

EFAULT Der Bereich, der in *unitdata->addr*, *unitdata->opt* oder *unitdata->udata* angegeben ist, liegt außerhalb des Prozess-Adressbereichs.

ENOBUFS
Zurzeit stehen zu wenig System-Ressourcen zur Verfügung, um den Sendeauftrag auszuführen.

EINTR Der Aufruf wurde durch ein Signal unterbrochen.

Siehe auch

t_rcvudata(), t_rcvuderr(), fcntl()

t_strerror() - Fehlermeldung ausgeben

```
#include <xti.h>
char *t_strerror(int errnum);
```

Beschreibung

Mit der Funktion *t_strerror()* kann sich der Benutzer zu einer XTI-Fehlernummer bzw. zum entsprechenden *t_errno*-Fehlercode den zugehörigen Meldungstext generieren lassen.

t_strerror() bildet die durch den Parameter *errnum* spezifizierte XTI-Fehlernummer auf den zughörigen Meldungs-String ab und liefert als Rückgabewert einen Zeiger auf diesen Character-String. Dieser Meldungs-String wird vom Programm nicht verändert, kann aber durch nachfolgende *t_strerror()*-Aufrufe überschrieben werden. Der Meldungs-String wird nicht durch ein Newline-Zeichen abgeschlossen.

Returnwert

Die Funktion *t_strerror()* liefert einen Zeiger auf den generierten Character-String.

Siehe auch

t_error()

t_sync() - Transportbibliothek synchronisieren

```
#include <xti.h>
int t_sync(int fd);
```

Beschreibung

Mit der Funktion *t_sync()* kann der Benutzer für den durch *fd* spezifizierten Transportendpunkt die von der Transportbibliothek verwalteten Datenstrukturen mit Informationen des darunter liegenden Transportanbieters synchronisieren. Außerdem ermöglicht *t_sync()* zwei kooperierenden Prozessen, ihre Interaktionen mit dem Transportanbieter zu sychronisieren.
Wenn beispielsweise ein Prozess einen neuen Prozess erzeugt und *exec()* aufruft, muss der neue Prozess die Funktion *t_sync()* aufrufen,

– um die private Bibliotheks-Datenstruktur aufzubauen, die mit einem Transportendpunkt verbunden ist und

– um die Datenstruktur mit relevanten Informationen des Transportanbieters zu synchronisieren.

Dabei ist zu beachten, dass der Transportanbieter alle Benutzer eines Transportendpunkts als einen einzigen Benutzer ansieht. Wenn also mehrere Benutzerprozesse denselben Transportendpunkt verwenden, sollten sie ihre Aufgaben so koordinieren, dass der Transportanbieter nicht in einen fehlerhaften Zustand gerät. Zu diesem Zweck können die einzelnen Benutzerprozesse mit *t_sync()* den aktuellen Transportanbieter-Zustand abfragen, bevor sie weitere Aktionen veranlassen.

Die Koordination mit *t_sync()* ist nur zwischen kooperierenden Prozessen erlaubt, da möglicherweise ein Prozess oder ein ankommendes Ereignis den Zustand des Transportanbieters verändert, nachdem *t_sync()* ausgeführt wurde.

Returnwert

t_sync() liefert bei erfolgreicher Ausführung den Zustand des Transportanbieters zurück. Im Fehlerfall wird -1 zurückgeliefert und *t_errno* wird gesetzt, um den Fehler anzuzeigen.

Als Returnwerte von *t_sync()* sind folgende Zustände des Transportanbieters möglich:

T_UNBND
: Der Transportendpunkt ist nicht an den Transportdienst gebunden.

T_IDLE
: Der Transportendpunkt ist an den Transportdienst gebunden.

T_OUTCON
: Eine abgeschickte Verbindungsanforderung wurde noch nicht bearbeitet.

T_INCON
: Eine eingetroffene Verbindungsanforderung wurde noch nicht bearbeitet.

T_DATAXFER
: Datentransfer-Phase

T_OUTREL
: Wunsch nach geordnetem Verbindungsabbau wurde abgeschickt (Warten auf Anzeige eines geordneten Verbindungsabbaus).

T_INREL
: Warten auf eine Anforderung zum geordneten Verbindungsabbau.

Fehler

TBADF
: Der angegebene Dateideskriptor verweist nicht auf einen Transportendpunkt.

TSTATECHNG
: Der Transportanbieter erfährt eine Zustandsänderung.

TSYSERR
: Während der Ausführung dieser Funktion ist ein Systemfehler aufgetreten.

Siehe auch

dup(), exec(), fork(), open()

t_unbind() - Transportendpunkt deaktivieren

```
#include <xti.h>
int t_unbind(int fd);
```

Beschreibung

Mit der Funktion *t_unbind()* kann der Transportbenutzer einen Transportendpunkt deaktivieren, dem zuvor mit der Funktion *t_bind()* eine Adresse zugeordnet worden ist. Der Parameter *fd* spezifiziert den Transportendpunkt, der deaktiviert werden soll.
Nach erfolgreicher Ausführung von *t_unbind()* nimmt der Transportanbieter keine weiteren Daten oder Ereignisse mehr an, die für den Transportendpunkt *fd* bestimmt sind.

Returnwert

0:
bei Erfolg.

-1:
bei Fehler. *t_ errno* wird gesetzt, um den Fehler anzuzeigen.

Fehler

TBADF
Der angegebene Dateideskriptor verweist nicht auf einen Transportendpunkt.

TLOOK
Auf dem durch *fd* übergebenen Transportendpunkt ist ein asynchrones Ereignis eingetreten und erfordert sofortige Bearbeitung.

TNOTSUPPORT
Diese Funktion wird vom darunter liegenden Transportanbieter nicht unterstützt.

TOUTSTATE
Innerhalb einer Sequenz von XTI-Funktionsaufrufen für den Transportendpunkt *fd* wurde die Funktion an der falschen Stelle aufgerufen.

TSYSERR
Während der Ausführung dieser Funktion ist ein Systemfehler aufgetreten.

Siehe auch

t_bind()

12 Kommunikationsanwendung übersetzen und binden

In diesem Kapitel ist Folgendes beschrieben:

- Übersetzen und Binden eines SOCKETS(POSIX)- oder XTI(POSIX)-Anwenderprogramms mit den Kommandos der POSIX-Shell.
- Übersetzen und Binden eines SOCKETS(POSIX)-Anwenderprogramms im BS2000/OSD am Beispiel zweier BS2000/OSD-Prozeduren.

12.1 Übersetzen und binden mit POSIX-Shell

Wenn die Quelldatei im UFS abgelegt ist, können Sie Ihr Anwenderprogramm mit dem folgenden Kommando der POSIX-Shell übersetzen:

```
c89 -c programm.c
```

Bei Bedarf können Sie die Schalter -O zum Optimieren des Programmcodes und -g zum Debuggen verwenden.

Das folgende Kommando bindet das übersetzte Programm:

```
c89 -o programm programm.o -lxnet
```

Das folgende Kommando übersetzt und bindet das Programm in einem Arbeitsgang:

```
c89 -o programm programm.c -lxnet
```

Damit Sie Ihr SOCKETS(POSIX)- bzw. XTI(POSIX)-Anwenderprogramm mit der POSIX-Shell übersetzen können, muss zusätzlich das Installationspaket POSIX_HEADER installiert sein. Die Paket-Installation ist beschrieben im POSIX-Handbuch „Grundlagen für Anwender und Systemverwalter“.

Die Funktionen von SOCKETS(POSIX) unterstützen die Möglichkeit des C-Compilers, Programme zu erzeugen, die ASCII-Literale enthalten, siehe auch Compiler-Handbuch „C/C++ V3.1A (BS2000/OSD)“.

12.2 Übersetzen und binden im BS2000/OSD

Die nachfolgend dargestellte Beispielprozedur zeigt, wie eine SOCKETS(POSIX)- bzw. XTI(POSIX)-Anwendung im BS2000/OSD übersetzt und gebunden werden kann.

Beispiel

```
/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 ** STARTEN DES COMPILERS **
/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 IM FOLGENDEN WIRD DAS UEBERSETZTE PROGRAMM MIT DEN BENOETIGTEN
/REMARK BIBLIOTHEKEN GEBUNDEN. DIE BIBLIOTHEKEN MUESSEN GEMAESS DER IN
/REMARK DIESER PROZEDUR DARGESTELLTEN REIHENFOLGE ANGEGEBEN WERDEN.
/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 ** STARTEN DES BINDERS ***
/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 Konfiguration und Konfigurationsdateien

Beim Start des POSIX-Subsystems werden alle notwendigen Schritte zur Konfiguration der Netzanbindung automatisch durchlaufen. Für den Benutzer sichtbar ist dabei nur der Start des Dämonprogramms *inetd* durch den *init*-Prozess.

In diesem Kapitel sind beschrieben:

- Dämonprogramm *inetd* (Internet-Superserver)
- Konfigurationsdateien für Rechner, Netze, Protokolle und Services
- Abhängigkeiten der SOCKETS(POSIX)- und XTI(POSIX)-Anwendungen vom BS2000/OSD-Transportsystem BCAM

13.1 Dämonprogramm inetd

inetd ist einer der Internet-Dämonen in UNIX-Systemen. Da *inetd* eine zentrale Rolle beim Starten der Internet-Dienste spielt, wird *inetd* auch „Internet-Super-Server“ genannt.

Wie auch in UNIX-Systemen wird *inetd* mit Hilfe der Datei */etc/inet/inetd.conf* konfiguriert. *inetd* wird beim Hochfahren des Systems gestartet. Dabei stellt *inetd* anhand der Datei *inetd.conf* fest, welche Dienste bei Bedarf über *inetd* gestartet werden sollen. *inetd* erzeugt dann für jeden in der Datei *inetd.conf* spezifizierten Dienst einen Socket und weist jedem dieser Sockets eine Portnummer zu.

Mit *select()*-Aufrufen (siehe Seite 151) für die einzelnen Sockets stellt *inetd* sicher, dass die Sockets bereit zum Lesen sind. Anschließend hört *inetd* die einzelnen Sockets mit der Funktion *listen()* auf Verbindungsanforderungen der Clients ab.
Mit jedem Socket, an dem eine Verbindungsanforderung ansteht, verfährt *inetd* wie folgt:

1. *inetd* nimmt die Verbindungsanforderung mit *accept()* an.
2. *inetd* erzeugt für den Socket mit *fork()* und *dup()* zwei Dateideskriptoren 0 (*stdin*) und 1 (*stdout*).
3. *inetd* startet für den Socket mit *exec()* die entsprechenden Dienste.

Der Einsatz von *inetd* hat also den Vorteil, dass nicht bereits beim Hochfahren des Systems alle Server-Prozesse gestartet werden müssen: Ein Server muss erst dann gestartet werden, wenn für ihn Anforderungen eines Clients vorliegen.

Außerdem erleichtert *inetd* die Aufgaben eines Servers, da *inetd* sich beim Verbindungsaufbau um den Großteil des Kommunikationsablaufs kümmert. Der Server kann voraussetzen, dass der ihm zugeordnete Kommunikationsendpunkt die Dateideskriptoren 0, 1 und 2 besitzt und bereits mit dem Client verbunden ist. Somit kann der Server sofort Funktionen wie *read()*, *write()*, *send()* oder *rcv()* ausführen, d.h. der Programmcode des Servers kann sehr einfach gehalten werden.

Ein Anwendungsprogrammierer, der über *inetd* gestartete Server entwickelt, kann die Adresse des Kommunikationspartners, d.h. die Adresse des Client-Sockets, mit der Funktion *getpeername()* (siehe Seite 106) ermitteln.

13.2 Konfigurationsdateien

In diesem Abschnitt sind die folgenden Dateien beschrieben:

– inetd.conf
– protocols
– services
– networks
– hosts

Wenn Sie Änderungen an diesen Dateien vorgenommen haben, müssen Sie das Dämonprogramm *inetd* mit dem folgenden POSIX-Kommando veranlassen, die Dateien neu zu lesen:

```
kill -1 Prozess-Nummer_von_inetd
```

13.2.1 inetd.conf - verfügbare Server

Die Datei enthält Einträge für die Server, die das Dämonprogramm *inetd* aufruft, wenn eine Anforderung über die Socket-Schnittstelle eintrifft. Im POSIX-Subsystem sind per Voreinstellung nur die Services *echo* und *time* des *inetd* sowie die Programme der R-Kommandos (*rlogin* ...) aktiviert.

Jeder Eintrag für einen Server besteht aus einer Zeile der folgenden Form:

```
Service-Name Socket-Typ Protokoll Wartezustand Kennung Server-Programm
Server-Argumente
```

Service-Name
: Name des Service, wie er in */etc/inet/services* steht

Socket-Typ
: Typ des Sockets. Datagramm- oder Stream-Socket

Protokoll
: Name des Protokolls, wie er in */etc/inet/protocols* steht. Anstelle von *tcp* und *udp* können auch *tcp6* und *udp6* stehen, sofern der betreffende Server IPv6 unterstützt.

Wartezustand
: Gibt an, ob der Server den Socket sofort freigibt (*nowait*) oder erst nach einer gewissen Zeit (*wait*).

Kennung
: Benutzerkennung, unter der der Server ablaufen soll

Server-Programm
: Pfadname des Server-Programms

Server-Argumente
: Mögliche Parameter für den Server-Aufruf

Beispiel

```
#
# 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
```

Zusätzlich installierte Server-Anwendungen können von der Systembetreuung in dieser Datei eingetragen werden.

13.2.2 protocols - verfügbare Protokolle

Die Datei enthält Informationen über die möglichen Protokolle. Jeder Eintrag für ein Protokoll besteht aus einer Zeile der folgenden Form:

```
Protokollname  Protokollnummer Aliase #kommentar
```

Beispiel

```
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
```

Die Datei ist als statisch anzusehen, da diese Nummern nur von Normungsgremien (OSI, IEEE und IANA) vergeben werden.

13.2.3 services - verfügbare Services

Die Datei enthält Informationen über die Services. Jeder Eintrag für ein Protokoll besteht aus einer Zeile der folgenden Form:

```
Service-Name   Portnummer/Protokoll   Aliase  # kommentar
```

Im Folgenden ein kleiner Auszug aus der Datei:

```
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
```

Diese Datei ist weitgehend statisch, da diese Nummern größtenteils normiert sind. Freie Nummern können allerdings in lokalen Netzen vergeben werden.

13.2.4 networks - erreichbare Netze

Die Datei enthält Informationen über erreichbare Netze. Jeder Eintrag für ein Netz besteht aus einer Zeile der folgenden Form:

```
Netzname  Netznummer Aliase
```

Aliase sind alternative Namen für das Netz, die nur am lokalen System bekannt sind.

Beispiel

```
loopback          127
firma             132.45
```

Die Datei enthält einen Standardeintrag. *loopback* bezeichnet eine Netz-Schnittstelle für die lokale Kommunikation.

Weitere erreichbare Netze können von der Systembetreuung in die Datei eingetragen werden. Es ist zu beachten, dass ein Netz nur dann erreichbar ist, wenn das System Routing-Informationen über dieses Netz besitzt.

13.2.5 hosts - erreichbare Rechner

Diese Datei enthält Informationen über erreichbare (bekannte) Rechner. Jeder Eintrag besteht aus einer Zeile der folgenden Form:

```
Rechneradresse Rechnername Aliase
```

Die Datei enthält den Standardeintrag

```
127.0.0.1     localhost  local
```

Eintragungen durch die Systembetreuung sind hier nur notwendig, wenn eine Anwendung die Funktion *gethostent()* (siehe Seite 97) benutzt, um die erreichbaren Rechnernamen und Rechneradressen zu erfahren. Mit den Funktionen *gethostbyname(), gethostbyaddr(), getipnodebyname()* und *getipnodebyaddr()* (siehe Seite 97 und Seite 100) kann die Information über DNS oder BCAM ermittelt werden.

13.3 Abhängigkeiten vom BS2000/OSD-Transportsystem BCAM

Dieser Abschnitt beschreibt, was Sie bei SOCKETS(POSIX)- und XTI(POSIX)-Anwendungen im Hinblick auf das BS2000/OSD-Transportsystem BCAM beachten müssen. Nähere Informationen zu BCAM entnehmen Sie im Handbuch „BCAM V17.0A“ [6].

BCAM als Kommunikationsmanager von SOCKETS(POSIX) und XTI(POSIX)

BCAM als Basis des Datenkommunikationssystems für BS2000/OSD-Rechner unterstützt mehrere Kommunikationsarchitekturen. Socket- und XTI-Anwendungen können über die Protokolle TCP/IP und UDP/IP der Internet-Architektur kommunizieren. Das Kommunikationssystem wird mit BCAM-Administrationskommandos verwaltet. In diesem Zusammenhang von Bedeutung sind die BCAM-Kommandos BCIN (Endsysteme dynamisch generieren) und BCSHOW (Zustandsinformationen abfragen, z.B. Portbelegung). Anstelle von BCIN und BCSHOW können Sie auch die die entsprechenden SDF-Kommandos verwenden. Weitere für Socket- und XTI-Anwendungen relevante BCAM-Kommandos finden Sie unter „Relevante Einstellungen beim Transportsystem BCAM“ auf Seite 317.

Einem Socket bzw. Transportendpunkt eine spezielle Internet-Adresse zuordnen

Bei Einsatz von SOCKETS(POSIX) auf einem Rechner mit mehr als einem Internet-Anschluss wird das Binden eines Sockets auf eine spezielle Internet-Adresse unterstützt. Dazu muss unter Umständen die Option SO_REUSEADDR mit der Funktion *setsockopt()* gesetzt werden. Für Sockets vom Typ SOCK_DGRAM bzw. Client-Sockets vom Typ SOCK_STREAM kann bis einchließlich BCAM V17 das tatsächlich zum Senden von Datagrammen bzw. Verbindungsanforderungen verwendete Interface auf diese Weise jedoch nicht ausgewählt werden.

Abhängigkeit der IPv6-Unterstützung durch SOCKETS(POSIX) von BCAM und SOCKETS(BS2000)

Die Kommunikation mit SOCKETS(POSIX) in der Internet-Adressfamilie AF_INET6 (IPv6) setzt die BCAM-Version 16.0 voraus.

Die mit der IPv6-Unterstützung gemäß RFC 2553 neu eingeführten Funktionen *getaddrinfo()*, *freeaddrinfo()*, *gai_strerror()*, *getipnodebyaddr()*, *getipnodebyname()*, *freehostinfo()*, *getnameinfo()*, *inet_ntop()* und *inet_pton()* können nur auf Systemen mit BCAM V16.0 (oder höher) ausgeführt werden.

Auf Systemen mit älteren BCAM-Versionen liefern sie immer den Fehler EAFNOSUPPORT.

Die Funktionen *getaddrinfo()*, *getipnodebyaddr()*, *getipnodebyname()* und *getnameinfo()* verwenden (auch für IPv4-Adressen) den DNS-Resolver-Service des Subsystems SOCKETS(BS2000). Dazu ist mindestens die SOCKETS(BS2000)-Version V2.0 notwendig.

Relevante Einstellungen beim Transportsystem BCAM

Nachfolgend sind die BCAM-Kommandos angegeben, die Auswirkungen auf SOCKETS(POSIX)- und XTI(POSIX)-Anwendungen haben. Ausführlich beschrieben sind die BCAM-Kommandos sowie die zugehörigen Parameter im Handbuch „openNet Server V3.0 (BS2000/OSD)“.

BCAM-Kommandos DCSTART und BCMOD (BCAM-Grenzwerte)

Die Operanden MAXNPA, MAXNPT und MAXCNN beschränken die Anzahl der Netzwerkanwendungen und Verbindungen.

BCAM-Kommando BCTIMES (BCAM-Zeiteinstellungen)

Der Operand CONN beschränkt die Wartezeit für Verbindungsanforderungen.
Der Operand DATAGRAM begrenzt die Verweildauer von Nachrichten des verbindungslosen Transportdienstes.
Der Operand LETT begrenzt die Verweildauer von Nachrichten des verbindungsorientierten Transportdienstes.

BCAM-Kommando BCOPTION (BCAM-Betriebsoptionen)

Der Operand BROADCAST legt fest, ob der Rechner Broadcast-Nachrichten empfangen darf. Das Senden von Broadcast-Nachrichten ist nicht eingeschränkt.

BCAM-Kommandos BCMOD und DCOPT (DCSTART-Parameter vordefinieren/modifizieren)

Der Operand FREEPORT# gibt die erste freie Portnummer an, die von BCAM dynamisch für eine Anwendung belegt werden darf. PRIVPORT# gibt die erste Socket-Portnummer an, die von nicht-privilegierten und privilegierten Anwendungen belegt werden darf. FREEPORT# muss immer größer oder gleich PRIVPORT# sein.

Unterstützung des Domain Name Service (DNS)

SOCKETS(POSIX) unterstützt das DNS-Konzept (Domain Name Service), wenn das Subsystem SOCKETS(BS2000) V2.0 und/oder der DNS-Resolver aus dem Produkt *interNet* Services (früher TCP-IP-SV) konfiguriert und gestartet sind, siehe „interNet Services V3.0 (BS2000/OSD) Administratorhandbuch“.
Der Domain Name Service sammelt Informationen über die an ein Netz angeschlossenen Rechner und stellt diese Informationen allen Rechnern über das Netz zur Verfügung.

Wenn Sie BCAM ab V16.0 einsetzen, steht Ihnen folgende DNS-Funktionalität zur Verfügung:

– für *gethostbyname()*, *gethostbyaddr()*:
  DNS-Resolver-Funktionalität in *inter*Net Services (TCP-IP-SV)
  DNS-Resolver-Funktionalität in SOCKETS(BS2000)

– für *getipnodebyname()*, *getipnodebyaddr()*, *getaddrinfo()*, *getnameinfo()*:
  DNS-Resolver-Funktionalität in SOCKETS(BS2000)

14 Einschränkungen zur Kompatibilität

Kompatibilität zu UNIX-Anwendungen

Die Implementierung der SOCKETS(POSIX)-Schnittstelle basiert auf der Implementierung von SINIX V5.41 und wurde erweitert um die Unterstützung von IPv6. Damit ist gewährleistet, dass SOCKETS(POSIX)-Anwendungen zu UNIX-Anwendungen weitgehend source-kompatibel sind. Es gelten die folgenden Einschränkungen:

- Die RAW-Socket-Schnittstelle wird nicht unterstützt.
- Out-of-Band-Daten werden nicht unterstützt.
- Wartepunkte für alle blockierenden Operationen auf POSIX-Dateideskriptoren liegen im POSIX-Subsystem und sind damit dem Anwendungsprogrammierer entzogen.

Kompatibilität zu SOCKETS(BS2000)-Anwendungen

SOCKETS(BS2000)-Anwendungen sind nicht kompatibel zu Anwendungen, die mit den Funktionen von SOCKETS(POSIX) entwickelt wurden.

XTI-Kompatibilitäten

Für XTI gelten folgende Beschränkungen:

- Das Senden von Vorrangdaten (expedited data) wird nicht unterstützt.
- Es werden nur die Transportdienste für TCP/IP und UDP/IP unterstützt.
- Es gibt keine Unterstützung von IPv6.

Literatur

Die Handbücher sind online unter *http://manuals.fujitsu-siemens.com* zu finden oder in gedruckter Form gegen gesondertes Entgelt unter *http://FSC-manualshop.com* zu bestellen.

[1] **C-Bibliotheksfunktionen** (BS2000/OSD)
für POSIX-Anwendungen
Referenzhandbuch

Zielgruppe
C- und C++-Programmierer
Inhalt
Das Handbuch dokumentiert die XPG4-konforme C-Programmierschnittstelle, die vom POSIX-Subsystem im BS2000 unterstützt wird. Mit dieser Programmierschnittstelle kann sowohl auf das POSIX-Dateisystem als auch auf BS2000-Dateien zugegriffen werden. Zusätzlich enthält die Programmierschnittstelle Erweiterungen, die die Kompatibilität mit der bisherigen C-Bibliothek gewährleisten.

[2] **POSIX** (BS2000/OSD)
Kommandos
Benutzerhandbuch

Zielgruppe
Das Handbuch wendet sich an alle Benutzer der POSIX-Shell.
Inhalt
Dieses Handbuch ist ein Nachschlagewerk. Es beschreibt das Arbeiten mit der POSIX-Shell sowie die Kommandos der POSIX-Shell in alphabetischer Reihenfolge.

[3] **POSIX** (BS2000/OSD)
Grundlagen für Anwender und Systemverwalter
Benutzerhandbuch

Zielgruppe
BS2000-Systemverwalter, POSIX-Verwalter, BS2000-Benutzer,
Benutzer von UNIX-Workstations
Inhalt
– Einführung und Arbeiten mit POSIX
– BS2000-Softwareprodukte im Umfeld von POSIX
– POSIX installieren
– POSIX steuern und Dateisysteme verwalten
– POSIX-Benutzer verwalten
– BS2000-Kommandos für POSIX

[4] **Reliant UNIX V5.45**
Netzwerkschnittstellen
Programmiererhandbuch

Zielgruppe
Das Handbuch richtet sich an den Anwendungsprogrammierer, der Software für den Einsatz in Netzen entwickeln will.
Inhalt
Es beschreibt, wie Sie Anwendungsprogramme schreiben können, die die Netzwerkfunktionen (TLI, Sockets, RPC) von Reliant UNIX nutzen.

[5] **C/C++ V3.1A** (BS2000/OSD)
C/C++-Compiler
Benutzerhandbuch

Zielgruppe
C- und C++-Anwender im BS2000/OSD.
Inhalt
– Beschreibung aller Tätigkeiten zum Erzeugen von ablauffähigen C- und C++-Programmen: Übersetzen, Binden, Laden, Testen;
– Programmierhinweise und weitergehende Informationen zu: Optimierung, Programmablaufsteuerung, Funktions- und Sprachverknüpfung, C- und C++-Sprachunterstützung des Compilers.

[6] ***open*Net Server V3.0** (BS2000/OSD)
BCAM V17.0A
Benutzerhandbuch

Zielgruppe
Das Handbuch richtet sich an Netzplaner, -generierer und -verwalter, die in BS2000-Systemen BCAM betreiben.
Inhalt
Das Handbuch beschreibt BCAM selbst, seine Einbettung in TRANSDATA und TCP/IP- und ISO-Netze, sowie Generierungs- und Administrationstätigkeiten. Generierungsbeispiele verdeutlichen die Beschreibung. Es werden BCAM-Tools zur Generierung und Diagnose beschrieben. Anschließend beschreibt das Handbuch ausführlich die zur Generierung und zum Betrieb nötigen BCAM-Kommandos. Es werden die zur statischen Generierung nötigen KOGS-Makros vorgestellt und die BCAM-Fehlermeldungen aufgelistet.

[7] ***inter*Net Services V3.0** (BS2000/OSD)
Administratorhandbuch

Zielgruppe
Das Handbuch richtet sich an Netzplaner, -generierer und -verwalter, die in BS2000/OSD Internet Services betreiben wollen.
Inhalt
Das Handbuch beschreibt die Funktionalität der Internet Services
BOOTP/DHCP, TFTP, DNS, FTP, LDAP und NTP in BS2000/OSD. Weitere Themen sind Installation, Administration, Betrieb, Logging- und Diagnose-Möglichkeiten der einzelnen Komponenten, TLS/SSL-Unterstützung im FTP- und im TELNET-Server, FTP-Exit und TELNET-Exits sowie Zufallszahlen-Generierung in BS2000/OSD und POSIX.

[8] ***inter*Net Services V3.0** (BS2000/OSD)
Benutzerhandbuch

Zielgruppe
Das Handbuch richtet sich an Netzplaner, -generierer und -verwalter sowie Nutzer, die die Internet Services in Verbindung mit BS2000/OSD nutzen wollen.
Inhalt
Das Handbuch stellt die Komponenten von *inter*Net Services vor. Ausführlich werden die Nutzung von FTP, TELNET, der FTAC-Schnittstelle für FTP und TELNET sowie der Mailreader beschrieben. Ein weiteres wesentliches Thema des Handbuchs ist die TLS/SSL-Unterstützung von FTP und TELNET. Netzverwalter benötigen dieses Handbuch zusätzlich zum Administratorhandbuch.

[9] ***open*Net Server** (BS2000/OSD)
IPv6 Einführung und Umstellhandbuch Stufe 1
Benutzerhandbuch

Zielgruppe
Das Handbuch wendet sich an alle, die über die Einführung von IPv6 in BS2000/OSD entscheiden sowie an alle, die die IPv6-Funktionalität auf BS2000/OSD-Mainframes nutzen oder IPv6 in BS2000/OSD installieren wollen.
Inhalt
Das Handbuch informiert über die kommerziellen und technischen Grundlagen von IPv6. Darüber hinaus wird der Übergang von IPv4 nach IPv6 anhand von Beispielen erläutert und der aktuelle Stand der Implementierung von IPv6 in BS2000/OSD dargestellt. Detaillierte Informationen zu den Themen „IPv6-Adressierung“ und „DNS-Nutzung“ werden im Anhang des Handbuchs geliefert.

Sonstige Literatur

X/Open CAE Specification
Networking Services, Issue 4

Stichwörter

B

Z

Inhalt

POSIX

SOCKETS/XTI für POSIX
Benutzerhandbuch

Zielgruppe

C- und C++-Programmierer, die mit SOCKETS- bzw. XTI-Funktionen Kommunikationsanwendungen auf der Basis der POSIX-Schnittstelle entwickeln.

Inhalt

- Einführung in SOCKETS(POSIX)
- Benutzerfunktionen von SOCKETS(POSIX)
- Einführung in XTI(POSIX)
- XTI-Trace
- Bibliotheksfunktionen von XTI(POSIX)
- Übersetzen und Binden von Kommunikationsanwendungen
- Konfiguration und Konfigurationsdateien, BCAM-Abhängigkeiten
- Einschränkungen zur Kompatibilität

Ausgabe: März 2005

Datei: posix_s.pdf

Dieses Handbuch wurde erstellt von
cognitas. Gesellschaft für Technik-Dokumentation mbH
www.cognitas.de

Fujitsu Siemens Computers GmbH
Handbuchredaktion
81730 München

Kritik
Anregungen
Korrekturen

Fax: 0 700 / 372 00001

e-mail: manuals@fujitsu-siemens.com
http://manuals.fujitsu-siemens.com

Absender

Kommentar zu POSIX
SOCKETS/XTI für 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.