

Deutsch



Fujitsu Software BS2000

openNet Server V3.6

SOCKETS(BS2000) V2.7A

Benutzerhandbuch

Ausgabe April 2015

Kritik... Anregungen... Korrekturen...

Die Redaktion ist interessiert an Ihren Kommentaren zu diesem Handbuch. Ihre Rückmeldungen helfen uns, die Dokumentation zu optimieren und auf Ihre Wünsche und Bedürfnisse abzustimmen.

Sie können uns Ihre Kommentare per E-Mail an manuals@ts.fujitsu.com senden.

Zertifizierte Dokumentation nach DIN EN ISO 9001:2008

Um eine gleichbleibend hohe Qualität und Anwenderfreundlichkeit zu gewährleisten, wurde diese Dokumentation nach den Vorgaben eines Qualitätsmanagementsystems erstellt, welches die Forderungen der DIN EN ISO 9001:2008 erfüllt.

cognitas. Gesellschaft für Technik-Dokumentation mbH
www.cognitas.de

Copyright und Handelsmarken

Copyright © 2015 Fujitsu Technology Solutions GmbH.

Alle Rechte vorbehalten.

Liefermöglichkeiten und technische Änderungen vorbehalten.

Alle verwendeten Hard- und Softwarenamen sind Handelsnamen und/oder Warenzeichen der jeweiligen Hersteller.

Inhalt

| | | |
|------------|--|-----------|
| 1 | Einleitung | 9 |
| 1.1 | Kurzbeschreibung des Produkts | 9 |
| 1.2 | Zielsetzung und Zielgruppen des Handbuchs | 9 |
| 1.3 | Konzept des Handbuchs | 10 |
| 1.4 | Änderungen gegenüber dem Vorgänger-Handbuch | 12 |
| 1.5 | Darstellungsmittel | 13 |
| 1.6 | Kompatibilität von SOCKETS(BS2000) V2.7 zu früheren Versionen | 14 |
| 1.7 | Lizenzrechtliche Bestimmungen | 15 |
| 2 | Grundlagen von SOCKETS(BS2000) | 19 |
| 2.1 | Netzanbindung über die SOCKETS(BS2000)-Schnittstelle | 20 |
| 2.2 | Include-Dateien | 21 |
| 2.3 | Socket-Typen | 22 |
| 2.3.1 | Stream-Sockets (verbindungsorientiert) | 22 |
| 2.3.2 | Datagramm-Sockets (verbindungslos) | 23 |
| 2.3.3 | Raw-Sockets | 23 |
| 2.4 | Socket-Adressierung | 24 |
| 2.4.1 | Socket-Adressen verwenden | 24 |
| 2.4.2 | Adressierung mit Internet-Adresse | 24 |
| 2.4.2.1 | Adress-Struktur sockaddr_in der Adressfamilie AF_INET | 25 |
| 2.4.2.2 | Adress-Struktur sockaddr_in6 der Adressfamilie AF_INET6 | 26 |
| 2.4.3 | Adress-Struktur sockaddr_iso der Adressfamilie AF_ISO | 28 |
| 2.5 | Socket erzeugen | 29 |
| 2.6 | Einem Socket einen Namen zuordnen | 31 |
| 2.6.1 | Adresse explizit zuordnen | 31 |

| | | |
|-------------|--|-----------|
| 2.6.2 | Adresse mit Wildcards zuordnen (AF_INET, AF_INET6) | 33 |
| 2.6.3 | Direkte Adresszuordnung in den Domänen AF_INET und AF_INET6 | 36 |
| 2.7 | Kommunikation in den Domänen AF_INET und AF_INET6 | 37 |
| 2.7.1 | Verbindungsorientierte Kommunikation in AF_INET und AF_INET6 | 37 |
| 2.7.1.1 | Verbindungsanforderung durch den Client | 37 |
| 2.7.1.2 | Verbindungsannahme durch den Server | 38 |
| 2.7.1.3 | Datenübertragung bei verbindungsorientierter Kommunikation | 40 |
| 2.7.1.4 | Beispiele für eine verbindungsorientierte Client-/Server-Kommunikation | 41 |
| 2.7.2 | Verbindungslose Kommunikation in AF_INET und AF_INET6 | 44 |
| 2.7.2.1 | Datenübertragung bei verbindungsloser Kommunikation | 44 |
| 2.7.2.2 | Beispiele für eine verbindungslose Kommunikation | 45 |
| 2.8 | Kommunikation in der Domäne AF_ISO | 48 |
| 2.8.1 | Verbindungsanforderung durch den Client | 48 |
| 2.8.2 | Verbindungsannahme durch den Server | 49 |
| 2.8.3 | Datenübertragung bei verbindungsorientierter Kommunikation | 50 |
| 2.9 | Verbindung abbauen und Socket schließen | 51 |
| 2.9.1 | Verbindungsabbau in den Domänen AF_INET und AF_INET6 | 51 |
| 2.9.2 | Verbindungsabbau in der Domäne AF_ISO | 54 |
| 2.10 | Ein-/Ausgabe-Multiplexen | 55 |
| 2.10.1 | Ein-/Ausgabe-Multiplexen mit der Funktion select() | 55 |
| 2.10.2 | Ein-/Ausgabe-Multiplexen mit der Funktion soc_poll() | 59 |
| 2.11 | Zusammenspiel der Funktionen der SOCKETS-Schnittstelle | 62 |
| 2.11.1 | Zusammenspiel der Funktionen bei verbindungsorientierter Kommunikation | 62 |
| 2.11.2 | Zusammenspiel der Funktionen bei verbindungsloser Kommunikation | 64 |
| 3 | Adressumwandlung bei SOCKETS(BS2000) | 65 |
| 3.1 | Rechnernamen in Netzadressen umwandeln und umgekehrt | 66 |
| 3.2 | Protokollnamen in Protokollnummern umwandeln | 68 |
| 3.3 | Service-Namen in Portnummern umwandeln und umgekehrt | 69 |
| 3.4 | Byte-Reihenfolge umwandeln | 70 |
| 3.5 | Beispiel zur Adressumwandlung | 71 |
| 4 | Erweiterte Funktionen von SOCKETS(BS2000) | 73 |
| 4.1 | Nicht-blockierende Sockets | 74 |

| | | |
|------------|---|------------|
| 4.2 | Multicast-Nachrichten (AF_INET, AF_INET6) | 75 |
| 4.3 | Socket-Optionen | 77 |
| 4.4 | Unterstützung von virtuellen Hosts | 78 |
| 4.5 | Handoff (Verschieben eines Accept-Socket) | 80 |
| 4.5.1 | Allgemeine Beschreibung | 80 |
| 4.5.2 | Funktionsablauf | 80 |
| 4.6 | Raw-Sockets | 85 |
| 4.6.1 | ICMP | 85 |
| 4.6.2 | ICMPv6 | 86 |
| 5 | Client-/Server-Modell bei SOCKETS(BS2000) | 87 |
| 5.1 | Verbindungsorientierter Server | 88 |
| 5.1.1 | Verbindungsorientierter Server bei AF_INET / AF_INET6 | 88 |
| 5.1.2 | Verbindungsorientierter Server bei AF_ISO | 92 |
| 5.2 | Verbindungsorientierter Client | 96 |
| 5.2.1 | Verbindungsorientierter Client bei AF_INET / AF_INET6 | 96 |
| 5.2.2 | Verbindungsorientierter Client bei AF_ISO | 100 |
| 5.3 | Verbindungsloser Server | 104 |
| 5.4 | Verbindungsloser Client | 110 |
| 6 | Benutzerfunktionen von SOCKETS(BS2000) | 113 |
| 6.1 | Beschreibungsformat | 113 |
| | Funktionsname - Kurzbeschreibung der Funktionalität | 114 |
| 6.2 | Übersicht über die Funktionen | 115 |
| 6.3 | Beschreibung der Funktionen | 122 |
| | accept() - Eine Verbindung über einen Socket annehmen | 123 |
| | bind() - Einem Socket einen Namen zuordnen | 126 |
| | Byteorder-Makros - Byte-Reihenfolgen umsetzen | 128 |
| | connect() - Verbindung über einen Socket initiieren | 130 |
| | freeaddrinfo() - Speicher für addrinfo-Struktur freigeben | 133 |
| | freehostent() - Speicher für hostent-Struktur freigeben | 134 |
| | gai_strerror() - Textausgabe für den Error-Code von getaddrinfo() | 135 |
| | getaddrinfo() - Informationen über Rechnernamen, Rechneradressen und Services protokollunabhängig abfragen | 136 |
| | getbcamhost() - BCAM-Hostnamen abfragen | 141 |

| | |
|---|-----|
| getdtablesize() - Größe der Deskriptortabelle abfragen | 142 |
| gethostbyaddr(), gethostbyname() - Informationen über Rechnernamen und -adressen abfragen | 143 |
| gethostname() - Namen des Host abfragen | 145 |
| getipnodebyaddr(), getipnodebyname() - Informationen über Rechnernamen und Rechneradressen abfragen | 146 |
| getnameinfo() - Namen des Kommunikationspartners abfragen | 150 |
| getpeername() - Remote-Adresse der Sockets-Verbindung abfragen | 153 |
| getprotobyname() - Nummer des Protokolls abfragen | 155 |
| getservbyname(), getservbyport() - Informationen über Services abfragen | 157 |
| getsockname() - Local-Adresse der Sockets-Verbindung abfragen | 159 |
| getsockopt(), setsockopt() - Socket-Optionen abfragen und ändern | 161 |
| if_freenameindex() - Den durch if_nameindex() belegten dynamischen Speicher freigeben | 176 |
| if_indextoname() - Interface-Index auf Interface-Namen umsetzen | 177 |
| if_nameindex() - Liste von Interface-Namen mit dem dazugehörigen Interface-Index erzeugen | 178 |
| if_nametoindex() - Interface-Name auf Interface-Index umsetzen | 179 |
| inet_addr(), inet_lnaof(), inet_makeaddr(), inet_netof(), inet_network(), inet_ntoa() - IPv4-Internet-Adresse manipulieren | 180 |
| inet_ntop(), inet_pton() - Internet-Adressen manipulieren | 183 |
| listen() - Socket auf anstehende Verbindungen überprüfen | 185 |
| recv(), recvfrom() - Nachricht von einem Socket empfangen | 187 |
| recvmsg() - Nachricht von einem Socket empfangen | 190 |
| select() - Ein-/Ausgabe multiplexen | 194 |
| send(), sendto() - Nachricht von Socket zu Socket senden | 197 |
| sendmsg() - Nachricht von Socket zu Socket senden | 200 |
| shutdown() - Voll-Duplex-Verbindung abbauen | 204 |
| soc_close() (close) - Socket schließen | 206 |
| soc_eof(), soc_error(), soc_clearerr() (eof, error, clearerr) - Status-Information abfragen | 208 |
| soc_flush() (flush) - Daten aus Ausgabepuffer übertragen | 209 |
| soc_getc() (getc) - Zeichen aus dem Eingabepuffer lesen | 210 |
| soc_gets() (gets) - Character-String aus dem Eingabepuffer lesen | 211 |
| soc_ioctl() (ioctl) - Sockets steuern | 212 |
| soc_poll() - Ein-/Ausgabe multiplexen | 221 |
| soc_putc() (putc) - Zeichen in den Ausgabepuffer schreiben | 224 |
| soc_puts() (puts) - Character-String in den Ausgabepuffer schreiben | 225 |
| soc_read(), soc_readv() (read, readv) - Nachricht von einem Socket empfangen | 226 |
| soc_wake() - Eine mit select() oder soc-poll() wartende Task wecken | 228 |
| soc_write(), soc_writenv() (write, writev) - Nachricht von Socket zu Socket senden | 229 |
| socket() - Socket erzeugen | 232 |

| | | |
|------------|---|------------|
| 7 | SOCKETS(BS2000)-Schnittstelle für eine externe Börse | 235 |
| 7.1 | Beschreibung der Zusatzfunktionen | 236 |
| | setsockopt() - Socket-Optionen ändern | 236 |
| | soc_getevent() - Socket-Event abholen | 238 |
| 8 | Softwarepaket SOCKETS(BS2000) V2.7 | 241 |
| 8.1 | SOCKETS(BS2000)-Subsysteme | 241 |
| 8.2 | SOCKETS(BS2000)-Programme | 241 |
| 8.2.1 | ping4 | 241 |
| 8.2.2 | ping6 | 241 |
| 8.2.3 | nslookup | 242 |
| 8.3 | SOCKETS(BS2000)-DNS-Zugang | 243 |
| 8.4 | SOCKETS(BS2000) - Abfrage an FQDN-Datei | 244 |
| 8.5 | SOCKETS(BS2000)-Anwenderprogramm produzieren | 245 |
| 8.5.1 | Software-Voraussetzungen | 245 |
| 8.5.2 | Programmerstellung | 245 |
| | Literatur | 247 |
| | Stichwörter | 249 |

1 Einleitung

SOCKETS(BS2000) ist die Bezeichnung für die Socket-Funktionen innerhalb des BS2000. Diese Funktionen bieten die Entwicklungsumgebung für BS2000-Anwender, die Socket-Anwenderprogramme schreiben wollen.

1.1 Kurzbeschreibung des Produkts

Die Socket-Programmierung mit SOCKETS(BS2000) bietet eine Reihe von Möglichkeiten bei der Entwicklung von Kommunikationsanwendungen. SOCKETS(BS2000) ist eine der Schnittstellen zur Netzwerkprogrammierung innerhalb des BS2000. Damit können Kommunikationsanwendungen auf Basis der TCP/IP-Protokolle sowie der OSI-Protokolle entwickelt werden.

1.2 Zielsetzung und Zielgruppen des Handbuchs

Das vorliegende Handbuch wendet sich an Programmierer, die mit den Funktionen der SOCKETS(BS2000)-Schnittstelle Kommunikationsanwendungen entwickeln wollen. Kenntnisse der C-Programmierung werden vorausgesetzt.

1.3 Konzept des Handbuchs

Im vorliegenden Handbuch sind die verschiedenen Möglichkeiten der Socket-Programmierung beschrieben und anhand einiger einfacher Beispiele erläutert. Die Beispielprogramme zeigen die Verwendung von Socket-Funktionen sowohl für verbindungsorientierte Kommunikationsanwendungen über das TCP-Protokoll und den ISO-Service 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(BS2000)-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(BS2000)-Schnittstelle.
- Kapitel 7 beschreibt die Zusatz-Funktionen der Socket-Schnittstelle für BS2000 im Spezialmodus mit Nutzung einer externen Börse
- Kapitel 8 enthält Informationen zu folgenden Themen:
 - SOCKETS(BS2000)-Subsysteme
 - SOCKETS(BS2000)-Programme
 - Produktion eines SOCKETS(BS2000)-Anwenderprogramms mit den dazugehörigen Software-Voraussetzungen

Readme-Datei

Funktionelle Änderungen der aktuellen Produktversion und Nachträge zu diesem Handbuch entnehmen Sie bitte ggf. der produktspezifischen Readme-Datei.

Readme-Dateien stehen Ihnen online bei dem jeweiligen Produkt zusätzlich zu den Produkthandbüchern unter <http://manuals.ts.fujitsu.com> zur Verfügung. Alternativ finden Sie Readme-Dateien auch auf der Softbook-DVD.

Informationen unter BS2000

Wenn für eine Produktversion eine Readme-Datei existiert, finden Sie im BS2000-System die folgende Datei:

```
SYSRME.<product>.<version>.<lang>
```

Diese Datei enthält eine kurze Information zur Readme-Datei in deutscher oder englischer Sprache (<lang>=D/E). Die Information können Sie am Bildschirm mit dem Kommando `/SHOW-FILE` oder mit einem Editor ansehen.

Das Kommando `/SHOW-INSTALLATION-PATH INSTALLATION-UNIT=<product>` zeigt, unter welcher Benutzerkennung die Dateien des Produkts abgelegt sind.

Ergänzende Produkt-Informationen

Aktuelle Informationen, Versions-, Hardware-Abhängigkeiten und Hinweise für Installation und Einsatz einer Produktversion enthält die zugehörige Freigabemitteilung. Solche Freigabemitteilungen finden Sie online unter <http://manuals.ts.fujitsu.com>.

1.4 Änderungen gegenüber dem Vorgänger-Handbuch

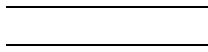
- Der [Abschnitt „Lizenzrechtliche Bestimmungen“](#) wurde hinzugefügt.
- Ergänzung von *getsockopt()* durch die Subfunktion SO_KEEPALIVE.
- Ergänzung von *sock_ioctl()* durch die Subfunktionen SIOCGIFNETMASK, SIOCGLIFBRDADDR, SIOCGLIFHWADDR, SIOCGLIFNETMASK.
- Neues Diagnose Tool NSLOOKUP.
- Neue Interface Control Flags IFF_CONTROLLAN, IFF_AUTOCONFIG.
- Streichung des Abschnitts „Broadcast-Nachrichten (AF_INET)“ im [Kapitel „Erweiterte Funktionen von SOCKETS\(BS2000\)“](#).
- Zusätzliche Events bei *sock_getevent()*.
- Erweiterte Möglichkeiten für den DNS-Zugang, siehe [Abschnitt „SOCKETS\(BS2000\)-DNS-Zugang“](#).

1.5 Darstellungsmittel

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
kursive Schrift

Programmtext in Beispielen; Syntaxdarstellungen

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 vorangehende 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 des [Kapitel „Benutzerfunktionen von SOCKETS\(BS2000\)“ auf Seite 113](#) 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 Kompatibilität von SOCKETS(BS2000) V2.7 zu früheren Versionen

SOCKETS(BS2000) V2.7 ist kompatibel zu SOCKETS(BS2000) V2.6, 2.5, 2.4, 2.3, 2.2, V2.1 und V2.0, d.h. schon bestehende Socket-Anwenderprogramme sind in der Version 2.7 ablauffähig.

Wenn ein Socket-Anwenderprogramm mit der Anwenderbibliothek SYSLIB.SOCKETS.027 produziert wurde, dann wird die Meldung „unresolved extern“ in folgenden Fällen ausgegeben:

- wenn es die neuen Funktionen *soc_poll()*, *if_nametoindex()*, *if_indextoname()*, *if_nameindex()* und *if_freenameindex()* nutzt und auf ein Subsystem SOCKETS(BS2000) V2.0 trifft.
- wenn es die neuen Funktionen *if_nametoindex()*, *if_indextoname()*, *if_nameindex()*, *if_freenameindex()* nutzt und auf ein Subsystem SOCKETS(BS2000) V2.1 trifft.

1.7 Lizenzrechtliche Bestimmungen

SOCKETS(BS2000) verwendet für die DNS-Anbindung Teile der Open Source Software *bind* und für das Ping/Ping6-Programm die Open Source Software *iputils*.

Nachfolgend sind die entsprechenden Lizenztexte abgedruckt.

```
/*
 *
 * Modified for AF_INET6 by Pedro Roque
 *
 * <roque@di.fc.ul.pt>
 *
 * Original copyright notice included bellow
 */

/*
 * Copyright (c) 1989 The Regents of the University of California.
 * All rights reserved.
 *
 * This code is derived from software contributed to Berkeley by
 * Mike Muuss.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 * notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 * notice, this list of conditions and the following disclaimer in the
 * documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 * must display the following acknowledgement:
 * This product includes software developed by the University of
 * California, Berkeley and its contributors.
 * 4. Neither the name of the University nor the names of its contributors
 * may be used to endorse or promote products derived from this software
 * without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
```

```
* OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.
*/
```

```
#ifndef lint
char copyright[] =
"@(#) Copyright (c) 1989 The Regents of the University of California.\n\
  All rights reserved.\n";
#endif /* not lint */
```

```
/*
* P I N G . C
*
* Using the InterNet Control Message Protocol (ICMP) "ECHO" facility,
* measure round-trip-delays and packet loss across network paths.
*
* Author -
*   Mike Muuss
*   U. S. Army Ballistic Research Laboratory
*   December, 1983
*
* Status -
*   Public Domain.  Distribution Unlimited.
```

```
License-Text arpa.nameser.h
```

```
/*
* Copyright (c) 1983, 1989, 1993
*   The Regents of the University of California.  All rights reserved.
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
* 1. Redistributions of source code must retain the above copyright
*   notice, this list of conditions and the following disclaimer.
* 2. Redistributions in binary form must reproduce the above copyright
*   notice, this list of conditions and the following disclaimer in the
*   documentation and/or other materials provided with the distribution.
* 3. All advertising materials mentioning features or use of this software
*   must display the following acknowledgement:
*   This product includes software developed by the University of
*   California, Berkeley and its contributors.
* 4. Neither the name of the University nor the names of its contributors
*   may be used to endorse or promote products derived from this software
*   without specific prior written permission.
*
* THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
* ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
```


* ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
* FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
* OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.

*/

/*

* Copyright (c) 1996 by Internet Software Consortium.

*

* Permission to use, copy, modify, and distribute this software for any
* purpose with or without fee is hereby granted, provided that the above
* copyright notice and this permission notice appear in all copies.

*

* THE SOFTWARE IS PROVIDED "AS IS" AND INTERNET SOFTWARE CONSORTIUM
DISCLAIMS

* ALL WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED
WARRANTIES

* OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL INTERNET SOFTWARE
* CONSORTIUM BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL
* DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR
* PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
* ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF

THIS

* SOFTWARE.

*/

/*

* From: Id: nameser.h,v 8.16 1998/02/06 00:35:58 halley Exp

*

* \$Id: nameser.h,v 1.12 1998/06/11 08:55:15 peter Exp \$

*/

License-Text lwres-client

/*

* Copyright (C) 2004, 2005 Internet Systems Consortium, Inc. ("ISC")

* Copyright (C) 2000, 2001 Internet Software Consortium.

*

* Permission to use, copy, modify, and distribute this software for any
* purpose with or without fee is hereby granted, provided that the above
* copyright notice and this permission notice appear in all copies.

*

* THE SOFTWARE IS PROVIDED "AS IS" AND ISC DISCLAIMS ALL WARRANTIES WITH
* REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
MERCHANTABILITY

* AND FITNESS. IN NO EVENT SHALL ISC BE LIABLE FOR ANY SPECIAL, DIRECT,
* INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING
FROM
* LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE
* OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
* PERFORMANCE OF THIS SOFTWARE.
*/

/* \$Id: context.h,v 1.15.18.2 2005/04/29 00:17:21 marka Exp \$ */

2 Grundlagen von SOCKETS(BS2000)

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\(BS2000\)“](#) auf [Seite 87](#) zusammengefasst. Die einzelnen Funktionen der SOCKETS(BS2000)-Schnittstelle sind im [Kapitel „Benutzerfunktionen von SOCKETS\(BS2000\)“](#) ab [Seite 113](#) ausführlich beschrieben.

2.1 Netzanbindung über die SOCKETS(BS2000)-Schnittstelle

Die SOCKETS(BS2000)-Schnittstelle ist eine der Schnittstellen zur Netzwerkprogrammierung innerhalb des BS2000. Damit entwickeln Sie Kommunikationsanwendungen auf der Basis der TCP/IP- und OSI-Servicedefinitionen.

Die SOCKETS(BS2000)-Schnittstelle ist in eigenen Include-Dateien definiert.

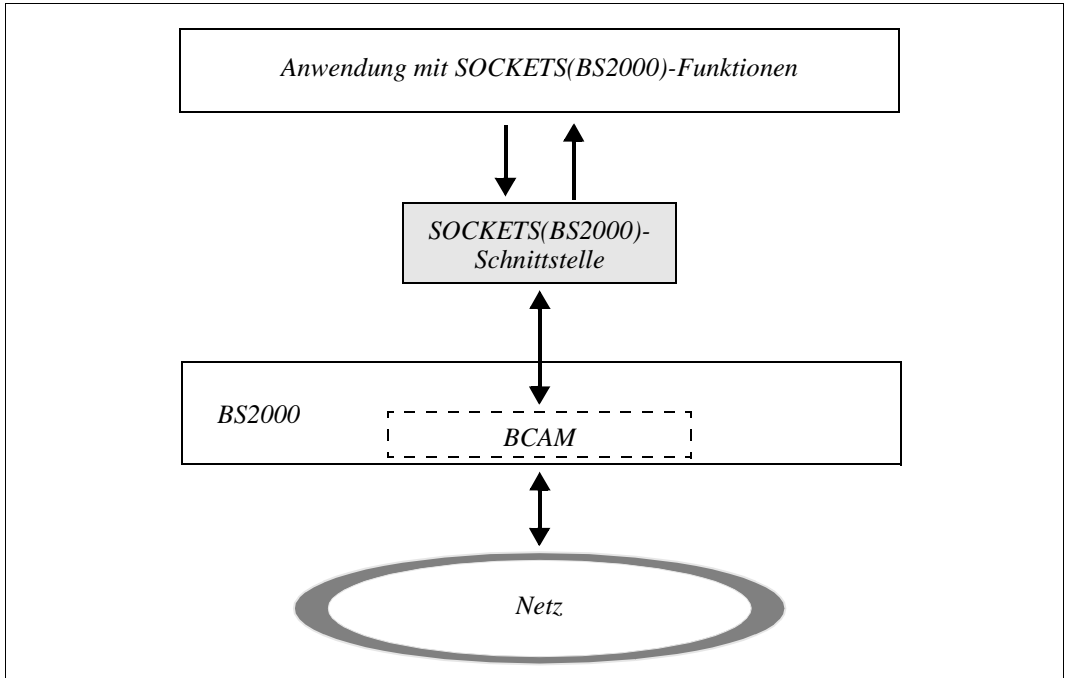


Bild 1: Sockets in BS2000

2.2 Include-Dateien

Bei der Installation von SOCKETS(BS2000) werden BSD V4.2- und RFC 2553-konforme Include-Dateien angelegt. Im [Kapitel „Benutzerfunktionen von SOCKETS\(BS2000\)“](#) ab [Seite 113](#) ist bei der Beschreibung jeder Socket-Funktion angegeben, welche Include-Datei(en) eine Anwendung für die Ausführung dieser Funktion einbinden muss.

SOCKETS(BS2000) 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>`

ioctl.h

Definitionen für die von der Socket-Funktion *soc_ioctl()* aufgerufenen Socket-Steuerfunktionen

iso.h

Definition der Adress-Struktur für die Adressfamilie AF_ISO

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.poll.h

Definitionen für die Funktion *soc_poll()*

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

timeval-Struktur für *select()* und Subfunktion *linger*

sys.uio.h

Datenstruktur *msghdr* für die Datenübertragung in Einzelpaketen bei *sendmsg()* und *recvmsg()*

2.3 Socket-Typen

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

Jeder Socket hat einen bestimmten Typ und gehört einer Task an. Zu einer Task kann es mehrere Sockets geben.

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 Tasks, die über Sockets kommunizieren. Die Socket-Schnittstelle im BS2000 unterstützt die Internet-Kommunikationsdomänen AF_INET und AF_INET6 sowie die ISO-Kommunikationsdomäne AF_ISO.

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

- Stream-Sockets
- Datagramm-Sockets
- Raw-Sockets

2.3.1 Stream-Sockets (verbindungsorientiert)

Stream-Sockets unterstützen die verbindungsorientierte Kommunikation. 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.

Verbindungsorientierte Übertragung in der Kommunikationsdomäne AF_INET und AF_INET6

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.

Verbindungsorientierte Übertragung in der Kommunikationsdomäne AF_ISO

Die verbindungsorientierte Kommunikation wird in AF_ISO satzorientiert abgewickelt, d.h. die Satzgrenzen bleiben erhalten. Basis der Kommunikationsanwendungen ist der ISO-Transportservice.

2.3.2 Datagramm-Sockets (verbindungslos)

Datagramm-Sockets unterstützen die verbindungslose Kommunikation in den Adressfamilien AF_INET und AF_INET6. 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 oder verloren gehen. Eine Task, die 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.

2.3.3 Raw-Sockets

Raw-Sockets bieten die Möglichkeit, auf Protokoll-Header-Ebene Daten zu schreiben und zu lesen. SOCKETS(BS2000) lässt dies für das **I**nternet **C**ontrol **M**essage **P**rotocol (ICMP), bzw. für das **I**nternet **C**ontrol **M**essage **P**rotocol for **I**pv6 (ICMPv6) zu. Damit ist es einer Sockets-Anwendung mit einem Raw-Socket möglich, einen ICMP-/ICMPv6-Echo-Request zu erzeugen und einen ICMP-/ICMPv6-Echo-Reply zu empfangen.

2.4 Socket-Adressierung

Ein Socket wird zunächst ohne Namen bzw. Adresse erzeugt. Damit Tasks 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 31](#)). Danach können über den Socket Nachrichten empfangen werden.

2.4.1 Socket-Adressen verwenden

Beim Aufruf der Funktionen *bind()*, *connect()*, *getpeername()*, *getsockname()*, *recvfrom()* und *sendto()* 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.

Wenn zunächst aus der Adress-Struktur die Adressfamilie ermittelt werden muss, um danach adressfamilien-spezifisch weiterarbeiten zu können, wird die allgemeine *sockaddr*-Struktur verwendet.

Die *sockaddr*-Struktur ist wie folgt in der Include-Datei `<sys.socket.h>` definiert.

```
struct sockaddr {
    u_short  sa_family; /* Adressfamilie */
    char     sa_data[50]; /* 50 byte für die längste Adresse (sockaddr_iso) */
};
```

In den beiden folgenden Abschnitten werden die Adress-Strukturen für die Adressfamilien `AF_INET` und `AF_INET6` sowie für die Adressfamilie `AF_ISO` beschrieben.

2.4.2 Adressierung mit Internet-Adresse

Bei SOCKETS(BS2000) 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 „[BCAM Band 1/2](#)“ beschrieben.

2.4.2.1 Adress-Struktur `sockaddr_in` in der Adressfamilie `AF_INET`

Bei der Adressfamilie `AF_INET` besteht ein Name aus einer 4 byte langen Internet-Adresse und einer Portnummer. Für die Adressfamilie `AF_INET` verwenden Sie die Adress-Struktur `sockaddr_in`. Diese Struktur besitzt eine Variante für `#define SIN_LEN`.

Die Struktur `sockaddr_in` ist in der Include-Datei `<netinet.in.h>` wie folgt deklariert:

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

Strukturvariante von `sockaddr_in` mit gesetztem `#define SIN_LEN` zur Unterstützung von BSD 4.4 Systemen:

```
struct sockaddr_in {
    u_char      sin_len;     /* Länge der Struktur*/
    u_char      sin_family;  /* Adressfamilie AF_INET */
    u_short     sin_port;    /* 16 bit Portnummer */
    struct in_addr sin_addr; /* 32 bit Internet-Adresse */
    char        sin_zero[8];
};
```

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

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

Die Strukturen für Rechner-, Protokoll- und Service-Namen sind im [Kapitel „Adressumwandlung bei SOCKETS\(BS2000\)“](#) ab [Seite 65](#) dargestellt.

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`. Diese Struktur besitzt zusätzliche Varianten für `#define SCOPE_ID` und `#define SIN6_LEN`.

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

```
struct sockaddr_in6 {
    short          sin6_family;          /* Adressfamilie AF_INET6 */
    u_short        sin6_port;           /* 16 bit Portnummer */
    u_int          sin6_flowinfo
    struct in6_addr sin6_addr;          /* IPv6-Adresse */
    char           sin6_zero[8];
};
```

Strukturvariante von `sockaddr_in6` mit gesetztem `#define SCOPE_ID` zur Unterstützung von Open Source :

```
struct sockaddr_in6 {
    short          sin6_family;          /* Adressfamilie AF_INET6 */
    u_short        sin6_port;           /* 16 bit Portnummer */
    u_int          sin6_flowinfo
    struct in6_addr sin6_addr;          /* IPv6-Adresse */
    u_int32_t      sin6_scope_id;
};
```

Strukturvariante von `sockaddr_in6` mit gesetztem `#define SIN6_LEN` zur Unterstützung von BSD 4.4 Systemen:

```
struct sockaddr_in6 {
    u_int8_t      sin6_len;             /* Länge der Struktur */
    sa_family_t   sin6_family;         /* Adressfamilie AF_INET6 */
    in_port_t     sin6_port;           /* 16 bit Portnummer */
    u_int32_t     sin6_flowinfo
    struct in6_addr sin6_addr;          /* IPv6-Adresse */
    u_int32_t     sin6_scope_id;
};
```

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

```
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 */
```

Speicherplatzzuweisung

Eine Speicherplatzzuweisung mit der dazugehörigen Initialisierung für die Variable *in6addr_any* muss im Code der Anwendung erfolgen. In der Include-Datei <netinet.in.h> wird die folgende Deklaration zur Verfügung gestellt:

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

in6addr_any hat den Wert ::0. In <netinet.in.h> ist eine entsprechende Konstante `IN6ADDR_ANY_INIT` definiert.

2.4.3 Adress-Struktur `sockaddr_iso` der Adressfamilie `AF_ISO`

Bei der Adressfamilie `AF_ISO` besteht ein Name aus einem Netzselektor `NSEL` und einem Transportselektor `TSEL`. Für die Adressfamilie `AF_ISO` verwenden Sie die Adress-Struktur `sockaddr_iso`.

Die Struktur `sockaddr_iso` ist in der Include-Datei `<iso.h>` wie folgt deklariert:

```
struct sockaddr_iso {
    u_char siso_len;           /* Länge dieser Struktur sockaddr_iso */
    u_char siso_family;       /* Adressfamilie AF_ISO */
    u_char siso_plen;         /* Präsentationselektorlänge */
                               /* (wird nicht unterstützt; default: 0) */
    u_char siso_sleng;        /* Sessionselektorlänge */
                               /* (wird nicht unterstützt; default: 0) */
    u_char siso_tlen;         /* Transportselektorlänge */
    struct iso_addr siso_addr; /* ISO-Anwendungsadresse */
    u_char siso_pad[6];       /* wird nicht unterstützt */
};

struct iso_addr {
    u_char isoa_len;          /* wird nicht unterstützt */
    char isoa_genaddr[40];    /* komplette Adresse ( NSEL/TSEL ) */
};
```

Das Kommunikationssystem für BS2000 erwartet als `NSEL` einen BCAM-Hostnamen. Der BCAM-Hostname hat eine feste Länge von 8 Zeichen, wobei am Namensende auch Leerzeichen zulässig sind. Mit Leerzeichen muss aufgefüllt werden, um die geforderte Länge von 8 Zeichen für `NSEL` zu erreichen. Der Transportselektor `TSEL` darf maximal 32 byte lang sein. Auf Grund der festen Längenvorgabe von `NSEL` ist es möglich, aus `isao_genaddr` den Transportselektor mit Hilfe der Transportselektorlänge `siso_tlen` zu selektieren.

BCAM-Hostname:

Er ist acht Zeichen lang. Es dürfen alphanumerische Zeichen und die Sonderzeichen `#`, `@`, `$` oder Leerzeichen am Namensende verwendet werden. In der Regel sollten Großbuchstaben verwendet werden, es wird aber Groß-/Kleinschreibung unterschieden. Ein nur numerischer Anteil ist nicht erlaubt.

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
- ISO-Kommunikationsdomäne AF_ISO

Für *domain* geben Sie deshalb AF_INET, AF_INET6 oder AF_ISO 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.
- Wenn Sie eine ICMP-Nachricht über einen Raw-Socket senden wollen, geben Sie SOCK_RAW für *type* an.

Der Parameter *protocol* wird nicht unterstützt und sollte den Wert 0 haben.

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 AF_INET:

```
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.

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.

Socket in der Domäne AF_ISO erzeugen

Der folgende Aufruf erzeugt einen Socket in der ISO-Domäne zur Nutzung des ISO-Transportservice:

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

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, zugeordnet werden. Erst dann können Partner den Socket adressieren, und der Socket-Nutzer kann Verbindungen aufbauen bzw. Daten senden und empfangen. Mit der Funktion `bind()` binden Sie einen Namen an den Socket, d.h. Sie ordnen dem Socket eine lokale Adresse zu.

2.6.1 Adresse explizit zuordnen

In diesem Fall rufen Sie `bind()` wie folgt auf:

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

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 25](#)). `namelen` enthält die Länge der Datenstruktur, die den Namen beschreibt.

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 26](#)). `namelen` enthält die Länge der Datenstruktur, die den Namen beschreibt.

In der Kommunikationsdomäne `AF_ISO` besteht `name` aus einem Netzselektor und einem Transportselektor. `name` wird übergeben in einer Variablen vom Typ `struct sockaddr_iso` (siehe [Seite 28](#)). `namelen` enthält die Länge der Datenstruktur, die den Namen beschreibt.

Adresse explizit zuordnen in den Domänen `AF_INET` und `AF_INET6`

Adresse explizit zuordnen in `AF_INET`

Der folgende Programmausschnitt skizziert, wie einem Socket in der Domäne `AF_INET` ein Name zugeordnet wird.

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

Adresse explizit zuordnen in AF_INET6

```

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

```

Bei der Wahl der Portnummer ist zu beachten:

- Portnummern kleiner als PRIVPORT# (siehe Handbuch „[BCAM Band 1/2](#)“) sind für privilegierte Anwendungen reserviert (Standardwert: 2050).
- Für einige Standardanwendungen gibt es feste Reservierungen bestimmter Portnummern:
 - Die Portnummer 3161 des SNMP-Agenten SNMP-Basic-Agent BS2000 wird verwendet für die interne Kommunikation zwischen dem Masteragent und den Subagenten (siehe Handbuch „[SNMP Management für BS2000](#)“).
 - Die Portnummer 1235 wird vom Domain Name Service (DNS) benötigt (siehe das Administrator-Handbuch zu „[interNet Services](#)“).
 - Zu beachten sind weitere „Well Known, Registered, Dynamic und/oder Private“ Portnummern, wie sie auf der Internet-Seite der IANA unter der Webadresse „<http://www.iana.org/assignments/port-numbers>“ veröffentlicht werden.

Adresse explizit zuordnen in der Domäne AF_ISO

Der folgende Programmausschnitt skizziert, wie einem Socket in der Domäne AF_ISO ein Name zugeordnet wird.

```

#include <sys.types.h>
#include <iso.h>
... ..
struct sockaddr_iso sin;
int s;
... ..
/* Hier müssen die Anweisungen stehen, die sin
mit dem Netzselektor und Transportselektor versorgen.*/
... ..
bind(s, &sin, sizeof sin);

```


2.6.2 Adresse 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. 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, bedeutet dies:

- 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 bzw. 3FFE:1:1000:1000:52C1:D5FF:FE0E:2B01, so kann eine Task, welcher der Socket *s* zugeordnet ist, Verbindungsanforderungen annehmen, die an 128.32.0.4 und 3FFE:1:1000:1000:52C1:D5FF:FE0E:2B01 adressiert sind.

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 MYPOR 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(MYPOR);
bind(s, &sin, sizeof sin);
```

Bei AF_INET6:

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

Automatische Adresszuordnung durch das System

Auch wenn einem Socket noch keine Adresse zugeordnet ist, können Sie für den Socket eine Funktion aufrufen, die eigentlich 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, d.h. der Socket wird mit *INADDR_ANY* auf alle IPv4-Adressen und mit *IN6ADDR_ANY* auf alle IPv6-Adressen und IPv4-Adressen des Rechners gebunden und erhält eine Portnummer aus einem freien Bereich.

2.6.3 Direkte Adresszuordnung in den Domänen AF_INET und AF_INET6

Ab der Version 2.2 ist es möglich, einen Socket selektiv auf eine ausgewählte Interface-Adresse zu binden (Multihoming Support). Dabei ist zu beachten, dass die gewünschte Adresse am betroffenen Rechner vorhanden sein muss und dass das Tupel IP-Adresse/Portnummer nicht belegt sein darf.

Dadurch ist es möglich, gezielt einen Listen-Socket auf einer Interface-Adresse und einem Port horchen zu lassen. Außerdem können für einen Port mehrere Listen-Sockets auf jeweils eine Interface-Adresse gebunden werden.

Um von einer Single-Adressierung auf eine Anyaddr-Adressierung oder umgekehrt wechseln zu können, wurde die Funktionalität der *setsockopt()*-Subfunktion *SO_REUSEADDR* erweitert. Ist der Socket vor dem *bind()* mit dieser Subfunktion markiert, dann ist es möglich, soweit das Transportsystem es zulassen kann, einen Socket auf eine Interface-Adresse zu binden, obwohl für diesen Port bereits ein Socket auf Anyaddr gebunden wurde. Das gilt auch für den umgekehrten Fall.

2.7 Kommunikation in den Domänen AF_INET und AF_INET6

In den Kommunikationsdomänen AF_INET und AF_INET6 wird zwischen verbindungsorientierter und verbindungsloser Kommunikation unterschieden.

2.7.1 Verbindungsorientierte Kommunikation in AF_INET und AF_INET6

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 Namens-Zuordnung beim Verbindungsaufbau.

Der Verbindungsaufbau zwischen zwei Sockets verläuft in der Regel asymmetrisch, wobei ein Socket die Rolle des Clients und der andere Socket die Rolle des Servers übernimmt.

2.7.1.1 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, &server, sizeof server);
```

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

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

Mit dem Parameter *server* werden die Internet-Adresse und die Portnummer des Servers bestimmt, mit dem der Client kommunizieren möchte.

Falls dem Socket des Clients 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 nachfolgender Abschnitt „[Verbindungsannahme durch den Server](#)“). Jedoch bleiben auch bei nicht erfolgreichem Verbindungsaufbau alle Namen erhalten, die vom System automatisch zugeordnet wurden.

2.7.1.2 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.

listen() veranlasst SOCKETS(BS2000) außerdem, die an den betreffenden Socket gerichteten 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;
int s, fromlen, newsock;
...
listen(s, 5);
fromlen = sizeof(from);
newsock = accept(s, &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, &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 die Server-Task warten können. Allerdings wertet SOCKETS(BS2000) diesen Parameter derzeit nicht aus, sondern nimmt Verbindungsanforderungen solange an, bis die maximale Anzahl verfügbarer Sockets belegt ist.

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 *accept()* solange nicht zurückkehrt, bis eine Verbindung angenommen ist. Außerdem hat die Server-Task beim Aufruf von *accept()* keine Möglichkeit, anzuzeigen, dass sie Verbindungswünsche nur von einem oder mehreren bestimmten Partnern annehmen möchte. Deshalb muss die Server-Task darauf achten, woher die Verbindung kommt. Die Server-Task muss die Verbindung beenden, wenn sie nicht mit einem bestimmten Client kommunizieren möchte.

Im [Kapitel „Erweiterte Funktionen von SOCKETS\(BS2000\)“ auf Seite 73](#) ist näher beschrieben,

- wie eine Server-Task auf mehr als einem Socket Verbindungen annehmen kann,
- wie eine Server-Task verhindern kann, dass der Aufruf von *accept()* blockiert.

2.7.1.3 Datenübertragung bei verbindungsorientierter Kommunikation

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

Es gibt mehrere Funktionen zum Senden und Empfangen von Daten:

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

Diese Socket-Funktionen sind ausführlich beschrieben im [Abschnitt „Beschreibung der Funktionen“ ab Seite 122](#).

2.7.1.4 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.

Beispiel 1: Initiieren einer Streams-Verbindung durch den Client

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

/*
 * Dieses Programm erzeugt einen Socket und initiiert eine Verbindung mit
 * der in der Kommandozeile übergebenen Internet-Adresse.
 * Über die Verbindung wird eine Nachricht gesendet.
 * Dann wird der Socket geschlossen und die Verbindung beendet.
 * Das Client-Programm erwartet die Eingabe eines Rechnernamens und
 * einer Portnummer. Es ist er Rechner, auf dem das Server-Programm läuft und
 * die Portnummer des listen-sockets des Server-Programms (im Beispiel die
 * Portnummer 2222).
 */

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");
    soc_close(sock);
    exit(0);
}

```

Beispiel 2: Annehmen der Streams-Verbindung durch den Server

```

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

/*
 * 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);
}

/* 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");
        else if (rval == 0)
            printf("Ending connection\n");
        else
            printf("-->%s\n", buf); }
    while (rval > 0);
    soc_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 eine Task abgebrochen wird oder ihr normales Ende erreicht.
 */

exit(0);
}
```

2.7.2 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. Anders als bei verbindungsorientierten Anwendungen, 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 29](#) beschrieben. Wenn eine bestimmte lokale Adresse benötigt wird, muss vor der ersten Datenübertragung die Funktion `bind()` aufgerufen werden (siehe [Seite 31](#)). Andernfalls ordnet das System die lokale Internet-Adresse und/oder die Portnummer zu, wenn zum ersten Mal Daten gesendet werden (siehe [Seite 35](#)).

2.7.2.1 Datenübertragung bei verbindungsloser Kommunikation

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

```
sendto(s, buf, buflen, flags, &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 Verwendung einer Datagramm-Schnittstelle ist zu beachten, dass es keine gesicherte Datenübertragung gibt. Somit kann ein `sendto()`-Aufruf nur dann eine Fehlerinformation zurückliefern, wenn dem lokalen System bekannt ist, dass die Nachricht nicht übertragen werden konnte.

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

```
recvfrom(s, buf, buflen, flags, &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 die Task mit *sendto()* ohne explizite Angabe einer Zieladresse abschickt, werden automatisch an den Partner mit der im *connect()*-Aufruf angegebenen Zieladresse gesendet.
- Eine Benutzer-Task 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.

2.7.2.2 Beispiele für eine verbindungslose Kommunikation

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

Beispiel 1 : Datagramme empfangen

```
#include <sys.types.h>
#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, &name, sizeof name) < 0) {
        perror("binding datagram socket");
        exit(1);
    }

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

```

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.
 */

```

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

2.8 Kommunikation in der Domäne AF_ISO

In der Domäne AF_ISO wird nur die verbindungsorientierte Kommunikation unterstützt. Miteinander kommunizierende Sockets werden über eine Zuordnung miteinander verbunden. In AF_ISO besteht eine Zuordnung aus einem lokalen Netzselektor und einem lokalen Transportselektor sowie einem fernen Netzselektor und einem fernen Transportselektor:

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

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

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

Die Aufrufe der nachfolgend in diesem Abschnitt vorgestellten Funktionen *connect()* und *accept()* vervollständigen die Namens-Zuordnung beim Verbindungsaufbau.

Der Verbindungsaufbau zwischen zwei Sockets verläuft in der Regel asymmetrisch, wobei ein Socket die Rolle des Clients und der andere Socket die Rolle des Servers übernimmt.

Beispiele zur Kommunikation in der Domäne AF_ISO finden Sie auf [Seite 92](#) (Server-Beispiel) und [Seite 100](#) (Client-Beispiel).

2.8.1 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 ISO-Domäne (AF_ISO) verläuft eine Verbindungsanforderung nach folgendem Schema:

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

Der Parameter *server* übergibt den Netz- und Transportselektor des Servers, mit dem der Client kommunizieren möchte. Vor dem Aufruf von *connect()* muss dem Socket des Clients ein Name zugeordnet sein, d.h. zuvor muss *bind()* für den Socket aufgerufen worden sein.

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 [Abschnitt „Verbindungsannahme durch den Server“ auf Seite 49](#)). Jedoch bleiben auch bei nicht erfolgreichem Verbindungsaufbau alle durch *bind()* zugeordnete Namen erhalten.

2.8.2 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. *listen()* veranlasst SOCKETS(BS2000) außerdem, die an den betreffenden Socket gerichteten Verbindungsanforderungen in eine Warteschlange zu stellen. Auf diese Weise geht normalerweise keine Verbindungsanforderung verloren, während der Server eine andere Verbindungsanforderung bearbeitet.

Ausnahme: Ablauf BCAM-Conn-Timer:

Dieser Timer muss bei Nutzung des ISO-Transport-Service stärker beachtet werden, weil die Verbindungsaufbau-Quittung hier im Gegensatz zu AF_INET und AF_INET6 erst durch ein Senden, z.B. mit *send()*, erzeugt und an den Partner gesendet wird.

2. Mit *accept()* nimmt der Server die Verbindung für den als „abhörend“ markierten Socket an. Mit den Funktionen *getsockopt()* oder *recvmsg()* lassen sich Verbindungsdaten auswerten, die zuvor bei der Verbindungsanforderung übertragen wurden. Im Gegensatz zur Internet-Domäne ist die Verbindung nach *accept()* noch nicht vollständig aufgebaut. Die Verbindung in Richtung Partner wird erst vollständig aufgebaut durch
 - Senden von Benutzerdaten oder
 - Senden von CFRM-Daten (confirm) mit der Funktion *sendmsg()*.

Der folgende Programmausschnitt skizziert die Verbindungsannahme durch den Server in der Domäne AF_ISO:

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

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 die

Server-Task warten können. Allerdings wertet SOCKETS(BS2000) diesen Parameter derzeit nicht aus, sondern nimmt Verbindungsanforderungen solange an, bis die maximale Anzahl verfügbarer Sockets belegt ist.

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. Wenn nach Ausführung von *send()* die Verbindung vollständig aufgebaut ist, können über den neu erzeugten Socket Daten ausgetauscht werden. Über den Socket *s* kann der Server weitere Verbindungen annehmen.

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

Im [Kapitel „Erweiterte Funktionen von SOCKETS\(BS2000\)“ auf Seite 73](#) ist näher beschrieben,

- wie eine Server-Task auf mehr als einem Socket Verbindungen annehmen kann und
- wie eine Server-Task verhindern kann, dass der Aufruf von *accept()* blockiert.

2.8.3 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 eine Benutzer-Task Nachrichten senden oder empfangen, ohne jedes Mal das Adressierungspaar anzugeben.

Es gibt mehrere Funktionen zum Senden und Empfangen von Daten:

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

Diese Socket-Funktionen sind ausführlich beschrieben im [Abschnitt „Beschreibung der Funktionen“ auf Seite 122](#).

2.9 Verbindung abbauen und Socket schließen

Verbindungsabbau und Schließen eines Sockets erfolgen je nach Kommunikationsdomäne (AF_INET/AF_INET6 oder AF_ISO) unterschiedlich.

2.9.1 Verbindungsabbau in den Domänen AF_INET und AF_INET6

In den Domänen AF_INET und AF_INET6 kann eine Verbindung wahlweise mit `soc_close()` oder `shutdown()` abgebaut werden. Das Schließen eines Sockets ist nur mit `soc_close()`, nicht jedoch mit `shutdown()` möglich.

Beim Verbindungsabbau wird unterschieden zwischen „graceful disconnect“ und „abortive disconnect“. Die spezifische Behandlung erfolgt durch das Transportsystem bzw. durch die TCP-Protokollmaschine. Mit den Funktionen `soc_close()` und `shutdown()` wird eine der beiden Möglichkeiten ausgewählt.

Die nachfolgenden Erläuterungen des Verbindungsabbaus mit `soc_close()` und `shutdown()` basieren auf der in [Bild 2](#) skizzierten Situation. Es soll eine Client-/Server-Verbindung abgebaut werden, über die in beiden Richtungen Daten transferiert wurden.

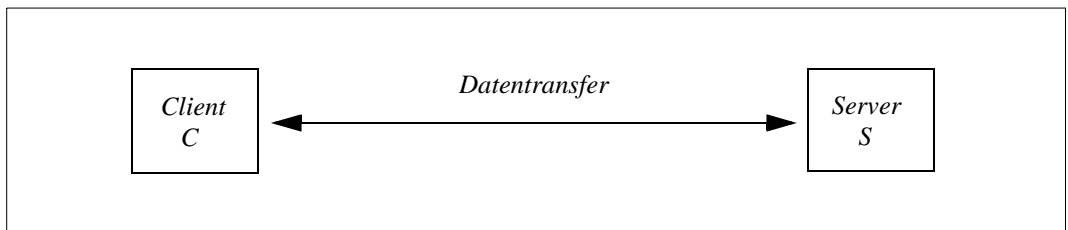


Bild 2: Client-/Server-Verbindung mit bidirektionalem Datentransfer

Verbindungsabbau („graceful“) mit `soc_close()`

Im Einzelnen werden folgende Schritte ausgeführt:

1. Nach dem Senden der letzten Daten leitet der Server S mit `soc_close()` auf den Socket von S den Verbindungsabbau ein. Dadurch wird der betroffene Socket von S für das Schreiben gesperrt, und dem Partner-Socket im Client C wird signalisiert, dass der Socket von S keine Daten mehr senden wird. Dies entspricht einem „graceful disconnect“. Nach einem „graceful disconnect“ wird die Verbindung zwar noch aufrecht erhalten, aber die Datenübertragung von S nach C ist gesperrt.
2. Nach Empfang des Signals für „graceful disconnect“ kann das Anwenderprogramm des Clients C alle noch nicht abgeholten Daten lesen, bis mit EOF das Ende des Datenstroms angezeigt wird.

3. Der Client C ruft *soc_close()* für den Socket von C auf. Dadurch wird an S ein „graceful disconnect“ gesendet und die Verbindung vollständig abgebaut. In C wird der Socket geschlossen. In S wird der Verbindungsabbau signalisiert und der Socket geschlossen.



Falls C auf das „graceful disconnect“-Ereignis mit einem Aufruf von *soc_close()* antwortet, ohne vorher zu versuchen, eventuell noch vorhandene Daten zu lesen, dann wird die Verbindung sofort vollständig abgebaut und Datenverlust ist die Folge.

Verbindungsabbau („graceful“) mit *shutdown()*

Im Einzelnen werden folgende Schritte ausgeführt:

1. Nach dem Senden der letzten Daten leitet der Server S mit *shutdown(..., SHUT_WR)* auf den Socket von S den Verbindungsabbau ein. Dadurch wird der betroffene Socket von S für das Schreiben gesperrt, und dem Partner-Socket im Client C wird signalisiert, dass der Socket von S keine Daten mehr senden wird. Dies entspricht einem „graceful disconnect“. Nach einem „graceful disconnect“ wird die Verbindung zwar noch aufrecht erhalten, aber die Datenübertragung von S nach C ist gesperrt.
2. Nach Empfang des Signals für „graceful disconnect“ kann das Anwenderprogramm des Clients C alle noch nicht abgeholten Daten lesen, bis mit EOF das Ende des Datenstroms angezeigt wird.
3. Der Client C ruft *shutdown(..., SHUT_WR)* für den Socket von C auf. Dadurch wird an S ein „graceful disconnect“ gesendet und die Verbindung vollständig abgebaut.
4. Die Sockets in S und C werden jeweils mit *soc_close()* geschlossen.



Falls C auf das „graceful disconnect“-Ereignis mit einem Aufruf *shutdown(..., SHUT_WR)* antwortet, ohne vorher zu versuchen, eventuell noch vorhandene Daten zu lesen, wird die Verbindung sofort vollständig abgebaut und Datenverlust ist die Folge.

Verbindungsabbau („abortive“) mit *soc_close()*

Im Einzelnen werden folgende Schritte ausgeführt:

1. Der Server S markiert seine Socket-Schnittstelle mit der Option `SO_LINGER` der Funktion *setsockopt()* und setzt das Verzögerungsintervall *l_linger* in der Struktur *linger* auf 0.
2. Wenn der Server die Funktion *soc_close()* aufruft, wird der Verbindungsabbau „abortive“ eingeleitet. Auf den Socket von S ist jetzt weder Schreiben noch Lesen möglich. Dem Partner-Socket in Client C wird der „abortive disconnect“ signalisiert, und der Socket im Server S wird geschlossen.
3. Nach dem Empfang des Signals „abortive disconnect“ kann das Anwenderprogramm im Client C keine Daten mehr lesen. Eventuell vorhandene, aber noch nicht aus dem Transportsystem abgeholte Daten sind verloren.
4. Der Client C kann deshalb nur mit *soc_close()* auf den Socket in S reagieren und damit den Socket in C schließen.

Verbindungsabbau („abortive“) mit *shutdown()*

Im Einzelnen werden folgende Schritte ausgeführt:

1. Der Server S leitet den Verbindungsabbau mit *shutdown(..., SHUT_RDWR)* ein. Auf dem Socket ist jetzt weder Schreiben noch Lesen möglich, und der Partner-Socket im Client C wird über den „abortive disconnect“ informiert.
2. Wenn das Anwendungsprogramm im Client C vor dem Empfang von „abortive disconnect“ eventuell vorhandene Daten vom Transportsystem nicht abgeholt hat, sind diese jetzt verloren.
3. Daher ist es nur sinnvoll, im Client C mit *shutdown(...,SHUT_RDWR)* zu antworten und die Sockets im Server S und Client C jeweils mit *soc_close()* zu schließen.

2.9.2 Verbindungsabbau in der Domäne AF_ISO

In der Domäne AF_ISO steht für den Verbindungsabbau nur die Funktion *soc_close()* zur Verfügung. Bereits mit dem ersten *soc_close()*-Aufruf für den Socket eines Verbindungsendpunktes wird die Verbindung vollständig abgebaut. Noch nicht abgeholte Daten auf der Partnerseite gehen dabei verloren. Verbindungsabbau-Daten, die vorher in den Socket eingetragen wurden (siehe *getsocopt()*, *setsocopt()*, [Seite 161](#)) können jedoch mit der Funktion *soc_close()* übertragen werden.

Verbindungsabbau („abortive“) mit *soc_close()*

Im Einzelnen werden folgende Schritte ausgeführt:

1. Der Server S schreibt bei Bedarf mit der Option TPOPT_DISC_DATA der Funktion *setsockopt()* Verbindungsabbaudaten in den Socket.
2. Wenn der Server die Funktion *soc_close()* aufruft, wird der Verbindungsabbau „abortive“ eingeleitet. Auf den Socket von S ist jetzt weder schreibender noch lesender Zugriff möglich. Dem Partner-Socket in Client C wird der „abortive disconnect“ signalisiert. Sofern vorhanden, werden die Verbindungsabbau-Daten übertragen. Der Socket im Server S wird geschlossen.
3. Nach dem Empfang des Signals „abortive disconnect“ kann das Anwenderprogramm im Client C die Verbindungsabbaudaten lesen, falls diese vom Server übertragen wurden. Weitere Daten kann das Anwenderprogramm nicht mehr lesen. Eventuell vorhandene, aber noch nicht aus dem Transportsystem abgeholte Daten gehen verloren.
4. Der Client C kann deshalb nur mit *soc_close()* auf das Signal von S reagieren und damit den Socket in C schließen.

2.10 Ein-/Ausgabe-Multiplexen

Oft ist es sinnvoll, Ein- und Ausgaben über mehrere Sockets zu verteilen. Für diese Art des Ein-/Ausgabe-Multiplexens können Sie wahlweise die Funktionen `select()` und `soc_poll()` verwenden. Es empfiehlt sich jedoch, die Funktion `select()` bevorzugt einzusetzen.

2.10.1 Ein-/Ausgabe-Multiplexen mit der Funktion `select()`

Mit `select()` kann ein Programm mehrere Verbindungen gleichzeitig überwachen.

Der folgende Programmausschnitt skizziert die Verwendung von `select()`:

```
#include <sys.time.h>
#include <sys.types.h>
...
char *readmask, *writemask, *exceptmask;
struct timeval timeout;
int nfds;
...
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 `exceptmask` wird derzeit von SOCKETS(BS2000) nicht ausgewertet.

Die Bitmasken zu den einzelnen Deskriptormengen werden als Bitfelder in Integer-Reihungen abgespeichert. Die maximal notwendige Größe der Bitfelder ermitteln Sie mit der Funktion `getdtablesize()` (siehe [Seite 142](#)). Der benötigte Speicher sollte dynamisch vom System angefordert werden.

Der Parameter `nfds` spezifiziert, wie viele 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) nicht interessiert sind, übergeben Sie beim `select()`-Aufruf als entsprechenden Parameter den Null-Zeiger.

Mit Makros bearbeiten Sie die Bitmasken. Insbesondere sollten Sie die Bitmasken vor der Bearbeitung auf 0 setzen. Die Makros zur Manipulation von Bitmasken sind erläutert auf [Seite 194](#) 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.

Wenn die Ausführung von *select()* wegen Timeout beendet wird, liefert *select()* den Wert 0 zurück. Die Bitmasken sind dann undefiniert.

Wenn *select()* auf Grund eines Fehlers beendet wird, liefert *select()* den Wert „-1“ zurück mit dem entsprechenden Fehler-Code in *errno*. Die Bitmasken sind dann ebenfalls undefiniert.

Nach der erfolgreichen Ausführung von *select()* überprüfen Sie mit dem Makro-Aufruf `FD_ISSET(fd, &mask)` den Status eines Deskriptors *fd*. 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 folgendem Programmcode (für AF_INET) wird auf eine Verbindungsaufbauanforderung gewartet. Wenn sie eintrifft, wird sie angenommen, und das Programm wird beendet.

```
#include <stdlib.h>
#include <sys.types.h>
#include <sys.socket.h>
#include <sys.time.h>
#include <netinet.in.h>
#include <netdb.h>
#include <stdio.h>
#define TRUE 1
#define TESTPORT 5555
/*
 * Dieses Programm überprüft mit select(), ob jemand versucht, eine
 * Verbindung aufzubauen, und ruft dann accept() auf.
 */
main()
{
    int sock;
    struct sockaddr_in server;
    struct sockaddr_in client;
    int clientlen;
    int msgsock;
    int fdsize;
    char * ready;
    struct timeval to;
    memset(&server,'\0',sizeof(server));
    memset(&client,'\0',sizeof(client));
    clientlen = sizeof(client);

    /* Speicher für Testen der Socket-Deskriptoren durch soc_select()
    anfordern */

    if ((fdsize = getdtablesize()) < 0) {
        perror("get fd_size");
        exit(1);
    }
    if ((ready = memalloc(fdsize/8)) == NULL) {
        perror("no memory space");
        exit(1);
    }
    /* 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);
}
/* Beginn der Annahme von Verbindungen. */
listen(sock, 5);
do {
    memset(ready, 0, fdsize/8);
    FD_SET(sock, (fd_set *)ready);
    to.tv_sec = 5;
    to.tv_usec=0;
    if (select(sock + 1, (fd_set *)ready, (fd_set *)0, (fd_set *)0, &to) < 0)
    {
        perror("select");
        continue;
    }
    if (FD_ISSET(sock, (fd_set *)ready)) {
        msgsock = accept(sock, (struct sockaddr *)&client, &clientlen);

        if (msgsock >= 0)
        {
            /* Erfolgreiche Annahme des Verbindungsaufbauwunsches */
            /* Folgeverarbeitung der Daten, die ueber diese Verbindung */
            /* transferiert werden */
            printf("Programmende nach erfolgreichem Verbindungsaufbau\n");
            break;
        }
        else
        {
            /* Es ist ein Fehler aufgetreten */
            /* Fehlermeldung und eventuell erneutes Warten auf einen */
            /* Verbindungsaufbauwunsch */
            printf("Programmende: Beim Verbindungsaufbau ist ein Fehler
aufgetreten\n");
            break;
        }
    }
} while (TRUE);
exit(0);
}
```

2.10.2 Ein-/Ausgabe-Multiplexen mit der Funktion `soc_poll()`

Auch mit `soc_poll()` kann ein Programm mehrere Verbindungen gleichzeitig überwachen.

Der folgende Programmausschnitt skizziert die Verwendung von `soc_poll()`:

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

fds[0].events = POLLIN;
fds[1].events = POLLOUT
fds[2].events = POLLIN;
...
soc_poll(fds, nfds, timeout);
```

Die zu prüfenden Socket-Deskriptoren und Ereignisse werden in einem Array von Strukturelementen des Typs `pollfd` übergeben. `fds` ist ein Zeiger auf diesen Array. Mit `nfds` wird die Anzahl der Strukturelemente spezifiziert.

Im vorliegenden Beispiel sind dies die Deskriptoren 0..2 und die Ereignisse POLLIN bzw. POLLOUT. POLLIN steht für die Lesebereitschaft und POLLOUT für die Schreibbereitschaft des Socket.

Mit dem Parameter `timeout` wird festgelegt, wie sich die Funktion `soc_poll()` verhalten soll, wenn kein zu prüfendes Ereignis eingetreten ist:

- Bei `timeout = 0` prüft `soc_poll()` nur einmal alle angegebenen Deskriptoren auf das zu testende Ereignis. Danach kehrt die `sock_poll()` wieder zurück, unabhängig davon, ob die Prüfung erfolgreich war oder nicht.
- Bei `timeout > 0` wird eine Wartezeit in Sekunden angegeben. Während dieser Wartezeit blockiert `soc_poll()`, solange keines der zu testenden Ereignisse eintritt.
- Bei `timeout = -1` blockiert `soc_poll()` bis eines der zu testenden Ereignisse eintritt.

Der Returnwert von `soc_poll()` gibt die Menge der Treffer an, d.h. mindestens ein Bit ist im Rückgabefeld `revents` des jeweiligen Struktur-Elements `pollfd` gesetzt.

Struktur `pollfd`, wie sie in `<sys.poll.h>` deklariert ist:

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

Beispiel: Verwenden von `sock_poll()` zum Überprüfen auf anstehende Verbindungsanforderungen

Der folgende Programmcode entspricht dem vorhergehenden Beispiel, es wurde nur die Funktion `select()` durch `sock_poll()` ersetzt.

```
#include <sys.types.h>
#include <stdlib.h>
#include <sys.socket.h>
#include <sys.poll.h>
#include <netinet.in.h>
#include <netdb.h>
#include <stdio.h>
#define TRUE 1
#define TESTPORT 5555
/*
 * Dieses Programm überprüft mit sock_poll(), ob jemand versucht, eine
 * Verbindung aufzubauen, und ruft dann accept() auf.
 */
main()
{
    int sock;
    struct sockaddr_in server;
    struct sockaddr_in client;
    int clientlen;
    int msgsock;
    struct pollfd fds[1];
    unsigned long nfds = 1;
    int timeout = 5;
    memset(&server, '\0', sizeof(server)); memset(&client, '\0', sizeof(client));

    clientlen = sizeof(client);
    /* Initialisieren des Strukturarrays fds für die Abfrage der
    Lesebereitschaft
    des listen sockets */
    fds[0].fd = 0;
    fds[0].events = POLLIN;
    fds[0].revents = 0;
    /* 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);
}
/* Beginn der Annahme von Verbindungen. */
listen(sock, 5);
do {
    fds[0].fd = sock;
    if ((soc_poll(fds, nfds, timeout)) <= 0){
        perror("soc_poll");
        continue;
    }
    else
    {
        if (fds[0].revents & POLLIN) {
            fds[0].revents = 0;
            msgsock = accept(sock, (struct sockaddr *)&client, &clientlen);
            if (msgsock >= 0)
            {
                /* Erfolgreiche Annahme des Verbindungsaufbauwunsches */
                /* Folgeverarbeitung der Daten, die ueber diese Verbindung */
                /* transferiert werden */
                printf("Programmende nach erfolgreichem Verbindungsaufbau\n");
                break;
            }
            else
            {
                /* Es ist ein Fehler aufgetreten */
                /* Fehlermeldung und eventuell erneutes Warten auf einen */
                /* Verbindungsaufbauwunsch */
                printf("Programmende: Beim Verbindungsaufbau ist ein Fehler
aufgetreten\n");
                break;
            }
        }
    }
} while (TRUE);
exit(0);
}
```

2.11 Zusammenspiel der Funktionen der SOCKETS-Schnittstelle

Die folgenden Abbildungen veranschaulichen das Zusammenspiel der Funktionen der SOCKETS(BS2000)-Schnittstelle. Ausführlich beschrieben sind die einzelnen Funktionen im [Kapitel „Benutzerfunktionen von SOCKETS\(BS2000\)“ auf Seite 113](#).

2.11.1 Zusammenspiel der Funktionen bei verbindungsorientierter Kommunikation

Je nach verwendeter Kommunikationsdomäne (AF_INET bzw. AF_INET6 oder AF_ISO) gibt es Unterschiede bei der Abwicklung der verbindungsorientierten Kommunikation.

Verbindungsorientierte Kommunikation in AF_INET und AF_INET6

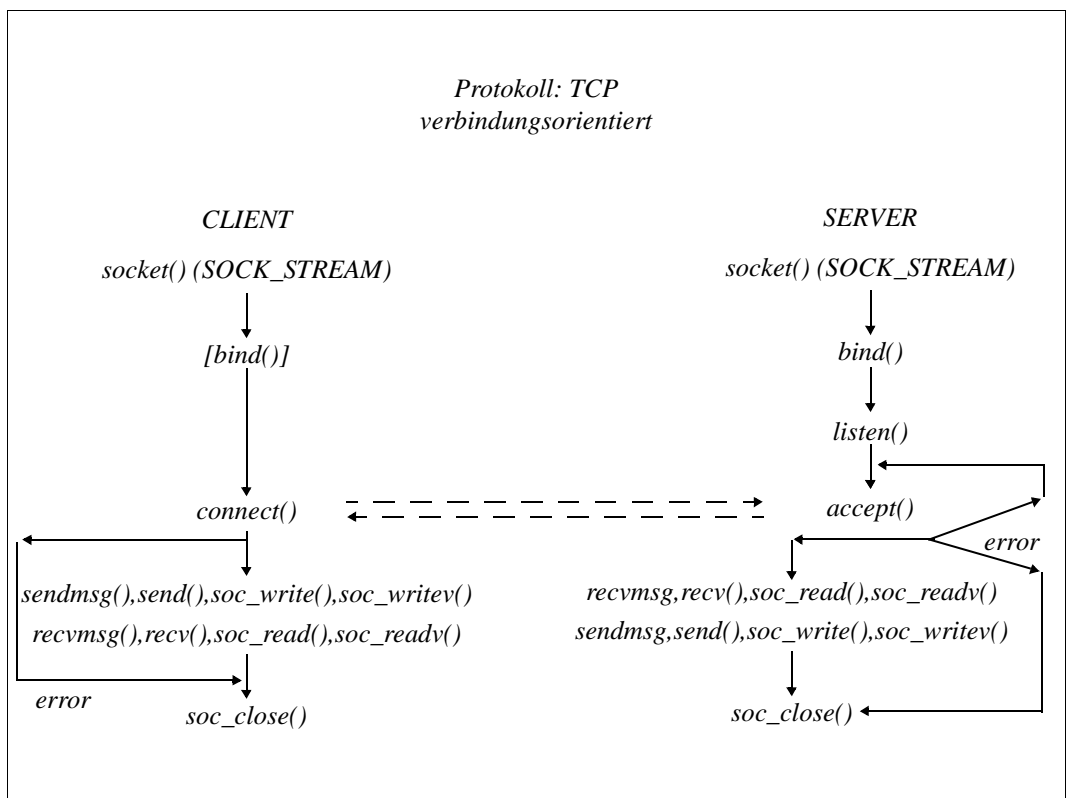


Bild 3: Zusammenspiel der Funktionen der SOCKETS(BS2000)-Schnittstelle bei Stream-Sockets (AF_INET, AF_INET6).

Verbindungsorientierte Kommunikation in AF_ISO

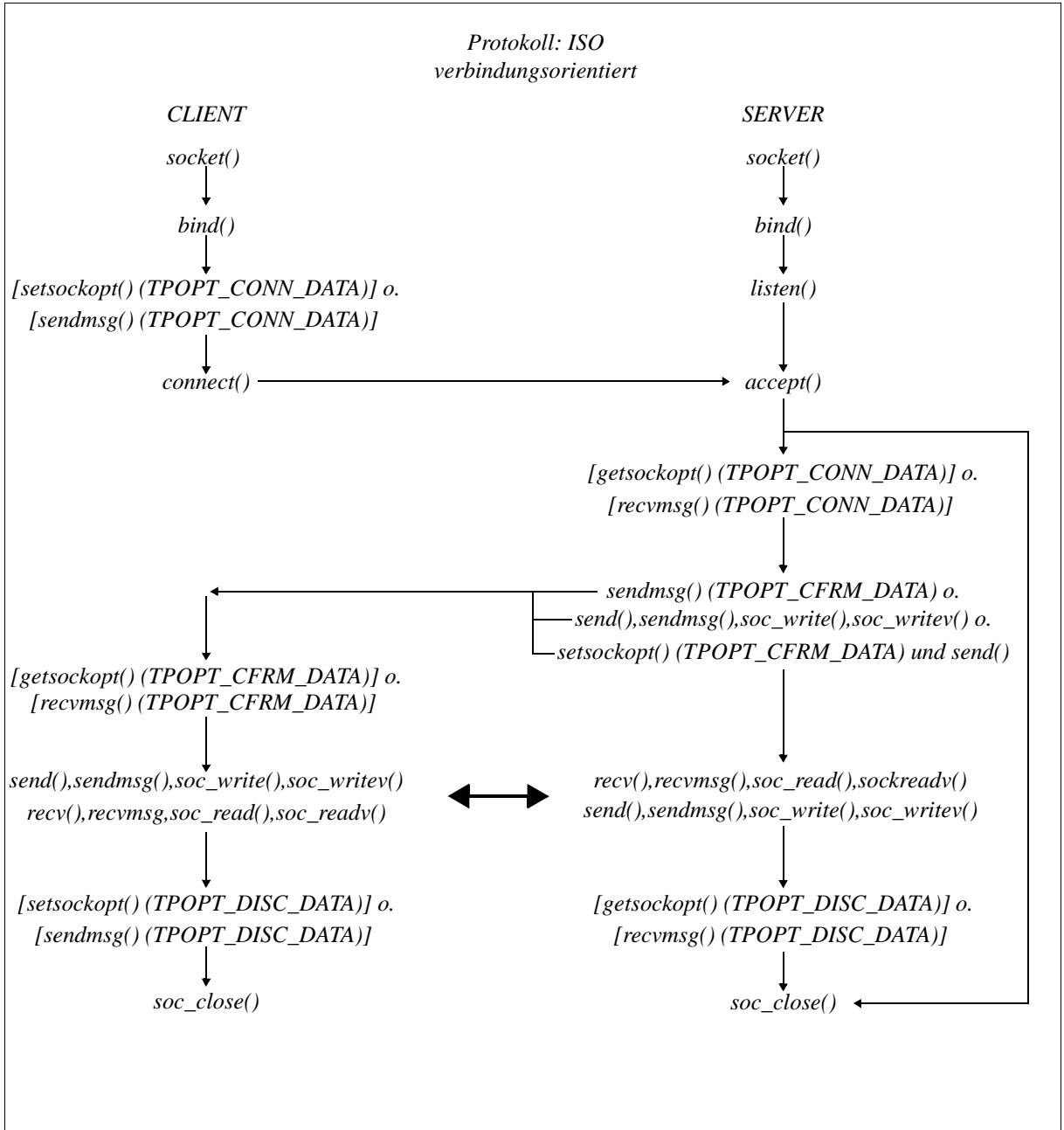


Bild 4: Zusammenspiel der Funktionen der SOCKETS(BS2000)-Schnittstelle (AF_ISO)

2.11.2 Zusammenspiel der Funktionen bei verbindungsloser Kommunikation

Bild 5 veranschaulicht das Zusammenspiel der Funktionen der SOCKETS(BS2000)-Schnittstelle bei Datagramm-Sockets (SOCK_DGRAM).

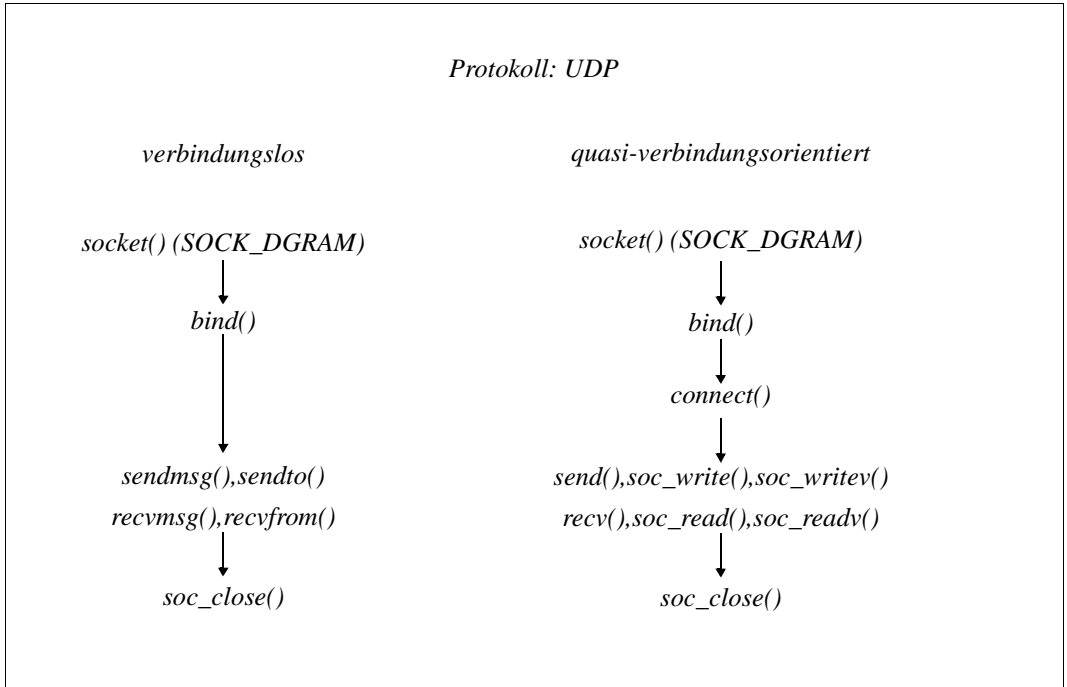


Bild 5: Zusammenspiel der Funktionen der SOCKETS(BS2000)-Schnittstelle bei Datagramm-Sockets

3 Adressumwandlung bei SOCKETS(BS2000)

Damit Prozesse über Sockets miteinander kommunizieren können, müssen Netzadressen ermittelt und erzeugt werden. Zu diesem Zweck stellt die SOCKETS(BS2000)-Bibliothek für die Kommunikationsdomänen AF_INET und AF_INET6 verschiedene Hilfsfunktionen und Makros bereit, die in diesem Kapitel vorgestellt werden.

Ausführlich beschrieben sind alle Hilfsfunktionen im [Kapitel „Benutzerfunktionen von SOCKETS\(BS2000\)“ auf Seite 113](#).

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

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 Umwandlungsfunktionen zeigen, finden Sie im [Kapitel „Client-/Server-Modell bei SOCKETS\(BS2000\)“ auf Seite 87](#).

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 Rechnernamen übergeben.

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

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 Rechnernamen übergeben.

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

gethostbyname() und *gethostbyaddr()*, sowie auch *getipnodebyname()* und *getipnodebyaddr()* 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()*, sowie von *getipnodebyname()* und *getipnodebyaddr()* zurückgelieferte *hostent*-Objekt enthält – wenn von der Datenbasis zur Verfügung gestellt – 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 in `<netdb.h>` 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 in `<netdb.h>` 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 */
};
```



Bis openNet Server V3.4 mit SOCKETS(BS2000) V2.5 fand die Umsetzung auf der Basis einer in SOCKETS(BS2000) enthaltenen statischen Liste statt.

Ab openNet Server V3.5 mit SOCKETS(BS2000) V2.6 wird eine Services-Datei mit dem Standardnamen `SYSDAT.BCAM.ETC.SERVICES` angeboten, die von BCAM (siehe Handbuch „[BCAM Band 1/2](#)“) verwaltet wird. Diese Datei wird mit der Standardbelegung der Ports 1-1023 ausgeliefert. Wenn Sie über entsprechende Nutzerrechte verfügen, können Sie diese Datei ändern. Sie können dann Standard-Port-Belegungen verändern und Port-Belegungen ergänzen.

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, müssen Sie in einem Internet-Anwenderprogramm Adressen selten direkt behandeln. Sie entwickeln dann Services weitgehend netzunabhängig. Ein Rest von Netzabhängigkeit bleibt jedoch bestehen, da in einem Anwenderprogramm die IP-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 der folgenden Tabelle zusammengefassten Makros setzen Bytes und Integers von Rechner-Byte-Reihenfolge in Netz-Byte-Reihenfolge um und umgekehrt.

Bibliotheks-Makros für die Umwandlung von Byte-Reihenfolgen:

| Aufruf | Bedeutung |
|-------------------------|---|
| <code>htonl(val)</code> | 32-bit-Felder von Rechner- in Netz-Byte-Reihenfolge umwandeln |
| <code>htons(val)</code> | 16-bit-Felder von Rechner- in Netz-Byte-Reihenfolge umwandeln |
| <code>ntohl(val)</code> | 32-bit-Felder von Netz- in Rechner-Byte-Reihenfolge umwandeln |
| <code>ntohs(val)</code> | 16-bit-Felder von Netz- in Rechner-Byte-Reihenfolge umwandeln |

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. So können diese Netzadressen einfach in die Strukturen, die dem System zur Verfügung stehen, kopiert werden. Sie sollten deshalb nur beim Interpretieren von Netzadressen auf Probleme mit Byte-Reihenfolgen treffen.

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

In der IPv6-Implementierung wird für die Netzadresse grundsätzlich eine Netz-Byte-Reihenfolge erwartet, d.h. es gibt per Definition keinen Unterschied zwischen Rechner-Byte-Reihenfolge und Netz-Byte-Reihenfolge. Deshalb gibt es auch keine entsprechende Umwandlungsfunktionen.

Gegebenenfalls muss nur die Portnummer umgewandelt werden.

3.5 Beispiel zur Adressumwandlung

Der nachfolgend dargestellte Client-Programmcode des *remote login* demonstriert die in den vorhergehenden Abschnitten erläuterte Adressumwandlung.

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

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

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

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

4 Erweiterte Funktionen von SOCKETS(BS2000)

In den meisten Fällen reichen die in den vorhergehenden Kapiteln beschriebenen Techniken für die Entwicklung verteilter Anwendungen aus. Gelegentlich ist es jedoch erforderlich, zusätzlich auf folgende Leistungsmerkmale von SOCKETS(BS2000) zurückzugreifen:

- nicht-blockierende Sockets
- Multicast-Nachrichten
- Socket-Optionen
- Unterstützung von virtuellen Hosts
- Handoff (Verschieben eines Accept-Socket)
- Raw-Sockets

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 markieren Sie mit der Funktion *soc_ioctl()* (siehe [Seite 212](#)) wie folgt als nicht-blockierend:

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

Wenn auf nicht-blockierenden Sockets die Funktionen *accept()*, *connect()* oder Ein-/Ausgabe-Funktionen ausgeführt werden, achten Sie sorgfältig auf den Fehler `EWOULDBLOCK`. `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.



Die Eigenschaft „nicht-blockierend“ eines Listen-Sockets wird nicht auf den durch *accept()* angelegten Socket vererbt.

4.2 Multicast-Nachrichten (AF_INET, AF_INET6)

Mit Multicast-Nachrichten kann ein Sender im Gegensatz zu Unicast-Nachrichten mehrere Empfänger erreichen. Im Unterschied zu Broadcast-Nachrichten findet jedoch eine Selektion statt, denn jeder Empfänger muss sich einer Multicast-Gruppe anschließen, um entsprechende Nachrichten zu bekommen. Ein Sender meldet sich nicht an, es ist aber möglich, gleichzeitig lokal zu empfangen.

Durch Multicast-Nachrichten werden System-Ressourcen geschont und Bandbreite im Netz eingespart, insbesondere dann, wenn es sich um eine Nutzung handelt, bei der es nur eine Senderichtung gibt. Praktische Anwendungsszenarien für Multicast-Nachrichten sind Filestreams, z.B. für Musik oder Video, Video-Konferenzen oder Nachrichten- oder Börsenticker.

Multicast-Nachrichten werden mit Datagramm-Paketen übertragen, also mit einem ungesicherten Dienst. Die Anwendung muss daher gewährleisten, dass die Daten unter Einhaltung der Integrität den Empfänger erreichen. Und sie muss dafür sorgen, dass die Daten nur an autorisierte Empfänger ausgeliefert werden.

Voraussetzungen

Für die Multicast-Nachrichtenübertragung werden sowohl im IPv4-Adressraum als auch im IPv6-Adressraum eigene Bereiche genutzt. Die genutzten Kommunikationssysteme wie z.B. BCAM im BS2000 müssen den Multicast-Betrieb erlauben und unterstützen.

In BCAM ist der Multicast-Betrieb mit den Standardeinstellungen zugelassen. Im Zweifelsfall sollte die Konfiguration überprüft und bei Bedarf administrativ eingegriffen werden. Näheres siehe Handbuch „[BCAM Band 1/2](#)“.

Wenn die Nachrichten den lokalen Bereich verlassen sollen, sind auch multicastfähige Router erforderlich, die entsprechend konfiguriert sein müssen.

SOCKETS-Funktionen zur Multicast-Unterstützung

SOCKETS(BS2000) bietet Funktionen, um Multicast-Nachrichten zu senden und zu empfangen und sich an Multicast-Gruppen an- oder abzumelden.

Für IPv4 ist der Adressbereich 224.0.0.0 bis 239.255.255.255 vorgesehen, wobei die Adressen 224.0.0.0 bis 224.0.0.255 für lokale Anwendungen reserviert sind und nicht geroutet werden.

Der Multicast-Adressbereich in IPv6 beginnt mit dem Präfix FF, gefolgt von 4 Bit flags und 4 Bit Scope. Die genaue Zuordnung ist im RFC „IP Version 6 Addressing Architecture“, derzeit aktuell RFC 4291, beschrieben.

Reservierte Multicast-Adressen in IPv4 und IPv6 können bei der Internet Assigned Numbers Authority (IANA) eingesehen werden.

Socket-Optionen für AF_INET

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

- IP_ADD_MEMBERSHIP: Anmeldung an eine Multicast-Gruppe
Nach der Anmeldung werden Daten dieser Gruppe zugestellt.
- IP_DROP_MEMBERSHIP: Abmeldung von einer Multicast-Gruppe
- IP_MULTICAST_IF: Sender-Interface zeigen oder festlegen
- IP_MULTICAST_TTL: Multicast-Hop-Limit zeigen oder festlegen
- IP_MULTICAST_LOOP: Es ist ein Empfang auf dem lokalen sendenden Host möglich

Socket-Optionen für AF_INET6

In der Adressfamilie AF_INET6 wird die Übertragung von Multicast-Nachrichten durch folgende Socket-Optionen unterstützt:

- IPV6_JOIN_GROUP: Anmeldung an eine Multicast-Gruppe
Nach der Anmeldung werden Daten dieser Gruppe zugestellt.
- IPV6_LEAVE_GROUP: Abmeldung von einer Multicast-Gruppe
- IPV6_MULTICAST_IF: Index des Sender-Interfaces zeigen oder festlegen
- IPV6_MULTICAST_HOPS: Multicast-Hop-Limit zeigen oder festlegen
- IPV6_MULTICAST_LOOP: Es ist ein Empfang auf dem lokalen, sendenden Host möglich

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., um für eine Socket-Verbindung die *keepalive*-Überwachung zu aktivieren bzw. das Zeitintervall für die Überwachung zu verändern.

Die allgemeine Form der Aufrufe lautet:

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

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

s bezeichnet den Socket, für den die Option gesetzt bzw. abgefragt werden soll.

level gibt die Protokollebene an, der die Option angehört. Normalerweise ist dies die Socket-Ebene, die durch die symbolische Konstante SOL_SOCKET (bei AF_INET, AF_INET6) bzw. SOL_TRANSPORT (bei AF_ISO) angezeigt wird. SOL_SOCKET und SOL_TRANSPORT sind definiert in <sys.socket.h>.

Weitere *level* sind SOL_GLOBAL, IP_PROTO_TCP, IP_PROTO_IPv4, IP_PROTO_IPv6, IPPROTO_ICMP und IPPROTO_ICMPv6. Aus Kompatibilitätsgründen wird auch IP_PROTO_IP wie IP_PROTO_IPv4 unterstützt. Eine Beschreibung dieser *level* finden Sie bei der Beschreibung der Funktionen *getsockopt()* und *setsockopt()* ab [Seite 161](#).

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.

Bei *setsockopt()* gibt *optlen* die Länge von *optval* an. Bei *getsockopt()* ist *optlen* ein Zeiger auf einen Integer-Wert, der beim Aufruf die Größe des Speicherplatzes angibt, auf den *optval* zeigt. Nach der Rückkehr von *getsockopt()* wird als Integer-Wert, auf den *optlen* zeigt, die aktuelle Länge des in *optval* zurückgelieferten Wertes zurückgeliefert.

4.4 Unterstützung von virtuellen Hosts

Zusätzlich zu einem realen Host (Standardhost) können mehrere virtuelle Hosts definiert werden. Der reale und der virtuelle Host werden durch die statische oder dynamische Generierung, die BCAM anbietet, eingerichtet. Zusätzliche Maßnahmen sind erforderlich, um die Adressierbarkeit der Anwendungen zu gewährleisten. Es ist möglich, dass ein virtueller Host über mehrere IP-Adressen verfügt.

Diese Funktionalität hat keine Auswirkungen auf bestehende oder neue Standardanwendungen. Sie wird mit Hilfe neuer Subfunktionen von *soc_ioctl()* und *getsockopt()*, *setsockopt()* erbracht, die den Sockets-Anwender in die Lage versetzen, die notwendigen Informationen über die Konfiguration mit virtuellen Hosts zu erhalten und entsprechend in den Anwendungen einzusetzen.

Beim Ablauf der *bind()*-Funktion wird entschieden, auf welchem Host die Anwendung abläuft. Zu diesem Zeitpunkt muss dem Socket mitgeteilt werden, welcher Host adressiert werden soll.

Bei einer Single-Adressierung erfolgt dies automatisch durch die angegebene IP-Adresse, bei einer ANYADDR- oder LOOPBACK-Adressierung ist die Angabe des entsprechenden BCAM-Hostnamen erforderlich. Er muss im Bedarfsfall mit der neuen *setsockopt()*-Subfunktion *SO_VHOSTANY* vor der Ausführung von *bind()* im Socket eingetragen werden. Das ist notwendig, weil sowohl die ANYADDR, als auch die LOOPBACKADDR keinem Host eindeutig zugeordnet werden kann.

Mit den neuen *soc_ioctl()*-Subfunktionen *SIOCGLVHNUM* und *SIOCGLVHCONF* können die Anzahl von virtuellen Hosts und die dazugehörigen BCAM-Hostnamen und Socket-Hostnamen ermittelt werden.

Es ist zu beachten, dass natürlich weiterhin eine Zuordnung von Sockets-Anwendungen zu einem virtuellen Host über die Application-Tabelle in BCAM erfolgen kann.

Darum ist es auch möglich, den realen Host mit der neuen Funktionalität zu adressieren.

BCAM-Hostname:

Er ist acht Zeichen lang. Es dürfen alphanumerische Zeichen und die Sonderzeichen #, @, \$ oder Leerzeichen am Namensende verwendet werden. In der Regel sollten Großbuchstaben verwendet werden, es wird aber Groß-/Kleinschreibung unterschieden. Ein nur numerischer Anteil ist nicht erlaubt.



Im Standardfall ist im Transportsystem BCAM die Funktionalität *HOST-ALIASING* aktiv. Dies kann bei der Nutzung der Funktionalität zur Unterstützung von virtuellen Hosts zu ungewünschten Effekten führen.

HOST-ALIASING bedeutet, dass ein Verbindungsaufbauwunsch zu einem virtuellen Host an den realen Host weitergeleitet wird, wenn die entsprechende Portnummer nur im realen Host eröffnet ist.

Mit *setsockopt(fd, SOL_SOCKET, SO_DISHALIAS, 1, 4)* ist es möglich, das HOST-ALIASING am Listen-Socket des realen Hosts zu unterbinden. Wird dieses Flag vor dem *bind()* im Socket gesetzt, dann schaltet der nachfolgende *bind()* im Transportsystem BCAM für diese Portnummer das HOST-ALIASING aus. Das hat zur Folge, dass ein Verbindungsaufbauwunsch zu dieser Portnummer auf einem virtuellen Host nur erfolgreich sein kann, wenn der Port mit der entsprechenden Adresse auf dem virtuellen Host auch eröffnet ist. Eine Weiterleitung an den realen Host findet für diese Anwendung dann nicht mehr statt.

4.5 Handoff (Verschieben eines Accept-Socket)

4.5.1 Allgemeine Beschreibung

Mit der Handoff-Funktionalität ist es möglich, einen Sockets-Verbindungsendpunkt zu verschieben, ohne dass der Verbindungsaufbau unterbrochen werden muss. D.h. die aktiv aufbauende Anwendung muss den Vorgang nicht wiederholen. Dies wird durch eine erweiterte Funktionalität unter Einbeziehung der ISO-Service-Funktionalität in der Domäne AF_ISO möglich. Dazu ist es notwendig, für die interne Kommunikation eine lokale AF_ISO-Verbindung aufzubauen.

Ein praktisches Beispiel ist die Annahme von Verbindungswünschen durch einen zentralen Listener und dann die Verschiebung des Endpunktes auf einen zugewiesenen Server.

4.5.2 Funktionsablauf

Für diese Funktionalität werden neue Subfunktionen für *sendmsg()*, *recvmsg()*, *setsockopt()* und *getsockopt()* zur Verfügung gestellt.

Dazu sind folgende Strukturen in der Include-Datei *sys.socket.h* notwendig:

```

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

struct red_info_tcp {
    short    fd;                /* file descriptor (listener)    */
    short    port;             /* port number                   */
    short    domain;          /* address family                */
    short    flags;           /* flags of success              */
    int      cid;              /* cid                            */
    int      if_index;        /* interface index listener process */
    int      rwindow;         /* max read window               */
    int      wwindow;         /* max write window              */
};

struct red_info_iso {
    short    fd;                /* file descriptor (listener)    */
    short    domain;          /* address family                */
    short    flags;           /* flags of success              */
    short    tsellen;         /* length of TSEL                */
    int      rwindow;         /* max read window               */
    int      wwindow;         /* max write window              */
    char     tsel[32];        /* TSEL application              */
};

```



```

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

```

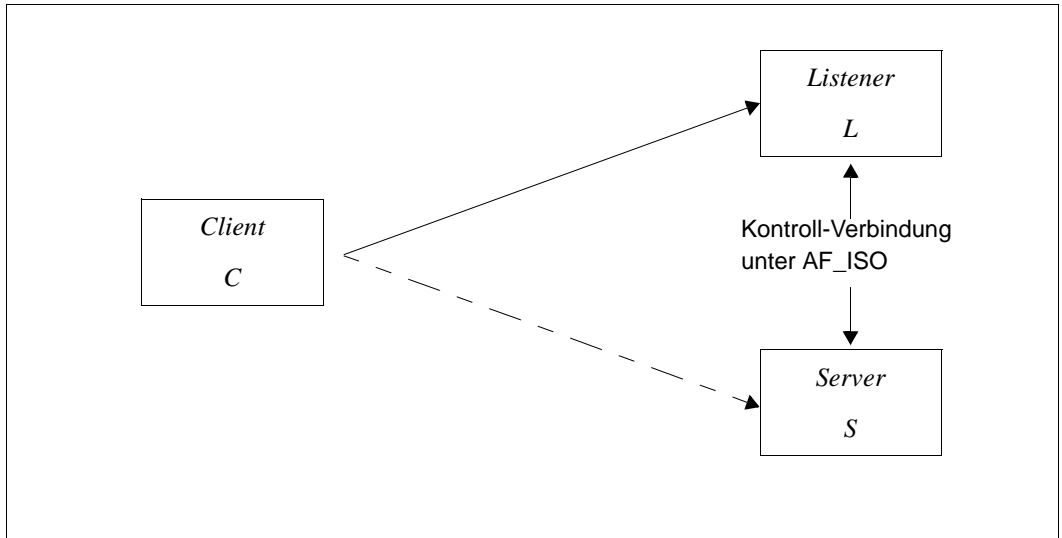
Ablaufreihenfolge

Annahme: Client = C, Listener = L, Server = S

Der Listen-Socket des Listeners L hat den File-Deskriptor $fd = 1$.

Zwischen Listener L und Server S wird eine Verbindung unter der Domäne AF_ISO aufgebaut, mit dem $fd 0$ auf der L-Seite und dem $fd 0$ auf der S-Seite. Diese Verbindung kann sowohl im Blocking- als auch im Nonblocking-Modus aufgebaut werden.

Im Nonblocking-Modus ist es zwingend erforderlich, dass die ausstehenden Ereignisse mit *select()* oder *soc_poll()* abgefragt werden.



Der AF_ISO Listen-Socket auf der S-Seite hat den *fd* 0. Das Verschieben läuft in folgenden Schritten ab:

- C baut eine Verbindung zu L auf.
- Die Verbindung wird von L angenommen und nach S verschoben.
- In L wird mit *sendmsg()* über die lokale AF_ISO-Verbindung die Information über die Domäne der von C aufgebauten Verbindung und der *fd* des Accept-Sockets dieser Verbindung an S weitergegeben.

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

msg.msg_control ist ein Zeiger auf eine Struktur vom Typ *cmsg_redhdr*.

In *msg.msg_control_len* wird die Länge der *struct cmsg_redhdr* eingetragen.

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

In *cmsg_redhdr.domain* muss die Adressfamilie des zu verschiebenden Endpunkts und in Abhängigkeit von dieser Adressfamilie der *fd* des Accept-Sockets in *cmsg_redhdr.redhdr_red_liso* (bei AF_ISO) oder *cmsg_redhdr.redhdr_red_ltcp* (bei AF_INET oder AF_INET6) im Element *fd* eingetragen werden.

- Auf der S-Seite wird mit *recvmsg()* auf der AF_ISO Verbindung gelesen.

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

msg.msg_control ist ein Zeiger auf eine Struktur vom Typ *cmsg_redhdr*

```
cmsg_redhdr.cmsg_len = sizeof(cmsg_redhdr)
cmsg_redhdr.cmsg_level = SOL_TRANSPORT
cmsg_redhdr.cmsg_type = TPOPT_REDDI_DATA
```

Aktion auf S:

Mit der Information von L, die in der übergebenen Struktur vom Typ *cmsg_redhdr* enthalten ist, wird durch einen internen *bind()* ein neuer Verbindungsendpunkt erzeugt. Der *fd* dieses Endpunktes wird im Feld *cmsg_redhdr.fd_server* zusammen mit der Adressfamilie in *cmsg_redhdr.domain* als Returnwert übergeben.

- e) Von der S-Seite wird an die L-Seite die Information gesandt, dass der neue Verbindungsendpunkt erfolgreich erstellt wurde.

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

msg.msg_control ist ein Zeiger auf eine Struktur vom Typ *cmsg_redhdr*

```
cmsg_redhdr.cmsg_len = sizeof(cmsg_redhdr)
cmsg_redhdr.cmsg_level = SOL_TRANSPORT
cmsg_redhdr.cmsg_type = TPOPT_REDDI_BDOK
```

Zusätzlich muss in *cmsg_redhdr.domain* die Adressfamilie des Sockets für den neuen Endpunkt und in *cmsg_redhdr.fd_server* der *fd* dieses Sockets eingetragen werden.

- f) Auf der L-Seite muss auf die Information, dass der neue Endpunkt vorhanden ist, gewartet werden, bevor die eigentliche Verschiebung ausgeführt werden kann.

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

msg.msg_control ist ein Zeiger auf eine Struktur vom Typ *cmsg_redhdr*

```
cmsg_redhdr.cmsg_len = sizeof(struct cmsg_redhdr)
cmsg_redhdr.cmsg_level = SOL_TRANSPORT
cmsg_redhdr.cmsg_type = TPOPT_REDDI_BDOK
```

Zusätzlich muss in *cmsg_redhdr.domain* die Adressfamilie des zu verschiebenden Endpunktes und in Abhängigkeit von dieser Adressfamilie der *fd* des Accept-Sockets in *cmsg_redhdr.redhdr_red_liso* (bei AF_ISO) oder *cmsg_redhdr.redhdr_red_ltcp* (bei AF_INET oder AF_INET6) im Element *fd* eingetragen werden.

Im Feld *cmsg_redhdr.bind_ok* kann überprüft werden, ob mit dem Wert REDBIND_OK das erfolgreiche Einrichten des neuen Endpunktes auf der S-Seite quittiert wird.

Intern wird jetzt ein Datenstopp für diesen Verbindungsendpunkt ausgelöst, d.h. es können vom Client Daten gesendet werden, sie werden jedoch nicht mehr an den alten Verbindungsendpunkt zugestellt.

- g) Durch *setsockopt()* auf der L-Seite wird die Funktionalität des Endpunktes verschoben, d.h. die im Accept-Socket eingetragenen Partner-Informationen werden an den Socket des neuen Endpunkts im Server übertragen.

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

optval ist ein Zeiger auf eine Struktur vom Typ *cmsg_redhdr*

```
level = SOL_TRANSPORT
```

```
optname = TPOPT_REDI_CALL
```

```
cmsg_redhdr.cmsg_len = sizeof(struct cmsg_redhdr)
```

```
cmsg_redhdr.cmsg_level = SOL_TRANSPORT
```

```
cmsg_redhdr.cmsg_type = TPOPT_REDI_CALL
```

- h) Mit *getsockopt()* auf der S-Seite wird auf die Daten des Accept-Sockets der L-Seite gewartet.

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

optval ist ein Zeiger auf eine Struktur vom Typ *cmsg_redhdr*

```
level = SOL_TRANSPORT
```

```
optname = TPOPT_REDI_CALL
```

```
cmsg_redhdr.cmsg_len = sizeof(struct cmsg_redhdr)
```

```
cmsg_redhdr.cmsg_level = SOL_TRANSPORT
```

```
cmsg_redhdr.cmsg_type = TPOPT_REDI_CALL
```

In *cmsg_redhdr.domain* muss die Adressfamilie und in *cmsg_redhdr.fd_server* der *fd* des Sockets für den neuen Verbindungsendpunkt eingetragen werden.

Ist das Ereignis eingetroffen und abgeholt, wird die Verbindungsumgebung abschließend hergestellt und der Datenstopp für die Verbindung aufgehoben, d.h. Daten werden jetzt dem neuen Verbindungsendpunkt zugestellt.

Der Accept-Socket des ursprünglichen Endpunktes kann jetzt mit *soc_close()* geschlossen werden, auch die AF_ISO-Verbindung für die Handoff-Kommunikation kann geschlossen werden.

4.6 Raw-Sockets

Mit einem Raw-Socket kann sowohl ein ICMP-Protokoll-Header, z.B. für einen ICMP-Echo-Request, als auch ein ICMPv6-Protokoll-Header, z.B. für einen ICMPv6-Echo-Request, geschrieben werden.

4.6.1 ICMP

Das Protokoll ICMP (das immer im Zusammenhang mit IPv4 zu sehen ist) bietet die Möglichkeit, zu testen, ob ein Datenpaket ein Endsystem (Host) erreicht, und ob es quittiert wird. Eine detaillierte Beschreibung des ICMP entnehmen Sie dem RFC 792.

Beachten Sie beim Aufbau des Protokolls und der Daten folgende zwei Besonderheiten:

- Die ICMP-Header-Checksum muss von der Anwendung generiert werden.
- Als Identifier wird die Portnummer des Socket erwartet. Führen Sie daher vor dem Senden der Nachricht einen *bind()* auf den Raw-Socket aus.

Bevor Sie die *bind()*-Funktion aufrufen, um einen Port zu reservieren, müssen Sie für diesen Socket die Zustellung von möglichen ICMP-Fehlermeldungen aktivieren (siehe [Seite 161ff](#)):

```
setsockopt(...,IPPROTO_IPV4, IP_RECVERR,...,...)
```

Der ICMP-Header hat eine Länge von 4 Byte. Die Länge der folgenden Daten ist variabel. Beim Type ECHO-REQUEST und ECHO-REPLY gilt: Das erste Wort enthält den Identifier (Portnummer) und die Sequence-Nummer. Die nächsten zwei Worte enthalten einen Zeitstempel. Im ersten Wort ist die Zeit in Sekunden, im zweiten Wort in Mikrosekunden abgelegt:

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

Der dazugehörige IPv4-Header wird vom Transportsystem generiert. Die Anwendung hat jedoch die Möglichkeit, das Hop-Limit zu bestimmen. Dazu müssen Sie den Raw-Socket entsprechend einstellen, bevor Sie das Datenpaket (ICMP-Nachricht) senden.

Um das Hop-Limit gezielt einzustellen, verwenden Sie die Funktion *setsockopt(..., IPPROTO_ICMP, IP_TTL,...,...)* (siehe [Seite 161ff](#)).

Die ICMP-Echo-Request-Nachricht wird mit *sendmsg()* gesendet. Die Antwort des Endsystems wird als ICMP-Echo-Reply-Nachricht mit *recvmsg()* empfangen.

4.6.2 ICMPv6

Das Protokoll ICMPv6 (das immer im Zusammenhang mit IPv6 zu sehen ist) bietet die Möglichkeit zu testen, ob ein Datenpaket ein Endsystem (Host) erreicht, und ob es quittiert wird. Eine detaillierte Beschreibung des ICMPv6 entnehmen Sie dem RFC 4443.

Beachten Sie beim Aufbau des Protokolls und der Daten folgende zwei Besonderheiten (analog zu ICMP):

- Die ICMPv6-Header-Checksum muss von der Anwendung generiert werden.
- Als Identifier wird die Portnummer des Socket erwartet. Führen Sie daher vor dem Senden der Nachricht einen *bind()* auf den Raw-Socket aus.

Bevor Sie die *bind()*-Funktion aufrufen, um einen Port zu reservieren, müssen Sie für diesen Socket die Zustellung von möglichen ICMPv6-Fehlermeldungen aktivieren (siehe [Seite 161ff](#)):

```
setsockopt(...,IPPROTO_IPV6, IPV6_RECVERR,.....)
```

Der ICMPv6-Header hat eine Länge von 4 Byte. Die Länge der folgenden Daten ist variabel. Beim Type ECHO-REQUEST und ECHO-REPLY gilt: Das erste Wort enthält den Identifier (Portnummer) und die Sequence-Nummer. Die nächsten zwei Worte enthalten einen Zeitstempel. Im ersten Wort ist die Zeit in Sekunden, im zweiten Wort in Mikrosekunden abgelegt:

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

Der dazugehörige IPv6-Header wird vom Transportsystem generiert. Die Anwendung hat jedoch die Möglichkeit, das Hop-Limit zu bestimmen. Dazu müssen Sie den Raw-Socket entsprechend einstellen, bevor Sie das Datenpaket (ICMPv6-Nachricht) senden.

Um das Hop-Limit gezielt einzustellen, verwenden Sie die Funktion *setsockopt(..., IPPROTO_ICMPV6, IPV6_HOPLIMIT,.....)* (siehe [Seite 161ff](#)).

Die ICMPv6-Echo-Request-Nachricht wird mit *sendmsg()* gesendet. Die Antwort des Endsystems wird als ICMPv6-Echo-Reply-Nachricht mit *recvmsg()* empfangen.

5 Client-/Server-Modell bei SOCKETS(BS2000)

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 einem Server an. 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.

In den folgenden Abschnitten werden beschrieben:

- verbindungsorientierter Server
- verbindungsorientierter Client
- verbindungsloser Server
- verbindungsloser 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 eine bekannte Internet-Adresse zugegriffen.

Im Folgenden ist je ein Beispiel für einen verbindungsorientierten Server bei AF_INET und AF_INET6 sowie bei AF_ISO dargestellt.

5.1.1 Verbindungsorientierter Server bei AF_INET / AF_INET6

Die Programmierung der Programm-Hauptschleife wird in den folgenden Beispielen gezeigt.

In den Beispielprogrammen benutzt der Server die folgenden Funktionen der Socket-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
- *socket_close()*: Socket schließen

Beispiel: Verbindungsorientierter Server bei AF_INET

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

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

    int sock, length;
    struct sockaddr_in server;
    struct sockaddr_in client;
    int clientlen;
    int msgsock;
    char buf[1024];
```



```
int rval;
memset(&server, '\0', sizeof(server));
memset(&client, '\0', sizeof(client));
clientlen = sizeof(client);

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

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

Beispiel: Verbindungsorientierter Server bei AF_INET6

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

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

    /* Socket erzeugen */

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

    /* Dem Socket einen Namen zuordnen */

    server.sin6_family = AF_INET6;
    memcpy(server.sin6_addr.s6_addr, in6addr_any.s6_addr, 16) ;
    server.sin6_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 *)&client, &clientlen);

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

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

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

In den Programmbeispielen zu AF_INET und AF_INET6 werden folgende Arbeitsschritte durchgeführt:

1. Der Server erzeugt mit der Funktion *socket()* einen Kommunikationsendpunkt (Socket) und den zugehörigen Deskriptor.
2. Mit der Funktion *bind()* wird dem Server-Socket eine definierte Portnummer zugeordnet. Über diese Portnummer kann er im Netz adressiert werden.
3. Mit der Funktion *listen()* stellt der Server fest, ob Verbindungsanforderungen anstehen.
4. Verbindungsanforderungen kann der Server mit *accept()* annehmen. Der Returnwert von *accept()* wird überprüft um sicherzustellen, dass die Verbindung erfolgreich aufgebaut wurde.
5. Sobald die Verbindung aufgebaut ist, werden die Daten mit der Funktion *recv()* vom Socket gelesen.
6. Mit der Funktion *soc_close()* schließt der Server den Socket.

5.1.2 Verbindungsorientierter Server bei AF_ISO

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

- *getbcamhost()*: Rechnereintrag abfragen
- *socket()*: Socket erzeugen
- *bind()*: einem Socket einen Namen zuordnen
- *listen()*: einen Socket auf Verbindungsanforderungen „abhören“
- *accept()*: eine Verbindung über einen Socket annehmen
- *sendmsg()*: eine Nachricht von Socket zu Socket senden / Verbindung bestätigen
- *recv()*: Daten von einem Socket lesen
- *soc_close()*: Socket schließen

Beispiel: Verbindungsorientierter Server bei AF_ISO

```

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

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

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

main(argc, argv)
int argc;
char *argv[];
{
    void error_exit();
    int sockfd, newfd, clilen, ret;
    int tsellen, nsellen;
    char tsel[MAXTSEL];
    char nsel[MAXNSEL];

```

```
char buffer [MAXREC];
struct sockaddr_iso cli_addr, serv_addr;
struct msghdr message;
struct cmsghdr cmessage;

strcpy (tsel,"SERVER");
tsellen = strlen(tsel);
nssel[MAXNSEL-1] = '\0';

/* BCAM Host-Name besorgen */
errno = 0;
if(getbcamhost(nssel,sizeof(nssel)) < 0)
    error_exit("ISO_svr: getbcamhost failed ",errno);
else
    printf ("getbcamhost OK! (%s)\n",nssel);
nssellen = strlen(nssel);

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

/* Dem Socket einen Namen zuordnen */
memset ((char *) &serv_addr, 0, sizeof(serv_addr));
serv_addr.siso_len = sizeof (struct sockaddr_iso);
serv_addr.siso_family = AF_ISO;
serv_addr.siso_plen = 0;
serv_addr.siso_slenn = 0;
serv_addr.siso_tlen = tsellen;
serv_addr.siso_addr.isoa_len = tsellen + nssellen;
memcpy (serv_addr.siso_addr.isoa_genaddr,nssel,nssellen);
memcpy (serv_addr.siso_addr.isoa_genaddr + nssellen,tsel,tsellen);

errno = 0;

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

```

/* Beginn mit der Annahme von Verbindungsanforderungen */
errno = 0;
if (listen(sockfd, INT) < 0)
    error_exit("ISO_svr: Listen failed ",errno);
else
    printf("listen OK!\n");

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

/* Positive Bestaetigung (CONNECTION CONFIRM) des Verbindungsaufbau-
wunsches. Es wird kein echter Datentransfer ausgefuehrt! */
memset ((char *) &message, 0, sizeof(message));
memset ((char *) &cmessage, 0, sizeof(cmessage));
message.msg_control = (char *) &cmessage;
message.msg_controllen = sizeof (struct cmsghdr);
cmessage.cmsg_level = SOL_TRANSPORT;
cmessage.cmsg_type = TPOPT_CFRM_DATA;
cmessage.cmsg_len = sizeof (struct cmsghdr);

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

/* Daten von einem Socket lesen */
if ((ret = recv (newfd, buffer, MAXREC, 0)) < 0)
{
    if (errno != EPIPE) /* Broken Pipe */
        error_exit("ISO_svr: Receive in Error", errno);
}
else
    printf("recv OK!(%d)\n",ret);

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

```

```
    if (soc_close (sockfd) < 0)
        error_exit("ISO_svr: soc_close failed ",errno);
    else
        printf("soc_close (sockfd) OK!\n");
} /* END MAIN */

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

Im Programmbeispiel zu AF_ISO werden folgende Arbeitsschritte durchgeführt:

1. Mit der Funktion *getbcamhost()* ermittelt der Server den BCAM-Host-Namen.
2. Mit der Funktion *socket()* erzeugt der Server einen Kommunikationsendpunkt (Server-Socket) sowie den zugehörigen Deskriptor.
3. Dem neu generierten Socket ordnet der Server mit *bind()* einen Namen zu.
4. Mit *listen()* wird der (Server-)Socket für die Annahme von Verbindungsanforderungen vorbereitet.
5. Mit *accept()* nimmt der (Server-)Socket eine Verbindungsanforderung an.
6. Mit *sendmsg()* bestätigt der Server die Verbindungsanforderung (CFRM), d.h. die Verbindung ist nun aufgebaut. Mit *sendmsg()* wird kein Transfer von Benutzerdaten durchgeführt.
7. Mit *recv()* empfängt der (Server-)Socket Benutzerdaten vom Partner-Socket (Client-Socket).
8. Mit der Funktion *soc_close()* wird der (Server-)Socket wieder geschlossen.

5.2 Verbindungsorientierter Client

Im Folgenden ist je ein Beispiel für einen verbindungsorientierten Client bei AF_INET und AF_INET6 sowie bei AF_ISO dargestellt.

5.2.1 Verbindungsorientierter Client bei AF_INET / AF_INET6

Die Seite des Servers wurde bereits im Beispiel auf [Seite 88](#) dargestellt. Im Programmcode können Sie deutlich die separaten, asymmetrischen Rollen von Client und Server erkennen. Der 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. In den Programmbeispielen verwendet der Client die folgenden Funktionen der Socket-Schnittstelle:

- *socket()*: Socket erzeugen
- *gethostbyname()* / *getipnodebyname()*: Rechneintrag abfragen
- *connect()*: auf einem Socket eine Verbindung anfordern
- *send()*: Daten auf einen Socket schreiben
- *sock_close()*: Socket schließen

Beispiel: Verbindungsorientierter Client bei AF_INET

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

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

    int sock, length;
    struct sockaddr_in client;
    struct hostent *hp;
    char buf[1024];
```



```
/* Socket erzeugen */
sock = socket(AF_INET, SOCK_STREAM, 0);
if (sock < 0)
    { perror("Create stream socket");
      exit(1);
    }
/* Ausfüllen der Adress-Struktur */
client.sin_family = AF_INET;
client.sin_port = htons(TESTPORT);
hp = gethostbyname(argv[1]);
if (hp == NULL)
    { fprintf(stderr,"%s: unknown host\n", argv[1]);
      exit(1);
    }
memcpy((char *) &server.sin_addr, (char *)hp->h_addr,
        hp->h_length);

/* Verbindung starten */
if ( connect(sock, &client, sizeof(client) ) < 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);
    }

soc_close(sock);
}
```

Beispiel: Verbindungsorientierter Client bei AF_INET6

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

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

    /* Socket erzeugen */

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

    /* Ausfüllen der Adress-Struktur */

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

    /* Freigabe des dynamischen Speichers von hostent */
    freehostent (hp);
}
```

```
/* Verbindung starten */

if ( connect(sock, &client, sizeof(client) ) < 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);
}
soc_close(sock);
}
```

In den Programmbeispielen zu AF_INET und AF_INET6 werden folgende Arbeitsschritte durchgeführt:

1. Der Client erzeugt mit der Funktion *socket()* einen Kommunikationsendpunkt (Socket) und den zugehörigen Deskriptor.
2. Mit *gethostbyname()* ermittelt der Client die Adresse des Rechners (gilt nur für AF_INET). Der Rechnername wird als Parameter übergeben.
Mit *getipnodebyname()* ermittelt der Client die IPv6-Adresse des als Parameter übergebenen Rechnernamens. Diese neue Funktion könnte auch für das AF_INET-Beispiel verwendet werden. Danach muss für den gewünschten Rechner eine Verbindung zum Server hergestellt werden. Zu diesem Zweck initialisiert der Client die Adress-Struktur.
3. Die Verbindung wird mit *connect()* aufgebaut.
4. Nach dem Verbindungsaufbau werden mit der Funktion *send()* Daten auf den Socket geschrieben.
5. Mit der Funktion *soc_close()* wird der erzeugte Socket wieder geschlossen.

5.2.2 Verbindungsorientierter Client bei AF_ISO

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

- `getbcamhost()`: BCAM-Hostnamen abfragen
- `socket()`: Socket erzeugen
- `bind()`: dem Socket einen Namen zuordnen
- `connect()`: auf einem Socket eine Verbindung anfordern
- `send()`: Daten auf einen Socket schreiben
- `soc_close()`: Socket schließen

Beispiel: Verbindungsorientierter Client bei AF_ISO

```
/*
 * Beispiel: ISO CLIENT
 *
 * DESCRIPTION
 * 1. getbcamhost - socket - bind - connect
 * 2. send
 * 3. soc_close
 */

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

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

main(argc, argv)
int argc;
char *argv[];
{
    void error_exit();
    int sockfd, ret, lng;
    int tsellen, nsellen, par_tsellen, par_nsel;
    char tsel[MAXTSEL];
    char par_tsel[MAXTSEL];
    char nsel[MAXNSEL];
    char par_nsel[MAXNSEL];
    char buffer [MAXREC];
```

```
struct sockaddr_iso cli_addr, serv_addr;

lng = 1024 ;
strcpy (tsel,"CLIENT");
tsellen = strlen(tsel);
strcpy (par_tsel,"SERVER");
par_tsellen = strlen(par_tsel);
nssel[MAXNSEL-1] = '\0';

/* Partner Host-Name besorgen */
if (argc > 1)
{
    strcpy (par_nssel,argv[1]);
    if ((par_nselllen = strlen(par_nssel)) != MAXNSEL - 1)
    {
        printf ("Fehler: Falscher Rechnername!!\n");
        exit (-1);
    }
}
else
{
    printf ("Partner Host-Name wurde nicht als Argument in der
    Kommandozeile uebergeben!\n");
    exit (-1);
}

/* BCAM Host-Name besorgen */
errno = 0;
if (getbcamhost(nssel,sizeof(nssel)) < 0)
    error_exit("ISO_cli: getbcamhost failed ",errno);
else
    printf ("getbcamhost OK! (%s)\n",nssel);
nsellen = strlen(nssel);

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

/* Dem Socket einen Namen zuordnen */
memset ((char *) &cli_addr, 0, sizeof(cli_addr));
cli_addr.siso_len = sizeof (struct sockaddr_iso);
cli_addr.siso_family = AF_ISO;
cli_addr.siso_plen = 0;
cli_addr.siso_slenn = 0;
cli_addr.siso_tlen = tsellen;
```

```
cli_addr.siso_addr.isoa_len = tsellen + nsellen;
memcpy (cli_addr.siso_addr.isoa_genaddr, nsel, nsellen);
memcpy (cli_addr.siso_addr.isoa_genaddr + nsellen, tsel, tsellen);

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

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

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

sleep(2);

/* Daten auf den Socket schreiben */
ret = send (sockfd, buffer, lng, 0);
if (ret == -1)
    error_exit("ISO_cli: Send in Error", errno);
else
    printf ("send OK!(%d)\n", ret);
```

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

} /* END MAIN */

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

Im Programm werden folgende Arbeitsschritte durchgeführt:

1. Der Client besorgt sich aus dem Kommandozeilen-Argument *argc* der *main()*-Funktion den Namen des Partner-Host.
2. Mit der Funktion *getbcamhost()* ermittelt der Client den BCAM-Host-Namen.
3. Der Client erzeugt mit der Funktion *socket()* einen Kommunikationsendpunkt (Client-Socket) sowie den zugehörigen Deskriptor.
4. Dem neu generierten Socket ordnet der Client mit *bind()* einen Namen zu.
5. Mit *connect()* stellt der Client die Verbindung zum Kommunikationspartner (Server-Socket) her.
6. Mit *send()* sendet der Client Benutzerdaten an den Partner-Socket (Server-Socket).
7. Mit der Funktion *soc_close()* wird der (Client-)Socket wieder geschlossen.

5.3 Verbindungsloser Server

In den meisten Fällen arbeiten Server verbindungsorientiert. Einige Services basieren jedoch auf der Verwendung von Datagramm-Sockets und arbeiten somit verbindungslos.

In den Programmbeispielen verwendet der Server die folgenden Funktionen der Socket-Schnittstelle:

- `socket()`: Socket erzeugen
- `bind()`: einem Socket einen Namen zuordnen
- `recvfrom()`: Nachricht von einem Socket lesen
- `sock_close()`: Socket schließen

Das Programm wird in zwei Varianten vorgestellt:

- In der ersten Variante (Beispiele 1 und 3) wird das Programm beendet, wenn eine Nachricht ankommt (`read()`).
- In der zweiten Variante (Beispiele 2 und 4) wartet das Programm in einer Endlosschleife auf weitere Nachrichten, nachdem eine Nachricht gelesen wurde.

Beispiel 1: Verbindungsloser Server ohne Programmschleife für AF_INET

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

#define TESTPORT 2222

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

soc_close(sock);
}
```

Beispiel 2: Verbindungsloser Server mit Programmschleife für AF_INET

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

/* 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, &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, &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.
 */
}

```

Beispiel 3: Verbindungsloser Server ohne Programmschleife für AF_INET6

```

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

/*
 * 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_in6 server;
    struct in6_addr in6addr_any = IN6ADDR_ANY_INIT;
    char buf[1024];

    /* Erzeugen des Sockets, von dem gelesen werden soll. */
    sock = socket(AF_INET6, SOCK_DGRAM, 0);
    if (sock < 0 )
        { perror("Socket datagram");
          exit(1);
        }

    /* Dem Server "server" unter Verwendung von Wildcards einen Namen
     * zuordnen */
    server.sin6_family = AF_INET6;
    memcpy(server.sin6_addr.s6_addr, in6addr_any.s6_addr, 16) ;
    server.sin6_port = htons(TESTPORT);
    if (bind(sock, &server, sizeof server ) < 0)
        { perror("Bind datagram socket");

```

```

        exit(1);
    }

    /* Beginn des Lesens vom Server */
    length = sizeof(server);
    memset(buf,0,sizeof(buf));
    if ( recvfrom(sock, buf, 1024,0, &server, &length) < 0 )
        { perror("recvfrom");
          exit(1);
        }
    else
        printf("->%s\n",buf);
    soc_close(sock);
}

```

Beispiel 4: Verbindungsloser Server mit Programmschleife für AF_INET6

```

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

#define TESTPORT 2222

/* 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_in6 server;
    struct in6_addr in6addr_any = IN6ADDR_ANY_INIT;
    char buf[1024];

    /* Erzeugen des Sockets, von dem gelesen werden soll. */

    sock = socket(AF_INET6, SOCK_DGRAM, 0);
    if (sock < 0 )
        {
            perror("Socket datagram");
            exit(1);
        }
}

```

```

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

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

/* Beginn des Lesens vom Server */

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

/* 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.
 */
}

```

In den Programmbeispielen zu AF_INET und AF_INET6 werden folgende Arbeitsschritte durchgeführt:

1. Der Server erzeugt mit der Funktion *socket()* einen Kommunikationsendpunkt (Socket) und den zugehörigen Deskriptor.
2. Mit der Funktion *bind()* wird dem Server-Socket eine definierte Portnummer zugeordnet, sodass er über diese Portnummer vom Netz aus adressiert werden kann.
3. Von einem Socket des Typs SOCK_DGRAM kann mit der Funktion *recvfrom()* gelesen werden.
4. Als Ergebnis wird die Länge der gelesenen Nachricht geliefert. Wenn keine Nachricht vorhanden ist, wird der Prozess blockiert, bis eine Nachricht eintrifft.

5.4 Verbindungsloser Client

In diesen Programmbeispielen verwendet der Client die folgenden Funktionen der Socket-Schnittstelle:

- `socket()`: Socket erzeugen
- `gethostbyname()` / `getipnodebyname()`: Rechner-Eintrag abfragen
- `sendto()`: Nachricht an einen Socket senden
- `soc_close()`: Socket schließen

Beispiel: Verbindungsloser Client bei AF_INET

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

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

/*
 * Dieses Programm sendet ein Datagramm an einen Empfänger, dessen Name als
 * Argument in der Kommandozeile übergeben wird.
 */
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, &to, sizeof to) < 0) {
    perror("Sending datagram message");
    exit(1);
}
soc_close(sock);
}

```

Beispiel: Verbindungsloser Client bei AF_INET6

```

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

/*
 * Dieses Programm sendet ein Datagramm an einen Empfänger, dessen Name als
 * Argument in der Kommandozeile übergeben wird. */
main(argc, argv)
    int argc;
    char *argv[];
{
    int sock;
    int error_num;
    struct sockaddr_in6 to;
    struct hostent *hp;

    /* Erzeugen des Sockets, an den gesendet werden soll. */
    sock = socket(AF_INET6, 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 =getipnodebyname(argv[1], AF_INET6, 0, &error_num);
if ((hp == 0) || (error_num != NETDB_SUCCESS))
    {
        fprintf(stderr, "%s:unknown host\n", argv[1]);
        exit(1);
    }
memcpy((char *)&to.sin6_addr, (char *)hp->h_addr, hp->h_length);
to.sin6_family = AF_INET6;
to.sin6_port = htons(TESTPORT);

/* Freigabe des dynamischen Speichers von hostent */
freehostent(hp);

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

```

In den Programmbeispielen zu AF_INET und AF_INET6 werden folgende Arbeitsschritte durchgeführt:

1. Der Client erzeugt mit *socket()* einen Kommunikationsendpunkt (Socket) und den zugehörigen Deskriptor.
2. Mit *gethostbyname()* fragt der Client die Adresse des Rechners ab (bei AF_INET); es wird der Rechnername als Parameter übergeben.
Mit *getipnodebyname()* fragt der Client die IPv6-Adresse des als Parameter übergebenen Rechnernamens ab. Diese neue Funktion könnte auch für das AF_INET Beispiel verwendet werden.
Danach wird die Adress-Struktur initialisiert.
3. Mit *sendto()* sendet der Client ein Datagramm. *sendto()* liefert die Anzahl der übertragenen Zeichen zurück.
4. Mit *soc_close()* schließt der Client den Socket.

6 Benutzerfunktionen von SOCKETS(BS2000)

Dieses Kapitel beschreibt die Funktionen der Socket-Schnittstelle für BS2000. 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 Socket-Schnittstelle in alphabetischer Reihenfolge beschrieben.

6.1 Beschreibungsformat

Die Benutzerfunktionen von SOCKETS(BS2000) sind in einem einheitlichen Format beschrieben. Die Beschreibung der Funktionen ist gemäß dem Schema auf [Seite 114](#) aufgebaut.



Achten Sie im ANSI-Modus auf korrekte Typ-Umwandlung (Cast), um Compiler-Warnings zu vermeiden. Dies betrifft insbesondere die verschiedenen Socket-Strukturen (*sockaddr*, *sockaddr_in*, *sockaddr_in6*, *sockaddr_iso*):

Für den Funktionsaufruf verwenden Sie die allgemeine Struktur *sockaddr*.

Für das Eintragen bzw. Auslesen von Socket-Adressen verwenden Sie die spezifische Struktur der jeweiligen Adressfamilie.

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“.

Fehleranzeige durch *errno*

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. Dieser Abschnitt kann fehlen.

6.2 Übersicht über die Funktionen

Die folgende Übersicht über die Funktionen der SOCKETS(BS2000)-Schnittstelle fasst jeweils mehrere Funktionen unter aufgabenorientierten Gesichtspunkten zusammen.

In den drei rechten Spalten „INET“, „INET6“ und „ISO“ wird angezeigt, in welcher Adressfamilie (AF_INET, AF_INET6, AF_ISO) die betreffende Funktion unterstützt wird.

Verbindungen über Sockets aufbauen und beenden

| Funktion | Beschreibung | siehe | INET | INET6 | ISO |
|-------------|---|---------------------------|------|-------|-----|
| socket() | Socket erzeugen | Seite 232 | x | x | x |
| bind() | einem Socket einen Namen zuordnen | Seite 126 | x | x | x |
| connect() | Kommunikation über einen Socket initiieren (z.B. durch einen Client) | Seite 130 | x | x | x |
| listen() | Socket auf anstehende Verbindungen überprüfen (z.B. durch einen Server) | Seite 185 | x | x | x |
| accept() | Verbindung über einen Socket annehmen (z.B. durch einen Server) | Seite 123 | x | x | x |
| shutdown() | Verbindung in Lese- und/oder Schreibrichtung schließen | Seite 204 | x | x | |
| soc_close() | Socket schließen | Seite 206 | x | x | x |

Daten zwischen zwei Sockets übertragen

| Funktion | Beschreibung | siehe | INET | INET6 | ISO |
|------------------------------|---|---------------------------|-------------|--------------|------------|
| soc_read(), soc_readv() | Nachricht von einem Socket über eine auf- gebaute Verbindung empfangen | Seite 226 | x | x | x |
| recv() | Nachricht von einem Socket über eine auf- gebaute Verbindung empfangen | Seite 187 | x | x | x |
| recvfrom() | Nachricht von einem Socket empfangen | Seite 187 | x | x | x |
| recvmsg() | Nachricht von einem Socket empfangen. AF_ISO: Nachricht (Nutzdaten oder Verbin- dungsdaten) über eine aufgebaute Verbin- dung empfangen. | Seite 190 | x | x | x |
| send() | Nachricht von Socket zu Socket über eine aufgebaute Verbindung senden | Seite 197 | x | x | x |
| sendto() | Nachricht von Socket zu Socket senden | Seite 197 | x | x | x |
| sendmsg() | Nachricht von Socket zu Socket senden. AF_ISO: Nachricht (Nutzdaten oder Verbin- dungsdaten) über eine aufgebaute Verbin- dung senden. | Seite 200 | x | x | x |
| soc_write(), soc_writev() | Nachricht von Socket zu Socket über eine aufgebaute Verbindung senden | Seite 229 | x | x | x |
| select() | Ein-/Ausgabe multiplexen | Seite 194 | x | x | x |
| soc_poll() | Ein-/Ausgabe multiplexen | Seite 221 | x | x | x |

Daten vom/zum Socket-Puffer übertragen

| Funktion | Beschreibung | siehe | INET | INET6 | ISO |
|-----------------|--|---------------------------|-------------|--------------|------------|
| soc_getc(), | Zeichen aus dem Socket-Puffer lesen | Seite 210 | x | x | |
| soc_gets() | Character-String aus dem Socket-Puffer le- sen | Seite 211 | x | x | |
| soc_putc(), | Zeichen in den Socket-Puffer schreiben | Seite 224 | x | x | |
| soc_puts(), | Character-String in den Socket-Puffer schreiben | Seite 225 | x | x | |
| soc_flush() | Socket-Puffer leeren | Seite 209 | x | x | |

Informationen über Sockets erhalten

| Funktion | Beschreibung | siehe | INET | INET6 | ISO |
|-----------------|---|---------------------------|-------------|--------------|------------|
| getdtablesize() | Größe der Deskriptorentabelle abfragen | Seite 142 | x | x | x |
| getsockopt() | Socket-Optionen abfragen | Seite 161 | x | x | x |
| setsockopt() | Socket-Optionen setzen | Seite 161 | x | x | x |
| getpeername() | Namen des Kommunikationspartners abfragen | Seite 153 | x | x | x |
| getsockname() | Namen des Sockets abfragen | Seite 159 | x | x | x |

Konfigurationswerte überprüfen

| Funktion | Beschreibung | siehe | INET | INET6 | ISO |
|-------------------|--|---------------------------|-------------|--------------|------------|
| getaddrinfo() | die zu einem Rechner- und/oder Service-Namen korrespondierenden IP-Adressen und Portnummern abfragen | Seite 136 | x | x | |
| gai_strerror() | Beschreibung eines <i>getaddrinfo()</i> -Error-Codes abfragen | Seite 135 | x | x | |
| getbcamhost() | BCAM-Hostnamen abfragen | Seite 141 | | | x |
| gethostname() | Socket-Hostnamen des aktuellen Rechners abfragen | Seite 145 | x | x | x |
| gethostbyaddr() | den zu einer IPv4-Adresse gehörenden Rechnernamen abfragen | Seite 143 | x | | |
| gethostbyname() | die zu einem Rechnernamen gehörende IPv4-Adresse abfragen | Seite 143 | x | | |
| getipnodebyaddr() | den zu einer IPv4- oder IPv6-Adresse gehörenden Rechnernamen abfragen | Seite 146 | x | x | |
| getipnodebyname() | die zu einem Rechnernamen gehörende IPv4- oder IPv6-Adresse abfragen | Seite 146 | x | x | |
| getnameinfo() | die zu IP-Adresse und Portnummer korrespondierenden Rechner- und Service-Namen abfragen | Seite 150 | x | x | |
| getservbyport() | Namen eines Service abfragen | Seite 157 | x | x | |
| getservbyname() | Portnummer eines Service abfragen | Seite 157 | x | x | |
| getprotobyname() | Nummer eines Protokolls abfragen | Seite 155 | x | x | |
| if_nameindex() | Liste mit Interface-Namen und -Index des lokalen Rechners | Seite 178 | x | x | |

Internet-Adresse manipulieren

| Funktion | Beschreibung | siehe | INET | INET6 | ISO |
|-----------------|---|---------------------------|-------------|--------------|------------|
| inet_addr() | Zeichenkette von Punktschreibweise in ganzzahligen Wert konvertieren (Internetadresse) | Seite 180 | x | | |
| inet_network() | Zeichenkette von Punktschreibweise in ganzzahligen Wert konvertieren (Subnetz-Anteil) | Seite 180 | x | | |
| inet_makeaddr() | Internet-Adresse erstellen aus Subnetz-Anteil und subnetz-lokalem Adressteil | Seite 180 | x | | |
| inet_lnaof() | aus der Internet-Rechneradresse die lokale Netzadresse in der Byte-Reihenfolge des Rechners extrahieren | Seite 180 | x | | |
| inet_netof() | aus der Internet-Rechneradresse die Netznummer in der Byte-Reihenfolge des Rechners extrahieren | Seite 180 | x | | |
| inet_ntoa() | Internet-Rechneradresse in eine Zeichenkette gemäß der üblichen Internet-Punktschreibweise konvertieren | Seite 180 | x | | |
| inet_pton() | Konvertiert <ul style="list-style-type: none"> – eine IPv4-Adresse in dezimaler Punkt-Notation oder – eine IPv6-Adresse in sedezipimaler Doppelpunkt-Notation in die entsprechende binäre Adresse. | Seite 183 | x | x | |
| inet_ntop() | Konvertiert eine binäre IPv4-oder IPv6-Adresse in die entsprechende <ul style="list-style-type: none"> – IPv4-Adresse in dezimaler Punktform bzw. – IPv6-Adresse in sedezipimaler Doppelpunkt-Notation. | Seite 183 | x | x | |

Hilfsfunktionen

| Funktion | Beschreibung | siehe | INET | INET6 | ISO |
|--------------------|---|---------------------------|------|-------|-----|
| freeaddrinfo() | Speicherbereich für <i>addrinfo</i> -Struktur freigeben, der von der Funktion <i>getaddrinfo()</i> angefordert wurde. | Seite 133 | x | x | |
| freehostent() | Speicherbereich für <i>hostent</i> -Struktur freigeben, der von den Funktionen <i>getipnodebyaddr()</i> und <i>getipnodebyname()</i> angefordert wurde. | Seite 134 | x | x | |
| if_freenameindex() | Speicherbereich für Array mit <i>if_nameindex()</i> Struktur(en) freigeben, der von der Funktion <i>if_nameindex()</i> angefordert wurde. | Seite 176 | x | x | |
| htonl() | 32 bit große Felder von Rechner- in Netz-Byte-Reihenfolge umsetzen | Seite 128 | x | | |
| htons() | 16 bit große Felder von Rechner- in Netz-Byte-Reihenfolge umsetzen | Seite 128 | x | x | |
| if_indextoname() | Zum Index den korrespondierenden Namen ermitteln | Seite 177 | x | x | |
| if_nametoindex() | Zum Namen den korrespondierenden Index ermitteln | Seite 179 | x | x | |
| ntohl() | 32 bit große Felder von Netz- in Rechner-Byte-Reihenfolge umsetzen | Seite 128 | x | | |
| ntohs() | 16 bit große Felder von Netz- in Rechner-Byte-Reihenfolge umsetzen | Seite 128 | x | x | |

Steuerfunktionen

| Funktion | Beschreibung | siehe | INET | INET6 | ISO |
|------------|--|---------------------------|------|-------|-----|
| soc_ioctl | Sockets steuern | Seite 212 | x | x | x |
| soc_wake() | eine mit <i>select()</i> oder <i>soc_poll()</i> wartende Task wecken | Seite 228 | x | x | x |

Testmakros für AF_INET6

Die folgenden Testmakros sind in <netinet.in.h> definiert:

| Makro | Test |
|-------------------------|------------------------------|
| IN6_IS_ADDR_UNSPECIFIED | Adresse = 0 ? |
| IN6_IS_ADDR_LOOPBACK | Adresse = Loopback ? |
| IN6_IS_ADDR_LINKLOCAL | Adresse = IPv6 - LINKLOCAL ? |
| IN6_IS_ADDR_SITELOCAL | Adresse = IPv6 - SITELOCAL ? |
| IN6_ADDR_V4COMPAT | Adresse = IPv4-kompatibel ? |
| IN6_ADDR_V4MAPPED | Adresse = IPv4-mapped |
| IN6_ARE_ADDR_EQUAL | Adresse1 = Adresse2 ? |

6.3 Beschreibung der Funktionen

In diesem Abschnitt sind alle Benutzerfunktionen der SOCKETS(BS2000)-Schnittstelle in alphabetischer Reihenfolge beschrieben.

accept() - Eine Verbindung über einen Socket annehmen

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

#include <netinet.in.h> /* nur bei AF_INET und AF_INET6 */
#include <iso.h>        /* nur bei AF_ISO */

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

int s;
int *addrlen;

struct sockaddr_in *addr; /* nur bei AF_INET */
struct sockaddr_in6 *addr; /* nur bei AF_INET6 */
struct sockaddr_iso *addr; /* nur bei AF_ISO */

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

Beschreibung

Mit der Funktion *accept()* nimmt die Server-Task über den Socket *s* eine Verbindung an, die von einem Client mit der Funktion *connect()* angefordert wurde.

Damit *accept()* für den Socket *s* aufgerufen werden kann, müssen folgende Voraussetzungen erfüllt sein:

- *s* muss ein Stream-Socket sein (SOCK_STREAM), dem mit *bind()* ein Name (Adresse) zugeordnet wurde.
- *s* muss mit *listen()* als „abhörend“ markiert, d.h. für die Annahme von Verbindungsanforderungen zugelassen sein.

addr zeigt bei Rückkehr von *accept()* auf die Adresse der Partneranwendung, wie sie auf der Kommunikationsebene bekannt ist. Das exakte Format der zurückgelieferten Adresse wird durch die Domäne bestimmt, in der die Kommunikation stattfindet:

- Für die Adressfamilie AF_INET ist die zurückgelieferte Adresse vom Typ *struct sockaddr_in* (siehe [Seite 25](#)).
- Für die Adressfamilie AF_INET6 ist die zurückgelieferte Adresse vom Typ *struct sockaddr_in6* (siehe [Seite 26](#)).
- Für die Adressfamilie AF_ISO ist die zurückgelieferte Adresse vom Typ *struct sockaddr_iso* (siehe [Seite 28](#)).

addrlen zeigt auf ein Integer-Objekt, das zum Zeitpunkt des Aufrufs von *accept()* die Länge von **addr* (in Bytes) enthalten muss. Bei Rückkehr der Funktion *accept()* enthält **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.
3. *accept()* liefert als Ergebnis den Deskriptor für den neuen Socket.

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

- Wenn der Socket als blockierend (Standardfall) markiert ist, blockiert *accept()* die aufrufende Task solange, bis eine Verbindung möglich ist.
- Wenn der Socket als nicht-blockierend markiert ist, liefert *accept()* eine Fehlermeldung mit *errno* = EWOULDBLOCK.

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

Nach erfolgreicher Ausführung von *accept()* ist in den Adressfamilien AF_INET und AF_INET6 die Verbindung bereits vollständig aufgebaut. In der Adressfamilie AF_ISO ist für den vollständigen Verbindungsaufbau zusätzlich einer der beiden folgenden Schritte erforderlich (siehe auch [Bild 4 auf Seite 63](#)):

- Benutzerdaten senden an den Partner, der die Verbindung angefordert hat
- *sendmsg()* (siehe [Seite 200](#)) aufrufen (mit oder ohne Senden von Benutzerdaten)

Nach erfolgreichem Verbindungsaufbau können über den mit *accept()* neu erzeugten Socket Daten mit dem Socket ausgetauscht werden, 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.

Fehleranzeige durch *errno*

EBADF

s ist kein gültiger Deskriptor.

EFAULT

Die Länge des Bereichs für die Aufnahme der Adresse ist zu klein.

EMFILE

Die maximale Anzahl offener Sockets ist erreicht.

ENETDOWN

Die Verbindung zum Netzwerk ist nicht mehr aktiv.

EOPNOTSUPP

Der referenzierte Socket ist nicht vom Typ SOCK_STREAM, oder der referenzierte Socket ist nicht mit *listen()* als „abhörend“ markiert worden.

EWOULDBLOCK

Der Socket ist als nicht-blockierend gekennzeichnet, und es stehen keine Verbindungsanforderungen an.

Siehe auch

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

bind() - Einem Socket einen Namen zuordnen

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

#include <netinet.in.h> /* nur bei AF_INET und AF_INET6 */
#include <iso.h>         /* nur bei AF_ISO */

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

struct sockaddr_in *name; /* nur bei AF_INET */
struct sockaddr_in6 *name; /* nur bei AF_INET6 */
struct sockaddr_iso *name; /* nur bei AF_ISO */

ANSI-C:
int bind(int s, struct sockaddr* name, int 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. *namelen* spezifiziert die Länge der Datenstruktur, die den Namen beschreibt.

Returnwert

- 0:
bei Erfolg
- 1:
bei Fehler. *errno* wird gesetzt, um den Fehler anzuzeigen.

Fehleranzeige durch errno

EADDRINUSE

Der angegebene Name wird bereits benutzt.

EADDRNOTAVAIL

Der angegebene Name kann vom lokalen System nicht an den Socket gebunden werden.

EBADF

s ist kein gültiger Deskriptor.

EFAULT

Die Länge des Bereichs für die Aufnahme der Adresse ist zu klein.

EINVAL

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

ENETDOWN

Die Verbindung zum Netzwerk ist nicht mehr aktiv.

Siehe auch

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

Byteorder-Makros - Byte-Reihenfolgen umsetzen

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

u_long htonl(u_long hostlong);
u_short htons(u_short hostshort);
u_long ntohl(u_long netlong);
u_short ntohs(u_short netshort);
```

Beschreibung

Die Makros *htonl()*, *htons()*, *ntohl()* und *ntohs()* werden nur in den Adressfamilien `AF_INET` und `AF_INET6` benötigt. *htonl()*, *htons()*, *ntohl()*, *ntohs()* setzen Bytes vom Datentyp *integer* oder *short* von der Rechner-Byte-Reihenfolge in die Netz-Byte-Reihenfolge um bzw. umgekehrt:

- *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.

Diese Makros werden meistens im Zusammenhang mit IPv4-Adressen und Portnummern verwendet, wie sie z.B. von der Funktion *gethostbyname()* geliefert werden (siehe [Seite 143](#)).

Für IPv6-Adressen hat man sich nach RFC 2553 für die garantierte Netz-Byte-Reihenfolge entschieden. Deshalb werden für die Adressfamilie `AF_INET6` nur die 16-bit-Byteorder-Makros für die Portnummer benötigt.

Die Makros werden nur an Systemen benötigt, an denen Rechner- und Netz-Byte-Reihenfolge unterschiedlich ist. Da im BS2000 Rechner-Byte-Reihenfolge und Netz-Byte-Reihenfolge identisch sind, werden die Makros in der Include-Datei `<netinet.in.h>` als Null-Makros (Makros ohne Funktion) bereitgestellt.

Für die Erstellung portabler Programme ist die Verwendung der Byteorder-Makros jedoch dringend zu empfehlen.

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()`, `getservbyname()`

connect() - Verbindung über einen Socket initiieren

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

#include <netinet.in.h> /* nur bei AF_INET und AF_INET6 */
#include <iso.h>        /* nur bei AF_ISO */

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

int s;
int namelen;

struct sockaddr_in *name; /* nur bei AF_INET */
struct sockaddr_in6 *name; /* nur bei AF_INET6 */
struct sockaddr_iso *name; /* nur bei AF_ISO */

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

Beschreibung

Mit *connect()* initiiert eine Task über einen Socket *s* vom Typ `SOCK_STREAM` die Kommunikation mit einem Partner-Socket. Wenn der Partner-Socket vom Typ `SOCK_DGRAM` ist, wird die Partner-Information nur im Socket *s* abgelegt.

Der Parameter *s* bezeichnet den Socket, über den die Task die Kommunikation mit einem anderen Socket initiiert. *name* ist ein Zeiger auf die Adresse des Kommunikationspartners. Der Kommunikationspartner ist ein Socket, der der gleichen Adressfamilie angehört. In der Adressfamilie `AF_ISO` ist dies zwingend erforderlich.

Zwischen den Adressfamilien `AF_INET` und `AF_INET6` ist mithilfe von IPv4-Mapped-IPv6-Adressen eine Kommunikation in beiden Richtungen möglich. Somit kann ein `AF_INET`-Socket auf einem Rechner, der nur IPv4-Adressen besitzt, eine Verbindung herstellen mit einem `AF_INET6`-Partnersocket auf einem Rechner, der teilweise oder ausschließlich IPv6-Adressen besitzt.

**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.

Die Funktionalität von *connect()* wird im Detail durch die verwendete Adressfamilie festgelegt:

connect() bei *AF_INET* und *AF_INET6*

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 Partner herzustellen. Der Partner wird durch den Parameter *name* spezifiziert. Mit *connect()* initiiert z.B. eine Client-Task 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 eine Task mit *connect()* den Namen des Kommunikationspartners (Parameter *name*) fest, mit dem der Datenaustausch erfolgen soll. An diesen Kommunikationspartner sendet die Task dann die Datagramme. Außerdem ist dieser Kommunikationspartner der einzige Socket, von dem die Task Datagramme empfangen kann.

Wird sowohl eine IP-Adresse als auch ein Port ungleich 0 angegeben, erzeugt das Transportsystem eine Route und ordnet dieser ein lokales Interface zu. Dieses lokale Interface kann mit *getsockname()* abgefragt werden.

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.

connect() bei *AF_ISO*

Mit *connect()* wird die Verbindung zu einem ISO-Partner hergestellt. Der Partner muss die Verbindungsanforderung nicht nur mit *accept()* annehmen, sondern muss außerdem zur Bestätigung eine Übertragungsfunktion (*send()* oder *sendmsg()*) aufrufen, bei der allerdings nicht unbedingt Daten übertragen werden müssen. Dies lässt sich z.B. mit einem *sendmsg()*-Aufruf erreichen.

Auch hier kann die Verbindung zweier Endpunkte nur einmal durch *connect()* hergestellt werden.

Returnwert

0:

bei Erfolg

-1:

bei Fehler. *errno* wird gesetzt, um den Fehler anzuzeigen.

Fehleranzeige durch errno**EADDRINUSE**

Die angegebene Adresse wird bereits benutzt.

EAFNOSUPPORT

Adressen in der angegebenen Adressfamilie sind inkompatibel mit der Adressfamilie des Sockets.

EBADF

s ist kein gültiger Deskriptor.

ECONNREFUSED

Der Verbindungsversuch wurde zurückgewiesen. Der angeforderte Service war zum Zeitpunkt des Funktionsaufrufs vermutlich nicht verfügbar.

EFAULT

Die Länge des Bereichs für die Aufnahme der Adresse ist zu klein.

EINPROGRESS

Der Verbindungsaufbau wurde noch nicht erfolgreich beendet.

EISCONN

Der Socket hat bereits eine Verbindung.

ENETDOWN

Die Verbindung zum Netzwerk ist nicht mehr aktiv.

Hinweis

Wird die Verbindung mit einem nicht-blockierenden Socket vom Typ `SOCK_STREAM` aufgebaut (entweder mit `soc_ioctl()` `NONBLOCKING` gesetzt oder durch Nutzung einer externen Börse), kann es mit einer Anwendung, die mit Sockets \geq V2.6 produziert wurde, zu einem Returnwert -1 mit der `errno` `EINPROGRESS` kommen. Das bedeutet, dass die Verbindung zum Zeitpunkt der Rückgabe der Kontrolle an den Aufrufer noch nicht erfolgreich aufgebaut worden ist. Bevor dieser Socket benutzt wird, prüfen Sie daher mit `select()` oder `soc_poll()` die Schreibfähigkeit.

Wenn ein Schreib-/Lese-Zugriff erfolgt, bevor die Verbindung vollständig aufgebaut ist, wird dieser mit einem Returnwert -1 und der `errno` `EWOLDBLOCK` abgewiesen.

Siehe auch

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

freeaddrinfo() - Speicher für addrinfo-Struktur freigeben

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

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

struct addrinfo *ai;

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

Beschreibung

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

Der Parameter *ai* ist ein Zeiger 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, AI_NUMERICHOST */
    int          ai_family;    /* PF_INET, PF_INET6 */
    int          ai_socktype;  /* SOCK_STREAM, SOCK_DGRAM */
    int          ai_protocol;  /* 0 (in SOCKETS nicht unterstützt) */
    size_t       ai_addrlen;   /* Länge der Adresse */
    char*        ai_canonname; /* Kanonischer Name des Knotens */
    struct sockadr *ai_addr;   /* Socket-Adress-Struktur d. Adress-
                               /* familie AF_INET oder AF_INET6 */
    struct addrinfo *ai_next; /* Nächste Struktur der verketteten Liste */
};
```

Siehe auch

[getipnodebyname\(\)](#), [getipnodebyaddr\(\)](#)

freehostent() - Speicher für hostent-Struktur freigeben

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

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

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

Beschreibung

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

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

Die Struktur *hostent* ist wie folgt deklariert:

```
struct hostent {  
    char *h_name;           /* Socket-Hostname */  
    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 */  
};
```

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

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

```
Kernighan-Ritchie-C:  
char* gai_strerror(rcode);
```

```
int rcode;
```

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

Beschreibung

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

Returnwert

gai_strerror() liefert einen Zeiger auf den String zurück, der den erklärenden Text enthält. Stimmt der Wert von *rcode* mit keinem der in `<netdb.h>` für *getaddrinfo()* definierten Error-Codes überein, dann ist der Returnwert ein Zeiger auf einen String, der einen Hinweis auf einen unbekanntes Fehler enthält.

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

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

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

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

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

Beschreibung

Die Funktion *getaddrinfo()* ermöglicht die protokollunabhängige Abfrage von Rechnerinformationen für die Adressfamilien AF_INET und AF_INET6.

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* kann optional eine *addrinfo*-Struktur übergeben werden. 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, AI_NUMERICHOST*/
                                   /* AI_NUMERICSERV, AI_V4MAPPED, AI_ALL */
                                   /* AI_NUMERICHOST*/
    int             ai_family;        /* PF_INET, PF_INET6 */
    int             ai_socktype;      /* SOCK_STREAM, SOCK_DGRAM*/
    int             ai_protocol;      /* 0 (in SOCKETS nicht unterstützt) */
    size_t          ai_addrlen;       /* Länge der Adresse */
    char*           ai_canonname;     /* Kanonischer Name des Knotens */
    struct sockaddr *ai_addr;         /* Socket-Adress-Struktur der Adress-*/
                                   /* familie AF_INET oder AF_INET6 */

    struct addrinfo *ai_next;         /* Zeiger auf das nächste Objekt der
                                   verketteten Liste */};
```

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

Mit den Werten für die *addrinfo*-Komponenten *ai_flags*, *ai_family* und *ai_socktype* wird eine Auswahl festgelegt:

- *ai_family* = PF_UNSPEC bedeutet, dass jede Protokollfamilie gewünscht wird.
- *ai_socktype* = 0 bedeutet, dass für jeden Socket-Typ eine *addrinfo*-Struktur mit dem gewünschten Service angelegt werden soll.
- *ai_flags* = AI_PASSIVE bedeutet, dass die zurückgelieferte Sockets-Adress-Struktur für einen *bind()*-Aufruf verwendet werden soll. Wenn *nodename* = NULL ist (siehe oben), wird das IP-Adresselement 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()* mindestens die erste zurückgegebene *addrinfo*-Struktur im Element *ai_canonname* den Zeiger auf den mit einem Null-Byte abgeschlossenen kanonischen Namen des ausgewählten Rechners.



Der *ai_canonname* wird durch einen Reverse Lookup ermittelt. Falls dieser Reverse Lookup nicht erfolgreich ist, also zur mitgegebenen Adresse kein Name gefunden wurde, wird kein Fehler gemeldet, sondern im Element *ai_canonname* wird der Inhalt des Parameters *nodename* hineinkopiert, wenn er ungleich dem Null-Zeiger ist. Ist *nodename* ein Null-Zeiger, wird im Element *ai_canonname* ein Null-Zeiger eingetragen.

Bitte beachten Sie, dass der Inhalt von *nodename* auch eine Adresse sein kann! Auch diese wird dann kopiert.

- Wenn in den *ai_flags* der *hints*-Struktur das Bit `AI_NUMERICHOST` gesetzt ist, muss ein vom Null-Zeiger verschiedener *nodename* 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.
- Wenn in den *ai_flags* der *hints*-Struktur das Bit `AI_V4MAPPED` in Verbindung mit *ai_family* = `PF_INET6` gesetzt ist und keine IPv6-Adressen für den Namen geliefert werden, dann werden in der Ausgabeliste vorhandene IPv4-Adressen in Form einer IPv4-mapped IPv6-Adresse eingetragen.
Ist auch das Bit `AI_ALL` gesetzt, werden sowohl IPv6- als auch die IPv4-mapped IPv6-Adressen eingetragen.
- Wenn in den *ai_flags* der *hints*-Struktur das Bit `AI_ADDRCONFIG` gesetzt ist, dann werden zum Namen gehörige IPv4- oder IPv6-Adressen nur ausgegeben, wenn eine entsprechende Interface-Adresse auf dem lokalen Rechner definiert ist. Die Loopback-Adresse zählt hier nicht als konfigurierte Interface-Adresse.
- Wenn in den *ai_flags* der *hints*-Struktur das Bit `AI_NUMERICSERV` gesetzt ist, dann muss der von Null verschiedene Zeiger von *servname* auf eine numerische Portnummern-Zeichenkette verweisen. Ist das nicht der Fall, wird eine Fehlermeldung (`EAI_NONAME`) zurückgegeben.

hints = `NULL` bewirkt das Gleiche wie eine mit 0 initialisierte *addrinfo*-Struktur und *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, wobei das Element *ai_next* = `NULL` das letzte Kettenelement kennzeichnet. 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.

Fehlercode in <netdb.h> definiert:

EAI_ADDRFAMILY

Die Internet-Adressfamilien werden 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 Rechnernamen-Information

EAI_FAMILY

Die Protokoll-Familie wird nicht unterstützt.

EAI_MEMORY

Fehler bei Speicheranforderung

EAI_NODATA

Keine zum Rechnernamen korrespondierende Adressen 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.

In SOCKETS(BS2000) ist PF_UNSPEC = AF_UNSPEC.

Wenn DNS nicht genutzt wird, geben Sie keinen *full-qualified-domain-name* an, sondern nur den Rechnernamen, um die BCAM-Prozessor-Tabelle zu nutzen (z.B. *host* statt *host.mydomain.net*).

Für den Zugang zum DNS wird der für SOCKETS(BS2000), BCAM und POSIX-Sockets gemeinsame Resolver LWRESO genutzt. Näheres siehe Handbuch „[BCAM Band 1/2](#)“.

getbcamhost() - BCAM-Hostnamen abfragen

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

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

char *bcamname;
int bcamnamelen;

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

Beschreibung

Die Verwendung der Funktion *getbcamhost()* ist nur in der Adressfamilie AF_ISO sinnvoll.

getbcamhost() liefert im Parameter *bcamname* den BCAM-Hostnamen zurück. Der BCAM-Hostname wird bei der Nutzung des ISO-Transportservice in der Adressfamilie AF_ISO verwendet und entspricht dem lokalen Netzselektor NSEL.

Beim Aufruf von *getbcamhost()* muss im Parameter *bcamnamelen* die Länge der String-Variablen *bcamname* spezifiziert werden.

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

Definition BCAM-Hostname: siehe *getsockopt()*

Returnwert

- 0:
bei Erfolg
- 1:
bei Fehler. *errno* wird nicht gesetzt.

getdtablesize() - Größe der Deskriptortabelle abfragen

```
#include <sys.socket.h>
Kernighan-Ritchie-C:
int getdtablesize();

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

Beschreibung

Die Funktion *getdtablesize()* liefert die Größe der Socket-Deskriptortabelle in Bits zurück. Diese Deskriptortabelle gilt für alle unterstützten Adressfamilien, d.h. sie enthält alle möglichen Deskriptoren über alle Adressfamilien hinweg.

Returnwert

>0:
bei Erfolg.

-1:
bei Fehler. *errno* wird nicht gesetzt.

Siehe auch

select()

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

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

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

```
char *addr;
int len;
int type;
```

```
struct hostent *gethostbyname(name);
```

```
char *name;
```

```
ANSI-C:
```

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

Beschreibung

Die Verwendung der Funktionen *gethostbyaddr()* und *gethostbyname()* ist nur in der Adressfamilie AF_INET sinnvoll.

gethostbyaddr() und *gethostbyname()* liefern aktuelle Informationen über die im Netz bekannten Rechner. Dabei beschaffen sich *gethostbyaddr()* und *gethostbyname()* die benötigten Informationen (Rechnername bzw. Rechneradresse) von einem DNS-Server. Nur falls dies nicht erfolgreich ist, wird die Information aus den BCAM-Tabellen und -Strukturen ermittelt (siehe Handbuch „[BCAM Band 1/2](#)“).

Bei *gethostbyaddr()* ist *addr* ein Zeiger auf die Rechneradresse. Die Rechneradresse muss in binärem Format mit der Länge *len* vorliegen. Als Angabe für *type* ist nur AF_INET zulässig.

Bei *gethostbyname()* muss für *name* der Rechnername angegeben werden.

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

Die Struktur *hostent* ist wie folgt deklariert:

```
struct hostent {
    char *h_name;           /* Socket-Hostname */
    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 */
```

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)

h_length

Länge der Adresse (in Bytes)

****h_addr_list**

Ein Zeiger auf eine durch Null abgeschlossene Liste von Netzadressen für den Rechner. Diese Adressen der Länge *h_length* werden in Netz-Byte-Reihenfolge geliefert.

Returnwert

Zeiger auf ein Objekt vom Typ *struct hostent*. Im Fehlerfall wird der Null-Zeiger zurückgeliefert.

Hinweis

Die Informationen des zurückgelieferten *hostent*-Objekts befinden sich in einem statischen Bereich, der bei jedem neuen *gethostby...()*-Aufruf überschrieben wird. Die Informationen müssen somit kopiert werden, wenn sie gesichert werden sollen.

Ab der Version V2.2 wird mit *gethostbyname()* und *gethostbyaddr()* für den Zugang zum DNS der für SOCKETS(BS2000), BCAM und POSIX-Sockets gemeinsame Resolver LWRESO genutzt. Näheres siehe Handbuch, „[BCAM Band 1/2](#)“.

Die Nutzung des POSIX-Resolver-Dämons *dnssd* entfällt.

gethostname() - Namen des Host abfragen

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

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

char *name;
int namelen;

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

Beschreibung

Die Verwendung der Funktion *gethostname()* ist nur in den Adressfamilien AF_INET und AF_INET6 sinnvoll.

gethostname() liefert im Parameter *name* den eigenen Socket-Hostnamen zurück. Socket-Hostnamen sind netzweit eindeutige Namen, die in TCP/IP-Netzen für alle Verarbeitungsrechner vergeben werden, die über eine TCP/IP-Route erreichbar sind (siehe Handbuch „[BCAM Band 1/2](#)“).

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 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 nicht gesetzt.

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

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

Kernighan-Ritchie-C:

```
struct hostent *getipnodebyaddr(addr, len, type, error_num);
```

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

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

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

ANSI-C:

```
struct hostent* getipnodebyaddr(char* addr, int len, int type, int*
error_num);
```

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

Beschreibung

Die Verwendung der Funktionen *getipnodebyaddr()* und *getipnodebyname()* ist nur in den Adressfamilien AF_INET und AF_INET6 sinnvoll. *getipnodebyaddr()* und *getipnodebyname()* sind Erweiterungen der Funktionen *gethostbyaddr()* und *gethostbyname()* für die IPv6-Unterstützung.

getipnodebyaddr() und *getipnodebyname()* liefern aktuelle Informationen über die im Netz bekannten Rechner. Dabei beschaffen sich *getipnodebyaddr()* und *getipnodebyname()* die benötigten Informationen (Rechnername bzw. Rechneradresse) von einem DNS-Server. Nur falls dies nicht erfolgreich ist, wird die Information in BCAM ermittelt (siehe Handbuch „[BCAM Band 1/2](#)“).

Bei `getipnodebyaddr()` ist `addr` ein Zeiger auf die Rechneradresse. Die Rechneradresse muss in binärem Format mit der Länge `len` vorliegen. Als Angabe für `type` ist nur `AF_INET` oder `AF_INET6` zulässig.

Bei `getipnodebyname()` muss für `name` der Rechnername (Socket-Hostname) angegeben werden. Geben Sie den Namen an

- in Form eines Full-Qualified-DNS-Namens, also mit Rechnernamen- und Domain-Anteil (z.B. Rechnername.company.com) oder
- als teilqualifizierter `DNS_Name` (z.B. Rechnername.) oder
- nur als Rechnername (z.B. Rechnername).

Es ist außerdem möglich, eine IPv4-Adresse in dezimaler Punkt-Notation oder eine IPv6-Adresse in sedezimaler Doppelpunkt-Notation anzugeben. Dazu müssen die jeweils korrespondierenden Adressfamilien bei `af` angegeben werden. In der `hostent`-Rückgabe-Struktur wird dann die umgesetzte Binär-Adresse zurückgeliefert. Wird eine IPv4-Adresse in dezimaler Punkt-Notation und `af = AF_INET6` und `flags = AI_V4MAPPED` angegeben, dann wird in der Ausgabe-Struktur eine binäre IPv4-Mapped-IPv6-Adresse zurückgegeben.

Im Parameter `af` wird beim Aufruf die Adressfamilie spezifiziert: `AF_INET` oder `AF_INET6`.

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

In der Adressfamilie `af` können mit `flags` verschiedene Optionen angegeben werden (sie sind in `<netdb.h>` definiert):

AI_V4MAPPED

Der Aufrufer akzeptiert IPv4-Mapped-Adressen, wenn keine IPv6-Adresse zur Verfügung steht.

AI_ALL

Sofern vorhanden, werden IPv6-Adressen und IPv4-Mapped-Adressen zurückgeliefert. `af` muss den Wert `AF_INET6` haben.

AI_ADDRCONFIG

Abhängig vom Wert von `af`, wird nur eine IPv6- oder IPv4-Adresse 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 und wenn der Rechner, auf dem die Funktion aufgerufen wird, eine IPv6-Interface-Adresse hat, wird für den angegebenen Rechnernamen eine IPv6-Adresse zurückgeliefert.
- Wenn der Rechner, auf dem die Funktion aufgerufen wird, nur eine IPv4-Interface-Adresse besitzt, wird eine IPv4-Mapped-IPv6-Adresse zurückgegeben.

Die Funktionen *getipnodebyaddr()* und *getipnodebyname()* liefern einen Zeiger auf ein Objekt der nachfolgend beschriebenen Struktur *hostent*. Für dieses Objekt wird der Speicher dynamisch angefordert und muss mit der Funktion *freehostent()* vom Aufrufer wieder freigegeben werden.

Die Struktur *hostent* ist wie folgt deklariert:

```
struct hostent {
    char *h_name;           /* Socket-Hostname */
    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 */
```

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

h_length

Länge der Adresse (in Bytes)

****h_addr_list**

Ein Zeiger auf eine durch Null abgeschlossene Liste von Netzadressen für den Rechner. Diese Adressen der Länge *h_length* werden in Netz-Byte-Reihenfolge geliefert.

Returnwert

Zeiger auf ein Objekt vom Typ *struct hostent*. Im Fehlerfall wird der Null-Zeiger zurückgeliefert und die Variable *errnum* mit einem der folgenden Werte versorgt. Diese Werte sind in der Include-Datei <netdb.h> definiert.

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.

Hinweis

Wenn DNS nicht genutzt wird, ist es in der Regel sinnvoll, keinen Fully Qualified Domain Name (FQDN), sondern nur den Rechnernamen anzugeben, um die entsprechenden Adressen von BCAM zu erhalten (z. B. *host* statt *host.mydomain.net*).

FQDNs sind an Systemen, an denen DNS nicht genutzt wird, nur dann sinnvoll, wenn es eine FQDN-Datei mit Einträgen gibt.

Für den Zugang zum DNS wird der für SOCKETS(BS2000), BCAM und POSIX-Sockets gemeinsame Resolver LWRESO genutzt. Näheres siehe Handbuch „[BCAM Band 1/2](#)“.

getnameinfo() - Namen des Kommunikationspartners abfragen

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

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

const struct sockaddr *sa;
socklen_t salen;
char *host;
size_t hostlen;
char *serv;
size_t servlen;
int flags;
```

```
ANSI-C:
int getnameinfo (const struct sockaddr* sa, socklen_t salen, char* host,
                size_t hostlen, char* serv, size_t servlen, int flags);
```

Beschreibung

Die Funktion *getnameinfo()* gibt den Namen, der der beim Aufruf angegebenen IP-Adresse und Portnummer zugeordnet ist, als Text-String zurück. Die Werte werden entweder über den DNS-Service oder über systemspezifische Tabellen ermittelt.

Der Parameter *sa* ist ein Zeiger auf eine *sockaddr_in*-Struktur (bei IPv4) oder eine *sockaddr_in6*-Struktur (bei IPv6), die jeweils die IP-Adressen und Portnummern enthalten. *salen* gibt die jeweilige Länge dieser Strukturen an.


Bei erfolgreicher Ausführung von *getnameinfo()* ist *host* ein Zeiger auf den zur angegebenen IP-Adresse korrespondierenden Socket-Hostnamen, der mit dem Null-Byte abgeschlossen wird und dessen Länge dem Wert von *hostlen* entspricht. Entsprechendes gilt für den zur angegebenen Portnummer korrespondierenden Service-Namen, auf den der Zeiger *serv* verweist und der inklusive Null-Byte die Länge *servlen* hat.

Mit dem Wert 0 für *hostlen* oder *servlen* beim Aufruf von *getnameinfo()* wird angezeigt, dass im zugehörigen Parameter *host* kein Name bzw. im Parameter *serv* kein Service-Name oder keine Portnummer zurückgeliefert werden soll. Für die gewünschte Information muss aber ein Puffer ausreichender Größe bereitgestellt werden, der Rechner- und Service-Namen inklusive Null-Byte aufnehmen kann.

Spezifikation der maximalen Längen von DNS- und Servicenamen in der Include-Datei <netdb.h>:

```
#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:

- Wenn das *flags*-Bit NI_NOFQDN gesetzt ist, wird nur der Rechnernamen-Anteil eines FQDN zurückgeliefert.
 -  Für den Zugang zum DNS wird der für SOCKETS(BS2000), BCAM und POSIX-Sockets gemeinsame Resolver LWRESO genutzt. Näheres siehe Handbuch „[BCAM Band 1/2](#)“.
- Wenn das *flags*-Bit NI_NUMERICHOST gesetzt ist oder der Rechnername weder im DNS noch durch lokale Information ermittelt werden kann, wird der numerische Hostname nach Adressumwandlung in abdruckbarer Form zurückgeliefert.
- Wenn das *flags*-Bit NI_NAMEREQD gesetzt ist, wird ein Fehler gemeldet, falls der Rechnername im DNS nicht ermittelt werden kann.
Wenn das *flags*-Bit NI_NAMEREQD in Kombination mit NI_NOFQDN gesetzt, hat das Bit keine Wirkung.
- Wenn das *flags*-Bit NI_NUMERICSERV gesetzt ist, wird anstatt des Service-Namens die Portnummer in abdruckbarer Form zurückgeliefert.
- Wenn das *flags*-Bit NI_DGRAM gesetzt ist, wird der Service-Name für das udp-Protokoll zurückgeliefert. Ohne Angabe von NI_DGRAM wird immer der Service-Name für das tcp-Protokoll zurückgeliefert.

Returnwert

0:
bei Erfolg

>0:
bei Fehler

Da für den DNS-Resolver Thread-Festigkeit gefordert ist, können keine *errno*s gesetzt werden.

Wenn der Returnwert > 0 ist, entspricht er dem Wert eines Fehlercodes EAI_XXX, wie sie in <netdb.h> definiert sind.

<0:
bei Fehler

Es ist ein Fehler aufgetreten, der die Ausführung der Funktion verhindert. Deshalb wird *errno* gesetzt.

getpeername() - Remote-Adresse der Sockets-Verbindung abfragen

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

#include <netinet.in.h> /* nur bei AF_INET und AF_INET6 */
#include <iso.h>        /* nur bei AF_ISO */

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

int s;
int *namelen;

struct sockaddr_in *name; /* nur bei AF_INET */
struct sockaddr_in6 *name; /* nur bei AF_INET6 */
struct sockaddr_iso *name; /* nur bei AF_ISO */

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

Beschreibung

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

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

Die Integer-Variable, auf die der Parameter *namelen* zeigt, muss vor Aufruf von *getpeername()* mit der maximal möglichen Adresslänge (angegeben in Bytes) versorgt werden. 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.

Fehleranzeige durch errno

EBADF

Der Parameter *s* ist kein gültiger Deskriptor.

EFAULT

Die Länge des Bereichs für die Aufnahme der Adresse ist zu klein.

ENOBUFS

Es gibt nicht genug Speicherplatz im Puffer.

ENOTCONN

Der Socket hat keine Verbindung.

EOPNOTSUPP

Der Socket *s* ist nicht vom Typ SOCK_STREAM. Die Operation wird für den Socket-Typ von *s* nicht unterstützt.

Siehe auch

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

getprotobyname() - Nummer des Protokolls abfragen

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

```
Kernighan-Ritchie-C:  
struct protoent *getprotobyname(name);  
char *name;
```

```
ANSI-C:  
struct protoent* getprotobyname(char* name);
```

Beschreibung

Die Verwendung der Funktion *getprotobyname()* ist nur in den Adressfamilien AF_INET und AF_INET6 sinnvoll.

getprotobyname() liefert einen Zeiger auf ein Objekt der nachfolgend beschriebenen Struktur *protoent* zurück, das die zum Protokollnamen *name* gehörende Protokollnummer enthält. Die Struktur *protoent* ist in <netdb.h> 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. Aliasnamen werden derzeit nicht unterstützt.

p_proto

Nummer des Protokolls. Ergebnisfeld von *getprotobyname()*.

Returnwert

Zeiger auf ein Objekt vom Typ *struct protoent*. Im Fehlerfall wird der Null-Zeiger zurückgeliefert.

Hinweis

Die Informationen des zurückgelieferten *protoent*-Objekts befinden sich in einem statischen Bereich, der bei jedem neuen *getprotobyname()*-Aufruf überschrieben wird. Die Informationen müssen kopiert werden, wenn sie gesichert werden sollen.

getservbyname(), getservbyport() - Informationen über Services abfragen

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

Kernighan-Ritchie-C:

```
struct servent *getservbyname(name, proto);
```

```
char *name;
```

```
char *proto;
```

```
struct servent *getservbyport(port, proto);
```

```
int port;
```

```
char *proto;
```

ANSI-C:

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

```
struct servent* getservbyport(int port, char* proto)
```

Beschreibung

Die Verwendung der Funktionen *getservbyname()* und *getservbyport()* ist nur in den Adressfamilien AF_INET und AF_INET6 sinnvoll.

getservbyname() und *getservbyport()* liefern Informationen über die verfügbaren Services aus der Services-Datei mit dem Standardnamen SYSDAT.BCAM.ETC.SERVICES, die von BCAM verwaltetet wird (siehe Handbuch „[BCAM Band 1/2](#)“).

Beide Funktionen liefern einen Zeiger auf ein Objekt der nachfolgend beschriebenen Struktur *servent*.

getservbyname() liefert in dem *servent*-Objekt die zum Service-Namen *name* und zum Protokoll *proto* gehörende Portnummer. Wird *proto* mit NULL angegeben, werden der Service-Name und die Portnummer des ersten in der Liste gefundenen Protokolls ausgegeben.

getservbyport() liefert in dem *servent*-Objekt den zur Portnummer *nummer* und zum Protokoll *proto* gehörenden Service-Namen, sowie die maximal vier möglichen eingetragenen Aliasnamen. Wird *proto* mit NULL angegeben, werden der Service-Name und die Aliasnamen des ersten in der Liste gefundenen Protokolls für die angegebene Portnummer ausgegeben.

Die Struktur *servent* ist in `<netdb.h>` 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.

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

Returnwert

Zeiger auf ein Objekt vom Typ *struct servent*. Im Fehlerfall wird der Null-Zeiger zurückgeliefert.

Hinweis

Die Informationen des zurückgelieferten *servent*-Objekts befinden sich in einem statischen Bereich. Die Informationen müssen somit kopiert werden, wenn sie gesichert werden sollen.

getsockname() - Local-Adresse der Sockets-Verbindung abfragen

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

#include <netinet.in.h> /* nur bei AF_INET und AF_INET6 */
#include <iso.h>        /* nur bei AF_ISO */

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

struct sockaddr_in *name; /* nur bei AF_INET */
struct sockaddr_in6 *name; /* nur bei AF_INET6 */
struct sockaddr_iso *name; /* nur bei AF_ISO */

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

Beschreibung

Die Funktion *getsockname()* liefert im Parameter *name* den aktuellen Namen für den Socket *s*. *name* zeigt auf einen Speicherbereich. **name* enthält nach erfolgreicher Ausführung von *getsockname()* den Namen (die Adresse) des Sockets *s*. Die Integer-Variable auf die der Parameter *namelen* zeigt, muss vor Aufruf von *getsockname()* mit der Adresslänge (in Bytes) versorgt werden. 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

Fehleranzeige durch errno**EBADF**

Der Parameter *s* ist kein gültiger Deskriptor.

EFAULT

Die Länge des Bereichs für die Aufnahme der Adresse ist zu klein.

EOPNOTSUPP

Der Socket *s* ist nicht vom Typ SOCK_STREAM. Die Operation wird für den Socket-Typ von *s* nicht unterstützt.

Siehe auch

bind(), getpeername(), socket()

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

```
#include <sys.types.h>
#include <sys.socket.h>
#include <netinet.in.h> /* nur bei AF_INET oder AF_INET6 */
```

Kernighan-Ritchie-C:

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

```
int s;
int level;
int optname;
char *optval;
int *optlen;
```

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

```
int s;
int level;
int optname;
char *optval;
int optlen;
```

ANSI-C:

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

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

Beschreibung

Mit der Funktion *getsockopt()* kann der Benutzer über die Parameter *optname*, *optval* und *optlen* die Eigenschaften (Optionen) der Socket-Schnittstelle oder eines einzelnen Sockets *s* abfragen.

Mit der Funktion *setsockopt()* kann der Benutzer über die Parameter *optname*, *optval* und *optlen* die Eigenschaften (Optionen) der Socket-Schnittstelle oder eines einzelnen Sockets *s* ändern.

Ob die Eigenschaften der Socket-Schnittstelle oder eines einzelnen Sockets abgefragt bzw. geändert werden sollen, bestimmt der Benutzer über den Parameter *level*.

Als aktueller Wert für *level* sind folgende Werte zulässig:

SOL_GLOBAL ([Seite 162](#))

Diagnose-Ausgaben werden aktiviert.

SOL_SOCKET ([Seite 163](#))

Optionen des Sockets *s* der Adressfamilie AF_INET oder AF_INET6 werden abgefragt bzw. geändert.

SOL_TRANSPORT ([Seite 173](#))

Optionen des Sockets *s* der Adressfamilie AF_ISO werden abgefragt bzw. geändert.

IPPROTO_TCP ([Seite 172](#))

Optionen des Sockets *s* in der Adressfamilie AF_INET oder AF_INET6 zur Modifikation des Protokollverhaltens des TCP-Protokolls werden abgefragt oder geändert.

IPPROTO_IPV4 ([Seite 167](#))

Optionen des Sockets *s* in der Adressfamilie AF_INET zur Modifikation des Protokollverhaltens des IPv4-Protokolls werden abgefragt oder geändert. Aus Kompatibilitätsgründen wird IPPROTO_IP wie IPPROTO_IPV4 bewertet.

IPPROTO_IPV6 ([Seite 170](#))

Optionen des Sockets *s* in der Adressfamilie AF_INET6 zur Modifikation des Protokollverhaltens des IPv6-Protokolls werden abgefragt oder geändert.

IPPROTO_ICMP ([Seite 172](#))

Optionen des Sockets *s* in der Adressfamilie AF_INET vom Typ SOCK_RAW zur Modifikation des Protokollverhaltens des IP-Protokolls im Zusammenhang mit dem ICMP-Protokoll werden abgefragt oder geändert.

IPPROTO_ICMPV6 ([Seite 173](#))

Optionen des Sockets *s* in der Adressfamilie AF_INET6 vom Typ SOCK_RAW zur Modifikation des Protokollverhaltens des IPv6-Protokolls im Zusammenhang mit dem ICMPv6-Protokoll werden abgefragt oder geändert.

Optionen für Level SOL_GLOBAL

Im Level SOL_GLOBAL hat der Operand *s* keine Bedeutung und sollte den Wert 0 erhalten.

Mit *optname* = SO_DEBUG können Diagnoseausgaben in unterschiedlicher Detailtiefe aktiviert werden.

Mit *getsockopt()* und *optlen* mit einem Wert von 4 (*sizeof int*) wird in *optval* der Wert des zur Zeit eingestellten Diagnoseausgabe-Levels ausgegeben.

Mit dem Wert 0 ist die Diagnoseausgabe deaktiviert.

Mit `setsockopt()` und `optlen` mit einem Wert von 4 können in `optval` die in der Tabelle folgenden Diagnoseausgabe-Level festgelegt werden

*`optval` ≥ 1: Funktionsaufrufe

*`optval` ≥ 6: Funktionsaufrufe + Zusatzinformationen

*`optval` = 8: Funktionsaufrufe + Zusatzinformationen + BCAM-Parameterleiste nach Aufruf

*`optval` = 9: Funktionsaufrufe + Zusatzinformationen + BCAM-Parameterleiste nach Aufruf
+ BCAM-Parameterleiste vor Aufruf

Mit `setsockopt()` und `optname` = `SO_ASYNC` und `optlen` = 4 und `optval` mit einem Zeiger auf die Kurzkenung der Ereigniskennung der zu nutzenden externen Börse wird ein externer Wartepunkt für SOCKETS(BS2000) eingerichtet, wenn dies der erste Sockets-Aufruf des Subsystems ist (siehe [Seite 236](#))..

| <i>optname</i> | <i>*optlen</i> | Wertebereich von <i>optval</i> |
|----------------|----------------|--|
| SO_ASYNC | 4 | Zeiger auf Kurzkenung der Ereigniskennung (Event-ID) |

Optionen für Level `SOL_SOCKET` (`AF_INET`, `AF_INET6`)

In diesem Fall spezifiziert `s` den Socket, dessen Eigenschaften abgefragt bzw. geändert werden sollen. Für `optname` ist der Name der Option anzugeben, deren Wert abgefragt oder geändert werden soll.

Bei `getsockopt()` identifizieren `optval` und `optlen` jeweils einen Puffer, in dem der Wert der gewünschten Option(en) zurückgeliefert wird. `*optlen` enthält 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.

Bei `getsockopt()` können in der Adressfamilie `AF_INET` und `AF_INET6` für `optname` und `optlen` folgende Werte zurückgeliefert werden:

| <i>optname</i> | <i>*optlen</i> | Ausgabeformat von <i>optval</i> |
|-----------------|----------------|---------------------------------|
| SO_DEBUG | 4 | int |
| SO_DISHALIAS | 4 | int |
| SO_ERROR | 4 | int |
| SO_KEEPALIVE | 4 | int |
| SO_OUTPUTBUFFER | 4 | int |
| SO_RCVBUF | 4 | int |
| SO_SNDBUF | 4 | int |
| SO_TSTIPAD | 4 | int |

| <i>optname</i> | <i>*optlen</i> | Ausgabeformat von <i>optval</i> |
|----------------|----------------|---------------------------------|
| SO_TYPE | 4 | int |
| SO_VHOSTANY | 8 | *(char[8]) |



SO_KEEPALIVE

Ein Ausgabewert >0 gibt an, dass ein Timer-Wert mit *setsockopt()* eingestellt worden ist. Der Aufrufer kann den Verbindungsstatus für den gewählten Socket nicht erkennen. Ist die Verbindung aktiv, wird der Timerwert aus der Verbindungsinformation von BCAM ermittelt. Ist die Verbindung noch nicht aktiv, wird der Timerwert aus dem Socket ausgelesen.

Wird ein Timerwert = 0 ausgegeben, bedeutet dies nicht zwingend, dass KEEPALIVE ausgeschaltet ist! Eine globale Einstellung des KEEPALIVE-Timers durch die BCAM-Administration wird durch diese Funktion nicht sichtbar.

Bei *setsockopt()* ändern Sie über die Parameter *optval* und *optlen* Options-Werte. Dabei können Sie in den Adressfamilien AF_INET und AF_INET6 für *optname*, *optlen* und *optval* folgende Werte angeben:

| <i>optname</i> | <i>optlen</i> | <i>optval</i> |
|-------------------------------|--|---------------------------------|
| SO_BROADCAST (nur AF_INET) | 4 | |
| SO_DEBUG | 4 | $0 \leq \textit{optval} \leq 9$ |
| SO_DISHALIAS | 4 | 0, 1 |
| SO_KEEPALIVE | 4 | 0; 120...32767 |
| SO_LINGER | $\geq \text{sizeof}(\text{struct linger})$ | *(struct linger) |
| SO_REUSEADDR | 4 | 0, 1 |
| SO_VHOSTANY | 4 | *(char[9]) |

Der gültige Wertebereich von *optval* bei der Option SO_KEEPALIVE ist 0 und 120 ... 32767:

- Mit dem Wert 0 wird der Timer ausgeschaltet.
- Mit den Werten aus dem Bereich von 120 ... 32767 wird der Timer eingeschaltet und das Timer-Intervall mit dem angegebenen Wert (Einheit: Sekunden) eingestellt.

Bei Angabe eines Wertes außerhalb des gültigen Wertebereichs wird das Timer-Intervall mit dem Standardwert vom Transportsystem eingestellt.

SO_BROADCAST (nur AF_INET)

Diese Option hat für Sockets keine funktionelle Bedeutung. Es wird lediglich eine Syntaxprüfung durchgeführt.

Bei korrekter Syntax wird der Wert 0 zurückgeliefert, andernfalls der Wert -1.

SO_DEBUG

Bei *level* = SOL_GLOBAL legt diese Option den Diagnose-Level für die Sockets der aktiven Task fest.

Bei *level* = SOL_SOCKET hat die Option keine funktionale Bedeutung; es wird lediglich eine Syntaxprüfung durchgeführt.

Bei korrekter Syntax wird der Wert 0 zurückgeliefert, andernfalls der Wert -1.

SO_DISHALIAS

Mit dem Wert >0 wird im Socket eingetragen, dass mit dem *bind()*-Call für diese Anwendung das Host-Aliasing ausgeschaltet werden soll.

SO_ERROR

zeigt die Nummer des zuletzt ausgegebenen Fehlers an.

SO_KEEPALIVE

gibt an, ob für die aktuelle Verbindung die TCP-KEEPALIVE-Überwachung durchgeführt werden soll.

Im Einzelnen wird dabei spezifiziert,

- ob in der TCP-Protokollmaschine für die aktuelle Verbindung die KEEPALIVE-Überwachung aktiviert werden soll und
- welches Zeitintervall (in Sekunden) für diese Überwachung gewählt werden soll.

Die Wirkung von SO_KEEPALIVE hängt vom Status des zugehörigen Sockets ab:

- Besteht für den Socket noch keine aktive Verbindung, dann wird der Wunsch nach KEEPALIVE-Überwachung mit dem zugehörigen Wert des Timer-Intervalls in der Socket-Struktur vermerkt und beim Verbindungsaufbau an das Transportsystem mit übergeben.

Bei einer Server-Anwendung, d.h. im Fall eines passiven Verbindungsaufbaus, muss der aktive *listen()*-Socket mit SO_KEEPALIVE bearbeitet werden, damit bei jedem Verbindungsaufbau die Überwachung automatisch eingeschaltet wird.

- Besteht für den Socket bereits eine aktive Verbindung, dann wird die Information mithilfe eines internen Transportsystem-Aufrufs an die TCP-Protokollmaschine weitergereicht.

Mit dem Zeitintervall-Wert 0 wird die Überwachung deaktiviert:

- Bei einer bestehenden Verbindung wird die Überwachung sofort deaktiviert.

- Falls noch keine Verbindung besteht, wird die Überwachung beim Verbindungsaufbau deaktiviert. Im Server-Fall muss der *listen()*-Socket entsprechend markiert werden.



Aufgrund unterschiedlicher Implementierungen der TCP-Protokollmaschinen kann die Aufrechterhaltung der Verbindungen nicht garantiert werden (siehe auch RFC 1122).

SO_LINGER

Die Option `SO_LINGER` verwendet einen Parameter vom Datentyp *struct linger*. Dieser Parameter spezifiziert den gewünschten Status der Option und das Verzögerungsintervall.

Die Struktur *linger* ist in `<sys.socket.h>` definiert.

```
struct linger {
int l_onoff; /* option on/off */
int l_linger; /* linger time */
};
```

Der Parameter *l_linger* legt die maximale Zeit für die Ausführung von *soc_close()* fest, mit *l_onoff* wird die „Linger“-Funktion aktiviert und deaktiviert (0 = OFF, >0 = ON).

SO_OUTPUTBUFFER

Anzeige der von der Socket-Schnittstelle akzeptierten, aber noch nicht vom Partner-Transportsystem quittierten Benutzerdaten.

SO_RCVBUF

zeigt die Größe des Eingabepuffers an.

SO_REUSEADDR

Die Option hat keine funktionale Bedeutung, wenn die Anwendung mit der `SOCKETS(BS2000)`-Version \leq V2.1 produziert wurde oder auf `BCAM < V18` trifft; es wird dann lediglich eine Syntaxprüfung durchgeführt.

Mit einer Produktion ab `SOCKETS(BS2000)` V.2.2 wird die Funktionalität von `SO_REUSEADDR` im Rahmen der Multihoming-Unterstützung benötigt.

`SO_REUSEADDR` wirkt nur auf den angegebenen Socket und muss vor *bind()* gesetzt werden.

`SO_REUSEADDR` wird mit *optval* = 1 gesetzt und mit *optval* = 0 wieder zurückgesetzt.

Bei korrekter Syntax wird der Wert 0 zurückgeliefert, andernfalls der Wert -1.

SO_SNDBUF

zeigt die Größe des Ausgabepuffers an.

SO_TSTIPAD

vergleicht die übergebene IP-Adresse mit den Interface-Adressen des Socket-Hosts, auf dem die Socket-Anwendung läuft und teilt via *optval* das Vergleichsergebnis mit.

Die IP-Adresse wird mit dem Zeiger *optval* auf eine *struct in_addr* oder *struct in6_addr* als IPv4- oder IPv6-Adresse übergeben.

Der angegebene Socket muss nicht existieren. Es ist aber erforderlich, dass der File-Deskriptor im zulässigen Wertebereich liegt.

Der Wert des Parameters *optlen* spezifiziert, ob es sich um eine IPv4- oder IPv6-Adresse handelt. Somit muss der Wert von *optlen* je nach verwendetem Adresstyp der Länge von *struct in_addr* bzw. *struct in6_addr* entsprechen.

Als Rückgabewert werden die ersten 4 Byte der beim Aufruf übergebenen Adressstruktur überschrieben.

Rückgabewert von *optval*:

0: Die angegebene IP-Adresse ist eine eigene Interface-Adresse.

1: Die angegebene IP-Adresse ist keine eigene Interface-Adresse.

SO_TYPE

zeigt den Socket-Typ an.

SO_VHOSTANY

in der Zeichenfolge, auf die *optval* zeigt, wird der BCAM-Hostname des realen oder eines virtuellen Hosts angegeben, der dann im Socket eingetragen wird.

So ist es möglich, mit einer ANY- oder LOOPBACKADDR einen virtuellen Host zu adressieren, oder mit *soc_ioctl(..., SIOCGLIFCONF,...)* die Daten eines virtuellen Hosts auszulesen. Außerdem ist es möglich, auch einen realen Host zu adressieren, wenn unter einer Kennung gearbeitet wird, die durch einen Eintrag in der Application-Tabelle von BCAM einem virtuellen Host zugewiesen wurde.

Beim Lesen wird der im Socket eingetragene Name für den BCAM-Host ausgegeben.

BCAM-Hostname:

Er ist acht Zeichen lang. Es dürfen alphanumerische Zeichen und die Sonderzeichen #, @, \$ oder Leerzeichen am Namensende verwendet werden. In der Regel sollten Großbuchstaben verwendet werden, es wird aber Groß-/Kleinschreibung unterschieden. Ein nur numerischer Anteil ist nicht erlaubt.

Optionen für Level IPPROTO_IPV4 (AF_INET)

Bei *getsockopt()* können Sie in der Adressfamilie AF_INET für *optname* und *optlen* folgende Werte angeben:

| <i>optname</i> | <i>*optlen</i> | Ausgabeformat von <i>optval</i> |
|-------------------|--------------------------|---------------------------------|
| IP_MULTICAST_TTL | 4 | int |
| IP_MULTICAST_IF | ≥ sizeof(struct in_addr) | *(struct in_addr) |
| IP_MULTICAST_LOOP | 4 | int |

| <i>optname</i> | <i>*optlen</i> | Ausgabeformat von <i>optval</i> |
|----------------|----------------|---------------------------------|
| IP_RECVERR | 4 | int |

Rückgabewert für IP_MULTICAST_TTL:

Wert des eingestellten Hop-Limits.

Rückgabewert für IP_MULTICAST_IF:

IPv4-Adresse des Interfaces, über das gesendet werden soll.

Rückgabewert für IP_MULTICAST_LOOP:

0: Loopback OFF

1: Loopback ON

Rückgabewert für IP_RECVERR:

0: Das Flag ist nicht gesetzt

1: Das Flag ist gesetzt

Bei *setsockopt()* ändern Sie Options-Werte über die Parameter *optval()* und *optlen()*. Dabei können Sie in der Adressfamilie AF_INET folgende Werte angeben:

| <i>optname</i> | <i>optlen</i> | <i>optval</i> |
|--------------------|--|------------------------------|
| IP_ADD_MEMBERSHIP | $\geq \text{sizeof}(\text{struct ip_mreq})$ | *(struct ip_mreq) |
| IP_DROP_MEMBERSHIP | $\geq \text{sizeof}(\text{struct ip_mreq})$ | *(struct ip_mreq) |
| IP_MULTICAST_TTL | 4 | $0 < \text{optval} \leq 255$ |
| IP_MULTICAST_IF | $\geq \text{sizeof}(\text{struct in_addr})$ | *(struct in_addr) |
| IP_MULTICAST_LOOP | 4 | ≥ 0 |
| IP_RECVERR | 4 | ≥ 0 |

Die Optionen IP_ADD_MEMBERSHIP und IP_DROP_MEMBERSHIP verwenden einen Parameter vom Datentyp *struct ip_mreq*. Dieser Parameter spezifiziert die IPv4-Adresse der gewünschten Multicast-Gruppe und die lokale IPv4-Adresse.

Die Struktur *ip_mreq* ist in `<netinet.in.h>` definiert:

```
struct ip_mreq {
    struct in_addr imr_multiaddr; /* IP multicast address of group */
    struct in_addr imr_interface; /* local IP address of interface */
};
```

IP_MULTICAST_TTL

zeigt oder setzt den Multicast-Hop-Limit.

Hop-Limit-Werte:

- 0: Senden nur innerhalb des lokalen Hosts (Loopback)
- 1: Senden innerhalb des lokalen Subnetzes
- >1: Senden über Routergrenzen hinweg

IP_ADD_MEMBERSHIP

aktiviert die Zustellung von Nachrichten einer gewählten Multicast-Gruppe an diesen Socket. Angabe der Multicast- und der lokalen Interface-Adresse (IPv4-Adresse oder `INADDR_ANY`, nicht `INADDR_LOOPBACK`).
`INADDR_ANY` ist das Default-Interface von BCAM zum Empfang von Multicast-Daten.

IP_DROP_MEMBERSHIP

deaktiviert die Zustellung von Nachrichten einer gewählten Multicast-Gruppe an diesen Socket. Angabe der Multicast- und der lokalen Interface-Adresse (IPv4-Adresse oder `INADDR_ANY`, nicht `INADDR_LOOPBACK`).
`INADDR_ANY` ist das Default-Interface von BCAM zum Empfang von Multicast-Daten.

IP_MULTICAST_IF

IPv4-Adresse des Interfaces, über das gesendet werden soll.

IP_MULTICAST_LOOP

wird vom Sender der Nachrichten eingestellt und ermöglicht den Empfang auch auf dem lokalen, sendenden Host.
0: OFF, 1: ON (Default: ON)

IP_RECVERR

aktiviert die Zustellung von ICMP-Fehlermeldungen an diesen Socket, wenn die Option vor dem `bind()` gesetzt wurde.

Optionen für Level IPPROTO_IPV6 (AF_INET6)

Bei `getsockopt()` können Sie in der Adressfamilie AF_INET6 für `optname` und `optlen` folgende Werte angeben:

| <i>optname</i> | <i>*optlen</i> | Ausgabeformat von <i>optval</i> |
|---------------------|----------------|---------------------------------|
| IPV6_MULTICAST_HOPS | 4 | int |
| IPV6_MULTICAST_IF | 4 | int |
| IPV6_MULTICAST_LOOP | 4 | int |
| IPV6_RECVERR | 4 | int |
| IPV6_V6ONLY | 4 | int |

Rückgabewert für IPV6_MULTICAST_HOPS:
Wert des eingestellten Hop-Limits.

Rückgabewert für IPV6_MULTICAST_IF:
Index des Sender-Interfaces.

Rückgabewert für IPV6_MULTICAST_LOOP:
0: Loopback OFF
1: Loopback ON

Rückgabewert für IPV6_RECVERR und IPV6_V6ONLY:
0: Das Flag ist nicht gesetzt
1: Das Flag ist gesetzt

Bei `setsockopt()` können Sie in der Adressfamilie AF_INET6 für `optname` und `optlen` folgende Werte angeben:

| <i>optname</i> | <i>optlen</i> | <i>optval</i> |
|---------------------|---|------------------------------|
| IPV6_JOIN_GROUP | $\geq \text{sizeof}(\text{struct ipv6_mreq})$ | $\text{*(struct ipv6_mreq)}$ |
| IPV6_LEAVE_GROUP | $\geq \text{sizeof}(\text{struct ipv6_mreq})$ | $\text{*(struct ipv6_mreq)}$ |
| IPV6_MULTICAST_HOPS | 4 | 0..255 |
| IPV6_MULTICAST_IF | 4 | 1..255 |
| IPV6_MULTICAST_LOOP | 4 | ≥ 0 |
| IPV6_RECVERR | 4 | ≥ 0 |
| IPV6_V6ONLY | 4 | ≥ 0 |

Die Optionen IPV6_JOIN_GROUP und IPV6_LEAVE_GROUP verwenden einen Parameter vom Datentyp `struct ipv6_mreq`. Dieser Parameter spezifiziert die IPv6-Adresse der gewünschten Multicast-Gruppe und den Index des lokalen Interfaces.

Die Struktur *ipv6_mreq* ist in `<netinet.in.h>` definiert:

```
struct ipv6_mreq {
    struct in6_addr ipv6mr_multiaddr; /* IPv6 multicast addr */
    int    ipv6mr_interface;         /* interface index */
};
```

IPV6_MULTICAST_HOPS

zeigt oder setzt den Multicast-Hop-Limit.

Hop-Limit-Werte:

- 0: Senden nur innerhalb des lokalen Hosts (Loopback)
- 1: Senden innerhalb des lokalen Subnetzes
- >1: Senden über Routergrenzen hinweg

IPV6_JOIN_GROUP

aktiviert die Zustellung von Nachrichten einer gewählten Multicast-Gruppe an diesen Socket. Angabe der IPv6-Multicastadresse und des Indexes der lokalen Interface-Adresse (Index für IPv6-Adresse oder Index 0, kein Index für Loopback).

Index 0 steht für das Default-Interface von BCAM zum Empfang von Multicast-Daten.

IPV6_LEAVE_GROUP

deaktiviert die Zustellung von Nachrichten einer gewählten Multicast-Gruppe an diesen Socket. Angabe der IPv6-Multicastadresse und des Indexes der lokalen Interface-Adresse (Index für IPv6-Adresse oder Index 0, kein Index für Loopback).

Index 0 steht für das Default-Interface von BCAM zum Empfang von Multicast-Daten.

IPV6_MULTICAST_IF

Index des IPv6-Interfaces, über das gesendet werden soll.

IPV6_MULTICAST_LOOP

wird vom Sender der Nachrichten eingestellt und ermöglicht den Empfang auch auf dem lokalen sendenden Host.

0: OFF, 1: ON (Default: ON)

IPV6_RECVERR

aktiviert die Zustellung von ICMP-Fehlermeldungen an diesen Socket, wenn die Option vor dem *bind()* gesetzt wurde.

IPV6_V6ONLY

Mit der Option *IPV6_V6ONLY* ist es möglich, einen Socket auf die Nutzung von echten IPv6-Adressen zu beschränken (*optval* ≥ 1), wenn es vor dem *bind()* im Socket gesetzt wird. Dadurch kann man eine Serveranwendung anbieten, die einen Listen-Socket in den Domänen *AF_INET* und *AF_INET6* mit der gleichen Portnummer eröffnet.

Optionen für Level IPPROTO_TCP (AF_INET, AF_INET6)

Bei *getsockopt()* können Sie in den Adressfamilien AF_INET und AF_INET6 für *optname* und *optlen* folgende Werte angeben:

| <i>optname</i> | <i>*optlen</i> | Ausgabeformat von <i>optval</i> |
|----------------|----------------|---------------------------------|
| SO_TCP_NODELAY | 4 | int |

Bei *setsockopt()* können Sie über die Parameter *optval()* und *optlen()* Option-Werte ändern. Dabei können Sie in den Adressfamilien AF_INET und AF_INET6 folgende Werte angeben:

| <i>optname</i> | <i>optlen</i> | <i>optval</i> |
|----------------|---------------|------------------------------|
| SO_TCP_NODELAY | 4 | 1 oder 0 (rücksetzen/setzen) |
| TCP_DELAY | 4 | 1 oder 0 (rücksetzen/setzen) |

SO_TCP_NODELAY (TCP_NODELAY)

erlaubt das Abschalten des Nagle-Algorithmus des TCP-Protokolls. Wird diese Option im Socket (*connect()* oder *listen()*) gesetzt, wird die Aktion bei *connect()* oder dem Quittieren der Verbindung aktiviert.

Ist die Verbindung schon aufgebaut, wird diese Option sofort wirksam.

TCP_DELAY

Aus- oder Einschalten des Delayed-Ack-Timers einer Verbindung. Mit *optval > 0* werden die verzögerten Quittungen ausgeschaltet, mit *optval = 0* werden sie wieder eingeschaltet.

Die Verbindung muss schon bestehen.

Optionen für Level IPPROTO_ICMP (AF_INET)

Bei *setsockopt()* können Sie über die Parameter *optval()* und *optlen()* Option-Werte ändern. Dabei können Sie in der Adressfamilie AF_INET folgende Werte angeben:

| <i>optname</i> | <i>optlen</i> | <i>optval</i> |
|-----------------|---------------|------------------------------------|
| IP_TTL | 4 | 1...255 |
| IP_MTU_DISCOVER | 4 | IP_PMTUDISC_DO IP_PMTUDISC_DONT |

IP_TTL

ändert das Hop-Limit im entsprechenden Feld des IP-Protokoll-Headers des ICMP-Echo-Request-Pakets.

IP_MTU_DISCOVER

setzt das DF-Flag im IPv4-Protokoll-Header, das die Fragmentierung eines ICMP-Echo-Request-Pakets zulässt oder verbietet.

IP_PMTUDISC_DONT Fragmentieren erlaubt
 IP_PMTUDISC_DO Fragmentieren soll verhindert werden

Optionen für Level IPPROTO_ICMPV6 (AF_INET6)

Bei *setsockopt()* können Sie über die Parameter *optval()* und *optlen()* Option-Werte ändern. Dabei können Sie in der Adressfamilie AF_INET6 folgende Werte angeben:

| <i>optname</i> | <i>optlen</i> | Ausgabeformat in <i>optval</i> |
|-------------------|---------------|---------------------------------------|
| IPV6_HOPLIMIT | 4 | 1...255 |
| IPV6_MTU_DISCOVER | 4 | IP_PMTUDISC_DO IP_PMTUDISC_DONT |

IPV6_HOPLIMIT

ändert das Hop-Limit im entsprechenden Feld des IPv6-Protokoll-Headers des ICMPv6-Echo-Request-Pakets.

IPV6_MTU_DISCOVER

IP_PMTUDISC_DONT Fragmentieren erlaubt
 IP_PMTUDISC_DO Fragmentieren soll verhindert werden

Diese Funktion zeigt keine Wirkung, weil auf der IPv6-Protokollebene im Gegensatz zu IPv4 kein Flag vorgesehen ist, das die Fragmentierung verhindern könnte. Die Endsysteme sind nach der IPv6-Protokollvorschrift zum Fragmentieren verpflichtet.

Optionen für Level SOL_TRANSPORT (nur bei AF_ISO)

In diesem Fall spezifiziert *s* den Socket, dessen Eigenschaften abgefragt bzw. geändert werden sollen. Für *optname* ist der Name der Option anzugeben, deren Wert abgefragt oder geändert werden soll.

Bei *getsockopt()* identifizieren *optval* und *optlen* jeweils einen Puffer, in dem der Wert der gewünschten Option zurückgeliefert wird. **optlen* enthält 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.

Bei *getsockopt()* können in der Adressfamilie AF_ISO für *optname* und **optlen* folgende Werte zurückgeliefert werden:

| <i>optname</i> | <i>*optlen</i> | Ausgabeformat in <i>optval</i> |
|-----------------|----------------------------|--|
| TPOPT_CONN_DATA | 0..33 | string inkl. Null-Byte (Länge gemäß <i>optlen</i>) |
| TPOPT_CFRM_DATA | 0..33 | string inkl. Null-Byte (Länge gemäß <i>optlen</i>) |
| TPOPT_DISC_DATA | 0..33 | string inkl. Null-Byte (Länge gemäß <i>optlen</i>) |
| TPOPT_REDI_CALL | sizeof(struct cmsg_redhdr) | struct cmsg_redhdr |

In **optlen* wird die tatsächliche Länge der Verbindungsdaten angegeben. Der Wertebereich von *optlen* beträgt 0..33, da die maximale Länge der Verbindungsdaten 32 byte beträgt.

Mit *setsockopt()* können in der Adressfamilie AF_ISO abhängig vom Socket-Status die Verbindungsdaten in den Socket geschrieben werden. Die nachfolgend beschriebenen Optionen sind auf der Socket-Ebene bekannt.

Beschreibung der Socket-Optionen im Fall AF_ISO:

TPOPT_CONN_DATA

Der Socket *s* ist erzeugt und hat mit *bind()* eine Adresse erhalten, aber *connect()* wurde noch nicht ausgeführt:

Mit TPOPT_CONN_DATA als aktuellem Parameterwert für *optname* können die Verbindungsdaten in den Socket *s* eingetragen werden, die beim Aufruf von *connect()* an den Partner gesendet werden sollen.

TPOPT_CFRM_DATA

Für den Socket *s* ist eine Verbindungsanforderung eingetroffen, die der Socket *s* noch nicht akzeptiert hat:

Mit TPOPT_CFRM_DATA als aktuellem Parameterwert für *optname* können die Verbindungsdaten in den Socket *s* eingetragen werden, die für das Akzeptieren der Verbindung an den Partner gesendet werden sollen (siehe auch [Bild 4 auf Seite 63](#)).

TPOPT_DISC_DATA

Die Verbindung zum Partner-Socket ist vollständig aufgebaut:

Mit TPOPT_DISC_DATA als aktuellem Parameterwert für *optname* können die Verbindungsdaten in den Socket *s* eingetragen werden, die beim *soc_close()* an den Partner gesendet werden sollen.

In *optlen* wird die tatsächliche Länge der Verbindungsdaten angegeben. Der Wertebereich von *optlen* beträgt 1..32, da die maximale Länge der Verbindungsdaten 32 byte beträgt und als Mindestlänge 1 byte geschrieben werden muss.

TPOPT_REDI_CALL

wird für die Handoff-Prozedur benötigt

(siehe [Kapitel „Erweiterte Funktionen von SOCKETS\(BS2000\)“](#) auf Seite 73)

Returnwert

0:

bei Erfolg.

-1:

bei Fehler. *errno* wird gesetzt, um den Fehler anzuzeigen.

Fehleranzeige durch errno**ENOPROTOOPT**

Die Option wird durch das Protokoll nicht unterstützt.

Für *level*, *optname*, *optvalue* oder *optlen* wurde ein ungültiger Wert spezifiziert.

ENOTSOCK

Der Deskriptor *s* verweist nicht auf einen Socket.

Siehe auch

socket()

if_freenameindex() - Den durch if_nameindex() belegten dynamischen Speicher freigeben

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

void if_freenameindex(struct if_nameindex *ptr);
```

Beschreibung

Die Funktion *if_freenameindex()* wird benötigt, um den Speicher wieder freigeben zu können, der zur Returninformation von *if_nameindex()* dynamisch angefordert wird.

if_indextoname() - Interface-Index auf Interface-Namen umsetzen

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

char * if_indextoname(unsigned int ifindex, char * ifname);
```

Beschreibung

Die Funktion *if_indextoname()* gibt zum angegebenen Interface-Index den korrespondierenden Interface-Namen zurück. Dazu wird mit *ifname* ein Zeiger auf einen Puffer mit der Mindestlänge IF_NAMESIZE (in <net.if.h> enthalten) geliefert.

Returnwert

Bei erfolgreicher Ausführung liefert *if_indextoname()* den Interface-Namen zurück, der im Feld *ifname* abgelegt ist. Andernfalls wird ein NULL-Zeiger zurückgeliefert.

Fehleranzeige durch errno

ENXIO

Zum angegebenen Index wurde kein korrespondierender Interface-Name gefunden.

if_nameindex() - Liste von Interface-Namen mit dem dazugehörigen Interface-Index erzeugen

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

struct * if_nameindex if_nameindex(void);
```

Beschreibung

Die Funktion *if_nameindex()* erstellt ein Array mit den Interface-Namen und dem dazu gehörenden Interface-Index.

Für jedes vorhandene Interface wird eine Struktur *if_nameindex* angelegt. Die Struktur *if_nameindex* ist in *<net.if.h>* wie folgt deklariert:

```
struct if_nameindex {
    unsigned int    if_index;        /*1, 2, .....*/
    char *         if_name;        /* mit Null-Byte abgeschlossener Name*/
};
```

Returnwert

Als Ergebnis wird ein Array mit Strukturen vom Typ *if_nameindex* zurückgegeben. Das Ende ist dadurch gekennzeichnet, dass die letzte Struktur *if_nameindex* die Werte 0 für *if_index* und NULL für *if_name* enthält.

Im Fehlerfall wird ein NULL-Zeiger zurückgeliefert und *errno* entsprechend gesetzt.

Fehleranzeige durch *errno*

EINVAL

Es sind keine Interface-Informationen verfügbar

Hinweis

Der benötigte Speicher für das Array wird dynamisch angefordert und muss durch die Funktion *if_freenameindex()* wieder freigegeben werden.

if_nametoindex() - Interface-Name auf Interface-Index umsetzen

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

unsigned int if_nametoindex(const char * ifname);
```

Beschreibung

Die Funktion *if_nametoindex()* gibt zum angegebenen Interface-Namen den korrespondierenden Interface-Index zurück. *ifname* ist eine mit einem Null-Byte abgeschlossene Zeichenfolge mit dem Interfacenamen.

Returnwert

Bei erfolgreicher Ausführung liefert *if_nametoindex()* den Interface-Index zurück. Andernfalls wird 0 zurückgeliefert.

Fehleranzeige durch *errno*

Es sind keine Fehler definiert.

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

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

Kernighan-Ritchie-C:

```
unsigned long inet_addr(cp);
char *cp;
```

```
int inet_lnaof(in);
struct in_addr in;
```

```
struct in_addr inet_makeaddr(net, lna);
int net;
int lna;
```

```
int inet_netof(in);
struct in_addr in;
```

```
unsigned long inet_network(cp);
char *cp;
```

```
char *inet_ntoa(in);
struct in_addr in;
```

ANSI-C:

```
unsigned long inet_addr(char* cp);
int inet_lnaof(struct in_addr in);
struct in_addr inet_makeaddr(int net, int lna);
int inet_netof(struct in_addr in);
unsigned long inet_network(char* cp);
char* inet_ntoa(struct in_addr in);
```

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 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 Parameter *cp* zeigt, von der im Internet üblichen Punktschreibweise in einen ganzzahligen Wert. Dieser 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 -1 zurückgeliefert.
- Bei erfolgreicher Ausführung liefert *inet_network()* den Subnetz-Anteil der Internet-Adresse zurück. Andernfalls wird -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.

Fehleranzeige durch errno

Es sind keine Fehler definiert.

Hinweis

Der Return-Wert von *inet_ntoa()* zeigt auf statische Daten, die durch nachfolgende Aufrufe von *inet_ntoa()* überschrieben werden können. Die Informationen müssen kopiert werden, wenn sie gesichert werden sollen.

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

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

Kernighan-Ritchie-C:

```
char *inet_ntop(af, addr, dst, size);
```

```
int af;
char *addr;
char *dst;
int size;
```

```
int inet_pton(af, addr, dst);
```

```
int af;
char *addr;
char *dst;
```

ANSI-C:

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

Beschreibung

Die Verwendung der Funktionen *inet_ntop()* und *inet_pton()* ist nur in den Adressfamilien AF_INET und AF_INET6 sinnvoll.

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

- Falls in *af* der Wert AF_INET übergeben wird, wird eine binäre IPv4-Adresse in eine abdruckbare dezimale Punkt-Notation konvertiert.
- Falls in *af* der Wert AF_INET6 übergeben wird, wird eine binäre IPv6-Adresse eine abdruckbare sedezimale Doppelpunkt-Notation konvertiert.

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 Sie beim Aufruf von *inet_ntop()* die Integer-Konstanten `INET_ADDRSTRLEN` (für IPv4-Adressen) bzw. `INET6_ADDRSTRLEN` (für IPv6-Adressen) als aktuelle Werte für *size* verwenden. 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 spezifiziert der im Parameter *af* übergebene Wert, ob es sich um eine IPv4-Adresse oder um eine IPv6-Adresse handelt:

- Falls in *af* der Wert `AF_INET` übergeben wird, wird eine IPv4-Adresse konvertiert.
- Falls in *af* der Wert `AF_INET6` übergeben wird, wird eine IPv6-Adresse konvertiert.

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 dient, achten Sie darauf, dass die Anfangsadresse des Zielbereichs *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 Operand

`ENOSPC`
Ergebnispuffer zu klein

listen() - Socket auf anstehende Verbindungen überprüfen

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

Kernighan-Ritchie-C:
int listen(s, backlog);

int s;
int backlog;

ANSI-C:
int listen(int s, int backlog);
```

Beschreibung

Die Funktion *listen()* wird in den Adressfamilien AF_INET und AF_INET6 nur für Sockets vom Typ SOCK_STREAM sowie in der Adressfamilie AF_ISO unterstützt.

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. Allerdings wertet SOCKET(BS2000) den Parameter *backlog* derzeit nicht aus, sondern nimmt Verbindungsanforderungen solange an, bis die maximale Anzahl verfügbarer Sockets belegt ist.

Damit eine Task 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
4. In AF_ISO ist darüber hinaus das Senden von Benutzerdaten oder Confirm-Daten - CFRM-Daten, *sendmsg()* - erforderlich (siehe auch [Bild 4 auf Seite 63](#)).

Mit `connect()` können nur Verbindungen zwischen zwei Sockets der Adressfamilie AF_ISO oder zwischen jeweils zwei Sockets der Adressfamilien AF_INET oder AF_INET6 oder zwischen jeweils einem Socket der Adressfamilien AF_INET oder AF_INET6 initiiert werden.

Richten Sie daher bei Nutzung aller Adressfamilien sowohl ein `listen()`-Socket der Adressfamilie AF_ISO, als auch ein `listen()`-Socket der Adressfamilien AF_INET oder AF_INET6 ein. Auf diese Weise ist der Verbindungsaufbau zu jeder unterstützten Adressfamilie gewährleistet.

Returnwert

0:

bei Erfolg

-1:

bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehleranzeige durch `errno`

EBADF

Der Parameter `s` ist kein gültiger Deskriptor.

EISCONN

Der Socket hat bereits eine Verbindung.

EOPNOTSUPP

Der Socket-Typ ist nicht SOCK_STREAM und wird von `listen()` nicht unterstützt.

Siehe auch

`accept()`, `connect()`, `socket()`

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

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

#include <netinet.in.h> /* AF_INET, AF_INET6 bei verbindungslosem Betrieb */

Kernighan-Ritchie-C:
int recv(s, buf, len, flags);

int s;
char *buf;
int len;
int flags;

int recvfrom(s, buf, len, flags, from, fromlen);

int s;
char *buf;
int len;
int flags;

struct sockaddr_in *from; /* AF_INET */
struct sockaddr_in6 *from; /* AF_INET6 */
int *fromlen;

ANSI-C:
int recv(int s, char* buf, int len, int flags);
int recvfrom(int s, char* buf, int len, int flags, struct sockaddr* from,
int* fromlen);
```

Beschreibung

Die Funktionen *recv()* und *recvfrom()* empfangen Nachrichten von einem Socket.

recv() kann Nachrichten nur von einem Socket empfangen, über den eine Verbindung aufgebaut ist (siehe Funktion *connect()* auf [Seite 130](#)).

recvfrom() kann Nachrichten von einem Socket mit oder ohne Verbindung empfangen. Der Funktionsaufruf von *recvfrom()* mit *from* ≠ Null-Zeiger und *fromlen* ≠ Null-Zeiger wird nur für Datagramme unterstützt.

Der Parameter *s* bezeichnet den Socket, von dem die Nachricht empfangen wird. *buf* spezifiziert den Speicherbereich, in den die Daten eingelesen werden sollen. *len* spezifiziert die Länge dieses Puffers.

Wenn der Parameter *from* nicht der Null-Zeiger ist (verbindungsloser Betrieb), wird in dem durch *from* referenzierten Speicherbereich die Absenderadresse der Nachricht abgelegt.

fromlen ist ein Ergebnisparameter. Vor dem Aufruf muss die Integer-Variable, auf die *fromlen* zeigt, die Größe des durch *from* referenzierten Puffers enthalten. Bei Rückkehr der Funktion enthält **fromlen* die aktuelle Länge der Adresse, die in **from* gespeichert ist. Die Funktion liefert die Länge der Nachricht zurück.

Der Parameter *flags* wird derzeit nur bei einem Datagramm-Socket mit dem Flag MSG_PEEK in den Adressfamilien AF_INET und AF_INET6 unterstützt. In den anderen Fällen sollte *flags* mit dem Wert 0 versorgt werden.

MSG_PEEK bewirkt, dass Daten gelesen werden können, ohne dass sie an der Quelle gelöscht werden. Darum ist ein Wiederholungslesen erforderlich.

Bei einem Datagramm-Socket (AF_INET, AF_INET6) muss die gesamte Nachricht in einer einzigen Operation gelesen werden. Wenn der angegebene Nachrichtenpuffer zu klein ist, werden die Daten, die nicht in den Puffer aufgenommen werden können, gelöscht.

Bei einem Stream-Socket (AF_INET, AF_INET6) werden Nachrichtengrenzen ignoriert. Sobald Daten verfügbar sind, werden sie an den Anrufer zurückgeliefert; es werden keine Daten gelöscht.

Bei einem Socket der Adressfamilie AF_ISO werden Nachrichtengrenzen berücksichtigt. 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 *soc_ioctl()* auf [Seite 212](#)). In diesem Fall wird -1 zurückgeliefert, wobei die Variable *errno* auf den Wert EWOULDBLOCK gesetzt wird.

Mit der Funktion *select()* stellen Sie fest, ob weitere Daten angekommen sind.

Returnwert

>0:

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

=0:

bei Erfolg.

Bei Sockets vom Typ SOCK_STREAM oder bei Sockets der Adressfamilie AF_ISO können keine Daten mehr empfangen werden. Der Partner hat seine Verbindung abgebaut.

Bei Sockets vom Typ SOCK_DGRAM wurde ein Datenpaket mit der Länge 0 empfangen, oder die Daten sind wegen Timeout vom Transportsystem gelöscht worden.

-1:

bei Fehler. *errno* wird gesetzt, um den Fehler anzuzeigen.

Fehleranzeige durch *errno*

EBADF

Der Parameter *s* ist kein gültiger Deskriptor.

ECONNRESET

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

EFAULT

Die Länge des Bereichs für die Aufnahme der Adresse ist zu klein.

EIO

Es sind Benutzerdaten verlorengegangen.

ENETDOWN

Die Verbindung zum Netzwerk ist nicht mehr aktiv.

ENOTCONN

Für den Socket besteht keine Verbindung.

EOPNOTSUPP

- Der Parameter *flags* enthält einen von 0 verschiedenen Wert.
- oder
- Der Socket-Typ ist nicht SOCK_STREAM, und *recv()* unterstützt nur Stream-Sockets.

EWOULDBLOCK

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

Siehe auch

connect(), getsockopt(), select(), send(), soc_ioctl(), soc_read(), soc_readv(), socket()

recvmsg() - Nachricht von einem Socket empfangen

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

```
Kernighan-Ritchie-C:
int recvmsg(s, msg, flags);
```

```
int s; flags
struct msghdr *msg;
```

```
ANSI-C:
int recvmsg(int s, struct msghdr* msg, int flags);
```

Beschreibung

Die Funktion *recvmsg()* wird in den Adressfamilien AF_INET, AF_INET6 und AF_ISO unterstützt und bietet je nach Parametrisierung (Parameter *msg*) folgende Funktionalität:

- Am Socket *s* können mit *recvmsg()* Benutzerdaten vom Partner-Socket empfangen werden.
- Nur in AF_ISO: Mit *recvmsg()* können Verbindungsdaten aus dem Socket *s* gelesen werden.

Als aktueller Parameter für *msg* ist ein Zeiger auf ein Objekt vom Datentyp *struct msghdr* anzugeben. Über die Komponente *msg->msg_control* (Datentyp *caddr_t* bzw. *char **) wird die gewünschte Funktionalität von *recvmsg()* ausgewählt:

- Wenn *msg->msg_control* der Null-Zeiger ist, werden Benutzerdaten empfangen.
- Nur in AF_ISO: Wenn *msg->msg_control* nicht der Null-Zeiger ist, wird *msg->msg_control* als ein Zeiger auf einen Speicherbereich mit der Struktur *cmsghdr* interpretiert, und es werden Verbindungsdaten aus dem Socket gelesen.

Aufgrund des internen Socket-Status von *s* wählt *recvmsg()* den Verbindungsdatentyp (CONN_DATA, CFRM_DATA oder DISC_DATA) aus und schreibt die zugehörigen Daten in *msg->msg_control->cmsg_data[]* (siehe Strukturen *msghdr* und *cmsghdr* auf der nächsten Seite).

Für die Handoff-Prozedur werden TPOPT_REDI_DATA und TPOPT_REDI_BDOK angeboten. Hier wird die Struktur *cmsg_redhdr* benötigt. Beschreibung siehe [Kapitel „Erweiterte Funktionen von SOCKETS\(BS2000\)“ auf Seite 73](#).

Struktur msghdr

Die Struktur *msghdr* ist in `<sys.socket.h>` wie folgt deklariert:

```
struct msghdr {
    caddr_t      msg_name;           /* optionale Adresse */
    int          msg_namelen;       /* Länge der Adresse */
    struct iovec *msg_iov;          /* scatter/gather-Felder */
    int          msg_iovlen;        /* Anzahl der Elemente in msg_iov */
    caddr_t      msg_control;       /* Hilfsdaten */
    int          msg_controllen;    /* Länge des Puffers für Hilfsdaten */
    int          msg_flags;         /* Flag für empfangene Nachricht */
};

struct msghdr *msg;
```

msg->msg_name und *msg->msg_namelen* werden nur in den Adressfamilien AF_INET und AF_INET6 mit dem Socket-Typ SOCK_DGRAM ausgewertet.

Ist *msg_name* nicht Null, dann wird der Inhalt als Zeiger auf einen Puffer interpretiert, in dem die Partner-Adressinformation eingetragen wird. *msg_namelen* gibt die Länge dieses Puffers an. Ist der Socket „non-connected“, so wird die Adressinformation des Absenders mit einer *sockaddr*-Struktur hinterlegt und *msg_namelen* enthält die Länge dieser Struktur.

Wenn diese Parameter nicht genutzt werden sollen, sollte *msg_name* den Wert des NULL-Zeigers und *msg_namelen* den Wert 0 haben.

msg->msg_iov ist ein Zeiger auf einen Speicherbereich mit Objekten vom Typ *struct iovec*.

msg->msg_iovlen gibt die Anzahl der Elemente (max. 16) dieses Speicherbereiches an.

msg->msg_control ist ein Zeiger auf ein Objekt vom Typ *struct cmsghdr*, in die die Funktion *recvmsg()* die erwarteten Verbindungsdaten einträgt.

msg->msg_controllen spezifiziert die Länge von **msg->msg_control*.

msg->msg_flags = MSG_EOR kennzeichnet das Satzende (nur AF_ISO).

Struktur iovec

Die Struktur *iovec* ist in `<sys.uio.h>` wie folgt deklariert:

```
struct iovec{
    caddr_t      iov_base; /* Puffer für Hilfsdaten */
    int          iov_len;  /* Pufferlänge */
};
```

Struktur cmsghdr

Die Struktur *cmsghdr* ist in `<sys.socket.h>` wie folgt deklariert:

```
struct cmsghdr {
    u_int    msg_len;           /* Anzahl der Datenbytes inkl. Header */
    int      msg_level;        /* erzeugendes Protokoll */
    int      msg_type;         /* protokollspezifischer Typ */
    u_char   msg_data[33]     /* Character-String für Verbindungsdaten */
};

struct cmsghdr *msg;
```

msg->msg_len enthält die Länge des Speicherbereiches von **msg->msg_control*.

In *msg->msg_level* wird `SOL_TRANSPORT` für den ISO-Transport-Service eingetragen (nur `AF_ISO`).

msg->msg_type kennzeichnet den Verbindungsdatentyp (`TPOPT_CONN_DATA`, `TPOPT_CFRM_DATA`, `TPOPT_DISC_DATA`).

In *msg->msg_data* werden die Verbindungsdaten bis zur maximalen Länge von 32 byte sowie das abschließende NULL-Byte eingetragen.

Der Parameter *flags* wird derzeit nur bei einem Datagramm-Socket mit dem Flag `MSG_PEEK` in den Adressfamilien `AF_INET` und `AF_INET6` unterstützt. In den anderen Fällen sollte *flags* mit dem Wert 0 versorgt werden. `MSG_PEEK` bewirkt, dass die Daten gelesen werden können, ohne dass sie an der Quelle gelöscht werden. Darum ist ein Wiederholungslesen erforderlich.

Returnwert

>0:

bei Erfolg:
Anzahl Bytes der empfangenen Benutzerdaten

=0:

- nur `AF_ISO`
bei Verbindungsdaten (`CONN_DATA`, `CFRM_DATA`, `DISC_DATA`)
- `AF_INET`, `AF_INET6`
Es können keine Daten mehr empfangen werden. Der Partner hat seine Verbindung geordnet abgebaut (nur bei Sockets vom Typ `SOCK_STREAM`).

Bei Sockets vom Typ `SOCK_DGRAM` wurde ein Datenpaket mit der Länge 0 empfangen, oder die Daten sind wegen Timeout vom Transportsystem gelöscht worden.

-1:

bei Fehler. *errno* wird gesetzt, um den Fehler anzuzeigen.

Fehleranzeige durch errno**EBADF**

Der Parameter *s* ist kein gültiger Deskriptor.

ECONNRESET

Die Verbindung zum Partner wurde abgebrochen.

EINVAL

Ein Parameter spezifiziert einen ungültigen Wert.

ENETDOWN

Die Verbindung zum Netzwerk ist nicht mehr aktiv.

ENOTCONN

Für den Socket besteht keine Verbindung.

EOPNOTSUPP

Der Funktionsaufruf enthält unzulässige Attribute.

EPIPE

Der Partner hat die Verbindung abgebrochen.

EWOULDBLOCK

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

select() - Ein-/Ausgabe multiplexen

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

```
Kernighan-Ritchie-C:
int select(nfds, readfds, writefds, exceptfds, timeout);

int nfds;
fd_set *readfds, *writefds, *exceptfds;
struct timeval *timeout;
```

```
FD_SET(fd, &fdset);
FD_CLR(fd, &fdset);
FD_ISSET(fd, &fdset);

int fd;
fd_set fdset;
```

```
ANSI-C:
int select(int nfds, fd_set* readfds, fd_set* writefds, fd_set* exceptfds,
struct timeval* timeout);
```

```
FD_SET(fd, &fdset);
FD_CLR(fd, &fdset);
FD_ISSET(fd, &fdset);

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.

Der Parameter *exceptfds* wird von SOCKETS(BS2000) derzeit nicht ausgewertet.

Die Bitmasken zu den einzelnen Deskriptormengen werden als Bitfelder in Integer-Reihungen abgespeichert. Die maximale Größe der Bitfelder ermitteln Sie mit der Funktion `getdtablesize()` (siehe [Seite 142](#)). Der benötigte Speicher sollte dynamisch vom System angefordert werden.

Der Parameter `nfds` gibt an, wie viele Bits in jeder Bitmaske zu prüfen sind. `select()` prüft in den einzelnen Bitmasken die Bits 0 bis `nfds-1`. `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 kann der Benutzer Bitmasken bzw. Deskriptormengen manipulieren:

`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 $\neq 0$: `fd` ist Element von `fdset`.
- Rückgabewert $= 0$: `fd` ist nicht Element von `fdset`.

Das Verhalten dieser Makros ist undefiniert, wenn der Deskriptor-Wert kleiner als 0 oder größer als die mit `getdtablesize()` ermittelte maximale Größe für die Bitfelder 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`, `writelfds` 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. Die Deskriptormengen sind dann undefiniert.

-1:

bei Fehler. *errno* wird gesetzt, um den Fehler anzuzeigen. Die Deskriptormengen sind dann undefiniert.

7F000000:

Trace-Event für User-Sockets-Trace

Fehleranzeige durch *errno*

EBADF

Eine der Deskriptormengen spezifiziert einen ungültigen Deskriptor.

EINTR

Der *select()*-Aufruf wurde von *soc_wake()* unterbrochen.

ENETDOWN

Die Verbindung zum Netzwerk ist nicht mehr aktiv.



Wenn virtuelle Hosts genutzt werden, hat ENETDOWN nicht unbedingt die Bedeutung, dass das gesamte Netzwerk ausgefallen ist. Es kann auch bedeuten, dass nur das Netzwerk eines virtuellen Hosts ausgefallen ist.

Hinweis

Unter seltenen Umständen 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 die Anwendung kritisch ist, dass Schreiboperationen auf einen Datei-Deskriptor nicht blockieren, sollte der Benutzer den Deskriptor mit einem *soc_ioctl()*-Aufruf auf nicht-blockierende Ein-/Ausgabe setzen.

Siehe auch

accept(), *connect()*, *listen()*, *recv()*, *send()*, *soc_ioctl()*, *soc_write()*, *soc_writenv()*

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

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

#include <netinet.in.h> /* AF_INET, AF_INET6 bei verbindungslosem Betrieb */

Kernighan-Ritchie-C:
int send(s, msg, len, flags);

int s;
char *msg;
int len, flags;

int sendto(s, msg, len, flags, to, tolen);

int s;
char *msg;
int len, flags;

struct sockaddr_in *to; /* AF_INET */
struct sockaddr_in6 *to; /* AF_INET6 */
int tolen;

ANSI-C:
int send(int s, char* msg, int len, int flags);
int sendto(int s, char* msg, int len, int flags, struct sockaddr* to,
           int tolen);
```

Beschreibung

Die Funktionen *send()*, *sendto()* senden Nachrichten von einem Socket an einen anderen Socket. *send()* kann nur bei einem Socket benutzt werden, über den eine Verbindung aufgebaut ist (siehe Funktion *connect()* auf [Seite 130](#)).

sendto() kann auch im verbindungslosen Betrieb verwendet werden. Der Funktionsaufruf von *sendto()* mit *to* \neq Null-Zeiger und *toLen* \neq 0 wird nur für Datagramme unterstützt.

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 vollständig transportiert zu werden, wird bei Datagramm-Sockets (d.h. nur AF_INET und AF_INET6) der Fehler EMSGSIZE geliefert und die Nachricht wird nicht gesendet.

Der Parameter *flags* wird derzeit nicht unterstützt und sollte mit dem Wert 0 versorgt werden. Ein Wert ungleich 0 führt zu einem Fehler, wobei die Variable *errno* auf den Wert EOPNOTSUPP gesetzt wird.

Wenn die Nachricht nicht sofort gesendet werden kann, blockiert *send()*, sofern der Socket nicht in den nicht-blockierenden Ein-/Ausgabemodus gesetzt wurde. Mit der Funktion *select()* können Sie feststellen, wann das Senden weiterer Daten möglich ist.

Returnwert

≥ 0 :

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

-1:

bei Fehler. *errno* wird gesetzt, um den Fehler anzuzeigen.

Fehleranzeige durch errno

EBADF

Der Parameter *s* ist kein gültiger Deskriptor.

EFAULT

Die Länge des Bereichs für die Aufnahme der Adresse ist zu klein, oder die Länge des Bereichs für die Nachricht ist zu klein.

EIO

E/A-Fehler. Die Meldung konnte nicht an das Transportsystem übergeben werden.

EMSGSIZE

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

ENETDOWN

Die Verbindung zum Netzwerk ist nicht mehr aktiv.

ENOTCONN

Für den Socket besteht keine Verbindung. Ein Lese-/Schreibversuch wurde zurückgewiesen.

EOPNOTSUPP

– Der Parameter *flags* wurde mit einem Wert ungleich 0 spezifiziert. Dies wird jedoch nicht unterstützt.

oder

– Der Socket hat nicht den Typ `SOCK_STREAM`, die Operation wird aber nur für Stream-Sockets unterstützt.

EWOULDBLOCK

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

Siehe auch

`connect()`, `getsockopt()`, `recv()`, `select()`, `soc_ioctl()`, `soc_write()`, `socket()`

sendmsg() - Nachricht von Socket zu Socket senden

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

```
Kernighan-Ritchie-C:
int sendmsg(s, msg, flags);

int s, flags;
struct msghdr *msg;
```

```
ANSI-C:
int sendmsg(int s, struct msghdr* msg, int flags);
```

Beschreibung

Die Funktion *sendmsg()* wird in den Adressfamilien AF_INET, AF_INET6 und AF_ISO unterstützt und bietet je nach Parametrisierung (Parameter *msg*) folgende Funktionalität:

- Mit *sendmsg()* können Benutzerdaten von einem Socket an einen Partner-Socket gesendet werden.
- Nur in AF_ISO: Mit *sendmsg()* können Verbindungsdaten in einen Socket *s* geschrieben werden.

Als aktueller Parameter für *msg* ist ein Zeiger auf ein Objekt vom Datentyp *struct msghdr* anzugeben. Über die Komponente *msg->msg_control* (Datentyp *caddr_t* bzw. *char **) wird die gewünschte Funktionalität von *sendmsg()* ausgewählt:

- Wenn *msg->msg_control* der Null-Zeiger ist, werden Benutzerdaten gesendet.
- Nur in AF_ISO: Wenn *msg->msg_control* nicht der Null-Zeiger ist, wird *msg->msg_control* als ein Zeiger auf einen Speicherbereich mit der Struktur *cmsghdr* interpretiert, und es werden Verbindungsdaten in den Socket geschrieben.

Auf diese Weise ermöglicht es *sendmsg()* in AF_ISO, die Quittung für eine Verbindungsanforderung an den Kommunikationspartner zu senden, ohne dass Benutzerdaten oder Verbindungsdaten übertragen werden.

Struktur msghdr

Die Struktur *msghdr* ist in `<sys.socket.h>` wie folgt deklariert:

```
struct msghdr {
    caddr_t    msg_name;           /* optionale Adresse */
    int        msg_namelen;       /* Länge der Adresse */
    struct iovec *msg_iov;        /* scatter/gather-Felder */
    int        msg_iovlen;        /* Anzahl der Elemente in msg_iov */
    caddr_t    msg_control;       /* Hilfsdaten */
    int        msg_controllen;    /* Länge des Puffers für Hilfsdaten */
    int        msg_flags;         /* Flag für empfangene Nachricht */
};

struct msghdr *msg;
```

msg->msg_name und *msg->msg_namelen* werden nur in den Adressfamilien AF_INET und AF_INET6 mit dem Socket-Typ SOCK_DGRAM ausgewertet. *msg_name* gibt die Adresse einer Socket-Adress-Struktur und *msg_namelen* die Länge dieser Adress-Struktur an. Wenn diese Parameter nicht genutzt werden sollen, sollte *msg_name* den Wert des NULL-Zeigers und *msg_namelen* den Wert 0 haben.

msg->msg_iov ist ein Zeiger auf einen Speicherbereich mit Objekten vom Typ *struct iovec*. *msg->msg_iovlen* gibt die Anzahl der Elemente (max. 16) dieses Speicherbereiches an. *msg->msg_control* ist ein Zeiger auf ein Objekt vom Typ *struct cmsghdr*, das vor Aufruf von *sendmsg()* mit den zu schreibenden Verbindungsdaten versorgt werden muss (nur AF_ISO). *msg->msg_controllen* spezifiziert die Länge von **msg->msg_control*. In *msg->msg_flags* kennzeichnet *sendmsg()* das Satzende mit MSG_EOR (nur AF_ISO).

Struktur iovec

Die Struktur *iovec* ist in `<sys.uio.h>` wie folgt deklariert:

```
struct iovec{
    caddr_t    iov_base; /* Puffer für Hilfsdaten */
    int        iovlen;   /* Pufferlänge */
};
```

Struktur cmsghdr

Die Struktur *cmsghdr* ist in `<sys.socket.h>` wie folgt deklariert:

```
struct cmsghdr {
    u_int    msg_len;           /* Anzahl der Datenbytes inkl. Header */
    int      msg_level;        /* erzeugendes Protokoll */
    int      msg_type;         /* protokollspezifischer Typ */
    u_char   msg_data[33]     /* Character-String für Verbindungsdaten */
};

struct cmsghdr *msg;
```

msg->msg_len enthält die Länge des Speicherbereiches von **msg->msg_control*.
 In *msg->msg_level* wird SOL_TRANSPORT für den ISO-Transport-Service eingetragen.
msg->msg_type kennzeichnet den Verbindungsdatentyp (TPOPT_CONN_DATA, TPOPT_CFRM_DATA, TPOPT_DISC_DATA).
 In *msg->msg_data* werden die Verbindungsdaten bis zur maximalen Länge von 32 byte sowie das abschließende NULL-Byte eingetragen.

Für die Handoff-Prozedur werden TPOPT_RED1_DATA und TPOPT_RED1_BDOK angeboten. Hier wird die Struktur *msg_redhdr* benötigt. Beschreibung siehe [Kapitel „Erweiterte Funktionen von SOCKETS\(BS2000\)“ auf Seite 73](#).

Returnwert

≥0:

Anzahl Bytes der gesendeten Benutzerdaten

AF_ISO

0 bei Verbindungsdaten (CONN_DATA, CFRM_DATA, DISC_DATA)

-1:

bei Fehler. *errno* wird gesetzt, um den Fehler anzuzeigen.

Fehleranzeige durch errno

EBADF

Der Parameter *s* ist kein gültiger Deskriptor.

ECONNRESET

Die Verbindung zum Partner wurde abgebrochen.

EINVAL

Ein Parameter spezifiziert einen ungültigen Wert.

EIO

E/A-Fehler. Die Nachricht konnte nicht an das Transportsystem übergeben werden.

ENETDOWN

Die Verbindung zum Netzwerk ist nicht mehr aktiv.

ENOTCONN

Für den Socket besteht keine Verbindung.

EOPNOTSUPP

Der Funktionsaufruf enthält falsche Attribute.

EPIPE

Der Partner hat die Verbindung abgebrochen.

EWOULDBLOCK

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

shutdown() - Voll-Duplex-Verbindung abbauen

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

```
Kernighan-Ritchie-C:  
int shutdown(s, how);
```

```
int s, how;
```

```
ANSI-C:  
int shutdown(int s, int how);
```

Beschreibung

Die Funktion *shutdown()* schränkt die Funktionalität des Socket ein und baut die Verbindung nach außen ganz oder teilweise ab. Der Socket bleibt aber bestehen. Mit der Funktion *soc_close()* können Sie den Socket schließen.

Die Funktion *shutdown()* wird in den Adressfamilien AF_INET und AF_INET6 unterstützt.

Mit dem Parameter *how* wird gesteuert, wie die zum Socket *s* gehörende Verbindung abgebaut werden soll. Folgende Werte von *how* sind möglich:

SHUT_RD:

Der Socket wird für Lesen gesperrt, d.h. eine Lesefunktion wird nicht mehr ausgeführt. Diese Funktionalität ist in der Anwendung problematisch, weil der Partner-Socket nicht über diese Einschränkung informiert wird.



Wenn der Partner-Socket weiterhin Daten sendet, kann es zu einer Stausituation kommen: Die gesendeten Daten belegen Speicherplatz im Transportsystem und diese Ressourcen können nicht freigegeben werden, weil die Daten vom Empfänger nicht abgeholt werden. Wenn die Speicherplatz-Ressourcen erschöpft sind, ist auch das Senden nicht mehr möglich.

SHUT_WR:

Der Socket wird für Schreiben gesperrt. Dem Partner-Socket wird mitgeteilt, dass jetzt keine Daten mehr von diesem Socket gesendet werden. Dies entspricht einem „graceful disconnect“.

SHUT_RDWR:

Der Socket wird für Lesen und Schreiben gesperrt. Dem Partner-Socket wird mitgeteilt, dass weder Daten gesendet noch gelesen werden. Dies entspricht einem „abortive disconnect“.

Returnwert

0:

bei Erfolg

-1:

bei Fehler. *errno* wird gesetzt, um den Fehler anzuzeigen.**Fehleranzeige durch errno**

EBADF

Der Parameter *s* ist kein gültiger Deskriptor.

ENOTCONN

Für den Socket besteht keine Verbindung.

Hinweis

Die Funktion *shutdown()* wurde bis zu SOCKETS(BS2000) < V.2.1 ohne Funktionalität unterstützt, d.h. der Aufruf wurde nicht abgelehnt, bewirkte aber keine Aktion.

Die oben beschriebene Funktionalität wird bereitgestellt, wenn das Anwenderprogramm mit der Anwenderbibliothek von SOCKETS(BS2000) ab Version 2.1 kompiliert wurde.

Siehe auch

soc_close()

soc_close() (close) - Socket schließen

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

```
Kernighan-Ritchie-C:  
int soc_close(s);
```

```
int s;
```

```
ANSI-C:  
int soc_close(int s);
```

Beschreibung

Die Funktionalität von *soc_close()* wird im Detail durch die verwendete Adressfamilie festgelegt.

soc_close() bei *AF_INET* und *AF_INET6*

soc_close() schließt den Socket *s* in Abhängigkeit von der Option *SO_LINGER* (siehe Funktion *setsockopt()* auf [Seite 161](#)).

Wenn *soc_close()* mit der Option *SO_LINGER* verwendet wird, versucht *soc_close()*, die Verbindung nach dem Senden der anstehenden Daten innerhalb der bei *SO_LINGER* spezifizierten Zeit zu beenden.

soc-close() bei *AF_ISO*

soc_close() schließt den Socket *s*. Im Netz und in BCAM vorhandene Daten gehen verloren.

Returnwert

0:

bei Erfolg.

-1:

bei Fehler. *errno* wird gesetzt, um den Fehler anzuzeigen.

Fehleranzeige durch errno

EBADF

Der Parameter *s* ist kein gültiger Deskriptor.**Siehe auch**

setsockopt()

soc_eof(), soc_error(), soc_clearerr() (eof, error, clearerr) - Status-Information abfragen

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

Kernighan-Ritchie-C:

```
int soc_eof(s);  
int s;
```

```
int soc_error(s);  
int s;
```

```
int soc_clearerr(s);  
int s;
```

ANSI-C:

```
int soc_eof(int s);  
int soc_error(int s);  
int soc_clearerr(int s);
```

Beschreibung und Returnwert

Die Funktion *soc_eof()* liefert einen Wert $\neq 0$ zurück, falls für den Socket *s* die EOF-Bedingung zutrifft, andernfalls liefert *soc_eof()* den Wert 0 zurück.

Die Funktion *soc_error()* liefert einen Wert $\neq 0$ zurück, falls auf dem Socket *s* ein Lese- oder Schreibfehler aufgetreten ist. Andernfalls liefert *soc_error()* den Wert 0 zurück. Die Fehleranzeige wird solange aufbewahrt, bis sie mit der Funktion *soc_clearerr()* gelöscht wird.

Die Funktion *soc_clearerr()* löscht die Fehleranzeige für den Socket *s*.

soc_flush() (flush) - Daten aus Ausgabepuffer übertragen

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

Kernighan-Ritchie-C:
int soc_flush(s);
int s;

ANSI-C:
int soc_flush(int s);
```

Beschreibung

Die Funktion *soc_flush()* wird nur in den Adressfamilien AF_INET und AF_INET6 unterstützt.

soc_flush() überträgt alle zum Socket *s* gehörenden Daten aus dem Ausgabepuffer in das Transportsystem.

Returnwert

0:

falls der Puffer geleert werden konnte oder der Puffer leer war.

EOF:

falls der Socket-Deskriptor ungültig ist oder der Datentransfer zum Transportsystem unmöglich war.

soc_getc() (getc) - Zeichen aus dem Eingabepuffer lesen

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

```
Kernighan-Ritchie-C:  
int soc_getc(s);  
int s;
```

```
ANSI-C:  
int soc_getc(int s);
```

Beschreibung

Die Funktion *soc_getc()* wird nur in den Adressfamilien AF_INET und AF_INET6 unterstützt und kann nur auf Stream-Sockets angewendet werden.

Die Funktion *soc_getc()* liest das nächste Zeichen aus dem Eingabepuffer des Sockets *s* und liefert das Zeichen als Ergebnis.

Returnwert

Integer-Wert des gelesenen Zeichens:
bei Erfolg

EOF:
falls wegen Dateiende (EOF) kein Zeichen gelesen werden konnte.

Fehleranzeige durch errno

EWOULDBLOCK

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

soc_gets() (gets) - Character-String aus dem Eingabepuffer lesen

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

Kernighan-Ritchie-C:
char *soc_gets(s, n, d);

char *s;
int n, d;

ANSI-C:
char* soc_gets(char* s, int n, int d);
```

Beschreibung

Die Funktion *soc_gets()* wird nur in den Adressfamilien AF_INET und AF_INET6 unterstützt und kann nur auf Stream-Sockets angewendet werden.

Die Funktion *soc_gets()* liest einen Character-String, bestehend aus den ersten maximal $n-1$ Zeichen des Eingabepuffers von Socket d , in den Puffer s . Gelesen wird maximal bis zum ersten Zeilenwechsel (dargestellt durch die Sequenz 0x15 in EBCDIC) oder bis zum Ende des Eingabepuffers von Socket d oder dem Erreichen von $n-1$ Zeichen. Der im Puffer s zurückgelieferte Character-String wird durch das Null-Byte terminiert.

Returnwert

Zeiger auf den Ergebnisstring:
bei Erfolg.

Null-Zeiger:
Bei Lesefehler.

Fehleranzeige durch errno

EWOULDBLOCK

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

soc_ioctl() (ioctl) - Sockets steuern

```
#include <sys.socket.h>
#include <ioctl.h>
#include <net.if.h>
```

Kernighan-Ritchie-C:
int soc_ioctl(s, request, argp);

```
int s;
unsigned long request;
char *argp;
```

ANSI-C:
int soc_ioctl(int s, unsigned long request, char* argp);

Beschreibung

Die Funktion *soc_ioctl()* führt Steuerfunktionen für Sockets aus.
s bezeichnet den Socket-Deskriptor.

Folgende Steuerfunktionen werden für Sockets in der Adressfamilie AF_INET unterstützt:

| request | *argp | Funktion |
|-----------------|----------------|---|
| FIONBIO | int | Blocking-Modus ein- und ausschalten |
| FIONREAD | int | Nachrichtenlänge im Puffer ermitteln |
| SIOCGIFCONF | struct ifconf | Interface-Konfiguration ermitteln |
| SIOCGIFADDR | struct ifreq | Internet-Adresse des Interface ermitteln |
| SIOCGIFBRDADDR | struct ifreq | Broadcast-Adresse des Interface ermitteln |
| SIOCGIFFLAGS | struct ifreq | Flags des Interface ermitteln |
| SIOCGIFNETMASK | struct ifreq | Netzmaske zum Interface ermitteln |
| SIOCGLIFADDR | struct lifreq | Interface-Adresse ermitteln |
| SIOCGLIFBRDADDR | struct lifreq | Broadcast-Adresse des Interface ermitteln |
| SIOCGLIFCONF | struct lifconf | Interface-Konfigurations-Liste ermitteln |
| SIOCGLIFFLAGS | struct lifreq | Interface-Flags ermitteln |
| SIOCGLIFHWADDR | struct lifreq | MAC-Adresse zum Interface ermitteln |
| SIOCGLIFINDEX | struct lifreq | Interface-Index ermitteln |
| SIOCGLIFNETMASK | struct lifreq | Netzmaske zum Interface ermitteln |

| | | |
|--------------|---------------|---|
| SIOCGLIFNUM | struct lifnum | Anzahl Interfaces ermitteln |
| SIOCGLAHCONF | struct lvhost | Liste aller aktiven Hosts ausgeben |
| SIOCGLVHCONF | struct lvhost | Liste der aktiven virtuellen Hosts ausgeben |
| SIOCGLVHNUM | int | Anzahl der aktiven virtuellen Hosts ermitteln |

Folgende Steuerfunktionen werden für Sockets in der Adressfamilie AF_INET6 unterstützt:

| request | *argp | Funktion |
|-----------------|----------------|---|
| FIONBIO | int | Blocking-Modus ein- und ausschalten |
| FIONREAD | int | Nachrichtenlänge im Puffer ermitteln |
| SIOCGLIFADDR | struct lifreq | Interface-Adresse ermitteln |
| SIOCGLIFBRDADDR | struct lifreq | Broadcast-Adresse des Interface ermitteln |
| SIOCGLIFCONF | struct lifconf | Interface-Konfigurations-Liste ausgeben |
| SIOCGLIFFLAGS | struct lifreq | Interface-Flags ermitteln |
| SIOCGLIFHWADDR | struct lifreq | MAC-Adresse zum Interface ausgeben |
| SIOCGLIFINDEX | struct lifreq | Interface-Index ermitteln |
| SIOCGLIFNETMASK | struct lifreq | Netzmaske zum Interface ermitteln |
| SIOCGLIFNUM | struct lifnum | Anzahl Interfaces ermitteln |
| SIOCGLAHCONF | struct lvhost | Liste aller aktiven Hosts ausgeben |
| SIOCGLVHCONF | struct lvhost | Liste der aktiven virtuellen Hosts ausgeben |
| SIOCGLVHNUM | int | Anzahl der aktiven virtuellen Hosts ermitteln |

Folgende Steuerfunktion wird für Sockets in der Adressfamilie AF_ISO unterstützt:

| request | *argp | Funktion |
|---------|-------|-------------------------------------|
| FIONBIO | int | Blocking-Modus ein- und ausschalten |

FIONBIO

Diese Option beeinflusst das Ausführungsverhalten von Socket-Funktionen auf dem Socket *s* bei Datenflusskontrolle sowie bei noch nicht abgeschlossenen Aktionen des Kommunikationspartners:

- **argp* = 0:
Socket-Funktionen blockieren, bis die Funktion ausgeführt werden kann.

- **argp* ≠ 0:
Socket-Funktionen kehren mit dem *errno*-Code EWOULDBLOCK zurück, wenn die Funktion nicht sofort ausgeführt werden kann. Mit *select()* oder *soc_poll()* kann festgestellt werden, welche Sockets zum Lesen oder Schreiben bereit sind.
Standardfall: FIONBO ist nicht gesetzt.

FIONREAD

Die Länge der aktuell im Eingangspuffer vorhandenen Nachricht (in Bytes) wird zurückgeliefert.

SIOCGIFCONF

Es wird eine Ausgabeliste in Form von unverketteten Elementen vom Typ *struct ifreq* (siehe [Seite 214](#) Option SIOCGIFADDR) erstellt. Den entsprechenden Speicherbereich für diese Liste stellt der Aufrufer bereit, indem er die Anfangsadresse und die Länge in die entsprechenden Felder der Struktur *ifconf* einträgt.

Es werden nur so viele Elemente vom Typ *struct ifreq* ausgegeben, wie in den zur Verfügung gestellten Puffer hineinpassen.

Diese Interfaces gehören zu einem Host. Im Normalfall ist das der Standardhost. Sind auch virtuelle Hosts konfiguriert, gibt es folgende Möglichkeiten, die entsprechenden Interfaces zu erhalten:

- Ist die Anwendung unter einer Kennung gestartet, die durch einen Eintrag in der BCAM-Application-Tabelle auf einen virtuellen Host verlagert wird, dann wird die Information für diesen virtuellen Host ausgegeben.
- Mit der *setsockopt()*-Subfunktion SO_VHOSTANY ist es möglich, vor dem Aufruf von *soc_ioctl()* auszuwählen, für welchen Host die Information ausgegeben werden soll. Die Struktur *ifconf* ist in `<net.if.h>` wie folgt deklariert:

```
struct ifconf {
    int          ifc_len;
    union {
        caddr_t ifcu_buf;
        struct ifreq *ifcu_req;
    } ifc_ifcu;
#define ifc_buf ifc_ifcu.ifcu_buf
#define ifc_req ifc_ifcu.ifcu_req
};
```

SIOCGIFADDR

Für das in der Struktur *ifreq* mit dem Interfacenamen *ifr_name* spezifizierte Interface wird die Internet-Adresse zurückgeliefert.

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

```
struct ifreq {
#define IFNAMSIZ 16
    char ifr_name[IFNAMSIZ]; /* Interfacename z.B. IF000003“ */
    union {
```

```

    struct sockaddr ifru_addr;
    struct sockaddr ifru_dstaddr;
    struct sockaddr ifru_broadaddr;
    short ifru_flags;
    int ifru_metric;
    caddr_t ifru_data;
} ifr_ifru;
#define ifr_addr      ifr_ifru.ifru_addr      /* Adresse */
#define ifr_dstaddr  ifr_ifru.ifru_dstaddr   /* Zieladr. d. Verbindung */
#define ifr_broadaddr ifr_ifru.ifru_broadaddr /* Broadcast-Adresse */
#define ifr_flags    ifr_ifru.ifru_flags    /* Flags */
#define ifr_metric   ifr_ifru.ifru_metric   /* Metrik */
#define ifr_data     ifr_ifru.ifru_data     /* für die Schnittstelle */
};

```

Zu beachten ist, dass SOCKETS(BS2000) derzeit Informationen für nur *ein* Interface liefert.

SIOCGIFBRDADDR

Für das in der Struktur *ifreq* (siehe [Seite 214](#), Option SIOCGIFADDR) spezifizierte Interface wird die Broadcast-Adresse zurückgeliefert, wenn es sich um ein IPv4-Interface handelt und wenn das Flag IFF_BROADCAST gesetzt ist. Das ist normalerweise nicht der Fall, weil mit Sockets-Sprachmitteln und der Transportsystemunterstützung kein Broadcast erzeugt werden kann.

SIOCGIFFLAGS

Für den in der *struct ifreq* spezifizierten Interface-Namen werden die Interface-Flags im Element *ifr_flags* zurückgeliefert:

- IFF_UP, wenn das Interface aktiv ist
- IFF_BROADCAST, wenn über dieses Interface Broadcast-Nachrichten gesendet werden können
- IFF_MULTICAST, wenn über dieses Interface Multicast-Nachrichten versendet werden können
- IFF_LOOPBACK, wenn über dieses Interface Nachrichten an Loopback versendet werden können
- IFF_CONTROLLAN, wenn über dieses Interface mit dem CONTROLLAN kommuniziert werden kann

SIOCGIFNETMASK

Für das in der Struktur *ifreq* spezifizierte IPv4-Interface wird im Element *ifr_addr* die Subnetmaske in der Form von auf „1“ gesetzten Bits des Netzanteils der Subnetmaske ausgegeben (z.B. „FFFFFF00“).

SIOCGLIFADDR

Für den in der Struktur *lifreq* spezifizierten Namen wird die Interface-Adresse zurückgegeben. Die Struktur *lifreq* ist eine besonders im Hinblick auf IPv6 erweiterte Struktur *ifreq*.

SIOCGLIFBRDADDR

Für das in der Struktur *lifreq* spezifizierte Interface wird die Broadcast-Adresse zurückgeliefert, wenn für dieses Interface das Flag IFF_BROADCAST gesetzt ist. Das ist normalerweise nicht der Fall, weil mit Sockets-Sprachmitteln und der Transportsystemunterstützung kein Broadcast erzeugt werden kann.

Für die Nutzung von MULTICAST gibt es eigene Subfunktionen unter *getsockopt()* / *setsockopt()* (siehe [Seite 161](#)).

SIOCGLIFCONF

Es wird eine Ausgabeliste in Form von unverketteten Elementen vom Typ *struct lifreq* erstellt. Den entsprechenden Speicherbereich für diese Liste stellt der Aufrufer bereit, indem er die Anfangsadresse und die Länge in die entsprechenden Felder der Struktur *lifconf* einträgt.

Es werden nur so viele Elemente vom Typ *struct lifreq* ausgegeben, wie in den zur Verfügung gestellten Puffer hineinpassen. Durch die Belegung der Elemente *lifc_family* und *lifc_flags* kann die Ausgabe gefiltert werden.

Werte für *lifc_family*: AF_INET, AF_INET6, AF_UNSPEC

Werte für *lifc_flags*: siehe SIOCGLIFFLAGS

Diese Interfaces gehören zu einem Host. Im Normalfall ist dies der Standardhost. Sind auch virtuelle Hosts konfiguriert, gibt es folgende Möglichkeiten, die entsprechenden Interfaces zu erhalten:

- Ist die Anwendung unter einer Kennung gestartet, die durch einen Eintrag in der BCAM-Application-Tabelle auf einen virtuellen Host verlagert wird, dann wird die Information für diesen virtuellen Host ausgegeben.
- Mit der *setsockopt()*-Subfunktion SO_VHOSTANY ist es möglich, vor dem Aufruf von *soc_ioctl()* auszuwählen, für welchen Host die Information ausgegeben werden soll. Die Struktur *lifconf* ist in `<net.if.h>` wie folgt deklariert:

```
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
};
```

SIOCGLIFFLAGS

Für den in der Struktur *lifreq* spezifizierten Interface-Namen werden die Interface-Flags im Element *lifc_flags* zurückgeliefert:

- IFF_UP, wenn das Interface aktiv ist

- IFF_BROADCAST, wenn über dieses Interface Broadcast-Nachrichten gesendet werden können
- IFF_MULTICAST, wenn über dieses Interface Multicast-Nachrichten versendet werden können
- IFF_LOOPBACK, wenn über dieses Interface Nachrichten an Loopback versendet werden können
- IFF_CONTROLLAN, wenn über dieses Interface mit dem CONTROLLAN kommuniziert werden kann
- IFF_AUTOCONFIG, wenn dieses Interface mit einer durch IPv6-Autoconfig generierten Adresse versorgt wurde. Dazu gehört auch die lokal im Rechner erzeugt IPv6-Link-Local-Adresse mit dem Prefix FE80::/10.

SIOCGLIFHWADDR

Für den in der Struktur *lifreq* spezifizierten Interface-Namen wird die MAC-Adresse ausgegeben.

SIOCGLIFINDEX

Für den in der Struktur *lifreq* spezifizierten Interface-Namen wird der Index ausgegeben.

SIOCGLIFNETMASK

Für den in der Struktur *lifreq* angegebenen Interfacenamen *lifr_name* wird im Element *lifr_addr* die Subnetmaske und im Element *lifr_addrln* die Prefixlänge in Bits ausgegeben. Handelt es sich um ein IPv4-Interface erfolgt die Ausgabe in der Form, dass alle betroffenen Bits des Netzanteils auf „1“ gesetzt sind (z.B. „FFFFFF00“). Handelt es sich um ein IPv6-Interface, wird der Netzanteil als Originaladresse ausgegeben und die folgenden Bits sind dann auf „0“ gesetzt (z.B. „FD11F052433485AA000000000000“).

SIOCGLIFNUM

Für die in der Struktur *lifnum* spezifizierte Adressfamilie wird die Anzahl der Interfaces ausgegeben.

SIOCGLAHCONF

Es wird eine nicht verkettete Liste mit Listenelementen vom Typ *struct lvhost* zurückgegeben. Diese enthalten den Socket-Hostnamen, den BCAM-Hostnamen, die Host-Nummer und ein Active-Flag vom realen Host und, falls vorhanden, auch von virtuellen Hosts. Der Speicher für diese Liste muss vom Anwender in **argp* mit dem Typ *struct lvhost* übergeben werden. Die Länge ist im Feld *lvhostlen* einzutragen.

Wenn Informationen zu allen Hosts ausgegeben werden sollen, ist ein Speicher von $n \times \text{sizeof}(\text{struct lvhost})$ erforderlich, wobei n der maximal möglichen Anzahl von aktiven Hosts entspricht.

Auf die Rückgabeinformation kann durch direkte Adressierung oder durch Indizierung zugegriffen werden. Im ersten Listen-Element wird im Feld *vhostsum* die Anzahl der zurückgegebenen Listen-Elemente vom Typ *struct lvhost* eingetragen. Im letzten Listen-Element wird das Feld *vhostlast* mit „1“ gekennzeichnet.

Ist der zur Verfügung gestellte Speicher nicht groß genug, wird im letzten Listen-Element das Feld *vhostlast* mit „-1“ gekennzeichnet.

SIOCGLVHCONF

Es wird eine nicht verkettete Liste mit Listenelementen vom Typ *struct lvhost* zurückgegeben, die im Unterschied zu SIOCGLAHCONF nur die Informationen über die virtuellen Hosts enthält.

Der Speicher für diese Liste muss vom Anwender in **argp* mit dem Typ *struct lvhost* übergeben werden. Die Länge ist im Feld *lvhostlen* einzutragen. Wenn Informationen zu allen virtuellen Hosts ausgegeben werden sollen, ist ein Speicher von $n \times \text{sizeof}(\text{struct lvhost})$ erforderlich, wobei n dem Rückgabewert von SIOCGLVHNUM entspricht.

Auf die Rückgabeinformation kann durch direkte Adressierung oder durch Indizierung zugegriffen werden. Im ersten Listen-Element wird im Feld *vhostsum* die Anzahl der zurückgegebenen Listen-Elemente vom Typ *struct lvhost* eingetragen. Im letzten Listen-Element wird das Feld *vhostlast* mit „1“ gekennzeichnet.

Die Struktur *lvhost* ist in `<net.if.h>` wie folgt deklariert:

```
struct lvhost {
    int          lvhostlen;    /* length of memory for lvhostlist */
    unsigned short vhostsum;  /* number of vhosts delivered      */
    unsigned short vhostlast; /* last element if not zero        */
    int          vhost_num;   /* vhost number, must be greater 1 */
    short       vhost_flag;   /* vhost active ?                  */
    char        vsockethost[33]; /* sockethostname of vhost        */
    char        vbcamhost[9];  /* bcamhostname of vhost          */
};
```

SIOCGLVHNUM

Die Anzahl der aktiven virtuellen Hosts wird zurückgegeben.

Die Struktur *lifreq* ist in `<net.if.h>` wie folgt deklariert:

```
struct lifreq {
#define IFHWADDRLEN 6
#define LIFNAMSIZ 32
    char          lifr_name[LIFNAMSIZ];
    union {
        int          lifru_addrlen;
        unsigned int lifru_ppa;
    } lifr_lifru1;
#define lifr_addrlen lifr_lifru1.lifru_addrlen
#define lifr_ppa     lifr_lifru1.lifru_ppa
    unsigned int  lifr_movetoindex;
    union {
        struct sockaddr_storage lifru_addr;
        struct sockaddr_storage lifru_dstaddr;
    }
};
```

```

        struct sockaddr_storage lifru_broadaddr;
        struct sockaddr_storage lifru_token;
        struct sockaddr_storage lifru_subnet;
        struct sockaddr        lifru_hwaddr;
        int                     lifru_index;
    union {
        unsigned int            lifru_flags_0,lifru_flags_1;
        u_int64_t               lifru_flags;
    } lifr_lifruflags;
        int                     lifru_metric;
        unsigned int            lifru_mtu;
        char                    lifru_data[1];
        char                    lifru_enaddr[6];
        int                     lif_muxid[2];
        struct lif_nd_req        lifru_nd_req;
        struct lif_ifinfo_req    lifru_ifinfo_req;
        char                    lifru_groupname[LIFNAMSIZ];
        unsigned int            lifru_delay;
    } lifr_lifru;
#define lifr_addr        lifr_lifru.lifru_addr
#define lifr_dstaddr    lifr_lifru.lifru_dstaddr
#define lifr_broadaddr  lifr_lifru.lifru_broadaddr
#define lifr_token       lifr_lifru.lifru_token
#define lifr_subnet     lifr_lifru.lifru_subnet
#define lifr_index      lifr_lifru.lifru_index
#define lifr_flags      lifr_lifru.lifr_lifruflags.lifru_flags
#define lifr_flags_l    lifr_lifru.lifr_lifruflags.lifru_flags_1
#define lifr_flags_h    lifr_lifru.lifr_lifruflags.lifru_flags_0
#define lifr_metric     lifr_lifru.lifru_metric
#define lifr_mtu        lifr_lifru.lifru_mtu
#define lifr_data       lifr_lifru.lifru_data
#define lifr_enaddr     lifr_lifru.lifru_enaddr
#define lifr_index      lifr_lifru.lifru_index
#define lifr_ip_muxid   lifr_lifru.lif_muxid[0]
#define lifr_nd         lifr_lifru.lifru_nd_req
#define lifr_ifinfo     lifr_lifru.lifru_ifinfo_req
#define lifr_groupname  lifr_lifru.lifru_groupname
#define lifr_delay      lifr_lifru.lifru_delay
#define lifr_hwaddr     lifr_lifru.lifru.lifru_hwaddr
};

```

Die Struktur *sockaddr_storage* ist in `<sys.socket.h>` wie folgt deklariert:

```
#define _SS_MAXSIZE      128          /* Implementation specific max size */
#define _PADSIZE        (_SS_MAXSIZE - (sizeof(u_int64_t) + 8 ))

struct sockaddr_storage {
    sa_family_t      ss_family;      /* address family */
#define __ss_family   ss_family
    char             res[6];         /* reserved for alignment */
    u_int64_t        addr;          /* address */
    char             pad[_PADSIZE]; /* pad up to max size */
};
```

Die Struktur *lifnum* ist in `<net.if.h>` wie folgt deklariert:

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

Returnwert

-1:

bei Fehler. *errno* wird gesetzt, um den Fehler anzuzeigen.

Fehleranzeige durch *errno*

EBADF

Der Parameter *s* ist kein gültiger Deskriptor.

EINVAL

request oder *arg* sind für dieses Gerät (Interface, Socket) nicht gültig.

soc_poll() - Ein-/Ausgabe multiplexen

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

```
Kernighan-Ritchie-C:
int soc_poll(fds, nfds, timeout);

struct pollfd fds[];
unsigned long nfds;
int timeout;
```

```
ANSI-C:
int soc_poll(struct pollfd fds[], unsigned long nfds, int timeout);
```

Beschreibung

Die Funktion *soc_poll()* überprüft eine Menge von Socket-Deskriptoren, die mit einem Array von Struktur-Elementen vom Typ *pollfd* übergeben werden. Abhängig vom gewünschten Test wird in jedem deskriptorspezifischen Struktur-Element vermerkt, ob auf diesem Socket-Deskriptor Nachrichten empfangen oder gesendet werden können, oder ob besondere Ereignisse aufgetreten sind.

Die Funktion *soc_poll()* wird in den Adressfamilien AF_INET, AF_INET6 und AF_ISO unterstützt.

Der Parameter *fds* ist ein Zeiger auf das vom Aufrufer zu übergebende Array mit je einem Element vom Typ *struct pollfd* für jeden zu testenden Socket-Deskriptor.

Der Parameter *nfds* gibt die Menge der zu testenden Deskriptoren an.

Der Parameter *timeout* spezifiziert die maximale Wartezeit in Sekunden, die der Funktion *soc_poll()* für die Prüfung der Deskriptoren zur Verfügung steht, falls kein Ereignis eingetreten ist:

- Wert von *timeout* = 0: Keine Wartezeit, es werden lediglich alle markierten File-Deskriptoren geprüft.
- Wert von *timeout* = -1: *soc_poll()* blockiert, bis ein Ereignis an mindestens einem der ausgewählten File-Deskriptoren eintritt.

Struktur pollfd

Die Struktur *pollfd* ist in `<sys.poll.h>` wie folgt deklariert:

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

Der Socket-Deskriptor *fd* kennzeichnet den zu prüfenden Socket.

events kennzeichnet die zu prüfenden Ereignisse auf diesem Socket.

revents liefert das Prüfergebnis zurück. Die Bits POLLNVAL, POLLERR, POLLHUP werden in *revents* immer gesetzt, wenn die Bedingung dafür erfüllt ist, unabhängig von den in *events* gesetzten Bits.

Die Ereignisabfrage im Elementfeld *events* ist mit den folgenden Bitmasken möglich:

- POLLIN
- POLLOUT

Folgende Bitmasken werden weder im Elementfeld *events* unterstützt, noch im Elementfeld *revents* gesetzt:

- POLLPRI
- POLLRDNORM
- POLLWRNORM
- POLLRDBAND
- POLLWRBAND

Folgende Ereignisanzeigen in der Bitmaske des Elementfeldes *revents* sind möglich:

POLLIN

Daten können bei einer bestehenden Verbindung nicht-blockierend gelesen werden.

POLLOUT

Daten können bei einer bestehenden Verbindung nicht-blockierend geschrieben werden.

POLLNVAL

Der mit dem Socket-Deskriptor ausgewählte Socket ist nicht verfügbar, ist nicht vom Typ `SOCK_STREAM` oder besitzt nicht den Status, der eine aktive Verbindung anzeigt. Dieses Flag wird nur als Ergebnis in das Feld *revents* geschrieben.

POLLERR

Dem ausgewählten Socket wurde ein Fehler gemeldet und die Verbindung ist inaktiv. Dieses Flag wird nur als Ergebnis in das Feld *revents* geschrieben.

POLLHUP

Die Anwendung oder das Transportsystem haben die Verbindung geschlossen. Dieses Flag wird nur als Ergebnis in das Feld *revents* geschrieben.

Wenn für einen Socket-Deskriptor *fd* ein negativer Wert angegeben wird, wird dieser Wert ignoriert und das *revents* Feld auf 0 gesetzt.

Returnwert

0:

Die im Parameter *timeout* angegebene Zeit ist abgelaufen, ohne dass eine Ereignisanzeige gesetzt wurde.

>0

Der positive Wert zeigt die Anzahl der Socket-Deskriptoren an, für die mindestens eine Ereignisanzeige im *revents*-Feld gesetzt wurde.

-1

bei Fehler. *errno* wird gesetzt, um Fehler anzuzeigen.

Fehler**EACCES**

Die Sockets-Funktion wird vom aufgerufenen Subsystem nicht unterstützt.

EINTR

Der *soc_poll()*-Aufruf wurde von *soc_wake()* unterbrochen.

EINVAL

Der Wert von *nfds* ist größer, als die maximal zugelassene Anzahl von Socket-Deskriptoren. Den maximalen Wert ermitteln Sie durch Aufruf von *getdtablesize()*.

Siehe auch

`select()`

soc_putc() (putc) - Zeichen in den Ausgabepuffer schreiben

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

Kernighan-Ritchie-C:
int soc_putc(c, s);

char c;
int s;

ANSI-C:
int soc_putc(char c, int s);
```

Beschreibung

Die Funktion *soc_putc()* wird nur in den Adressfamilien AF_INET und AF_INET6 unterstützt und kann nur auf Stream-Sockets angewendet werden.

Die Funktion *soc_putc()* schreibt das Zeichen *c* in den Ausgabepuffer des Sockets *s*. Die Zeichen werden bis zur maximalen Kapazität von 32 760 byte in den Ausgabepuffer des Sockets geschrieben und erst danach automatisch an das Transportsystem BCAM weitergereicht. Ein unbedingtes Entleeren des Ausgabepuffers in Richtung BCAM lässt sich mit der Socket-Funktion *soc_flush()* erreichen.

Returnwert

≠EOF:
bei Erfolg

EOF:
falls wegen Dateiende (EOF) nicht in den Puffer geschrieben werden konnte.

Siehe auch

soc_flush()

soc_puts() (puts) - Character-String in den Ausgabepuffer schreiben

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

Kernighan-Ritchie-C:
int soc_puts(s, d);

char *s;
int d;

ANSI-C:
int soc_puts(char* s, int d);
```

Beschreibung

Die Funktion *soc_puts()* wird nur in den Adressfamilien AF_INET und AF_INET6 unterstützt und kann nur auf Stream-Sockets angewendet werden.

Die Funktion *soc_puts()* schreibt den Character-String *s* in den Ausgabepuffer des Sockets *d*.

Returnwert

Nullzeiger:
bei Erfolg

EOF:
falls wegen Dateiende (EOF) nicht in den Puffer geschrieben werden konnte, oder im Fehlerfall.

Siehe auch

soc_flush()

soc_read(), soc_readv() (read, readv) - Nachricht von einem Socket empfangen

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

Kernighan-Ritchie-C:
int soc_read(s, buf, nbytes);

int s;
char *buf;
int nbytes;

int soc_readv(s, iov, iovcnt);

int s;
struct iovec *iov;
int iovcnt;

ANSI-C:
int soc_read(int s, char* buf, int nbytes);
int soc_readv(int s, struct iovec* iov, int iovcnt)
```

Beschreibung

Die Funktionen *soc_read()* und *soc_readv()* lesen Nachrichten

- von einem Stream-Socket *s* der Adressfamilie AF_INET oder AF_INET6,
- von einem Socket *s* der Adressfamilie AF_ISO.

soc_read() und *soc_readv()* können nur bei einem Socket verwendet werden, für den eine Verbindung aufgebaut ist.

Bei *soc_read()* zeigt der Parameter *buf* auf das erste Byte des Empfangspuffers *buf*. *nbytes* spezifiziert die Länge (in Bytes) des Empfangspuffers und damit die maximale Nachrichtenlänge.

Bei *soc_readv()* werden die empfangenen Daten im Vektor mit den Elementen *iov*[0], *iov*[1], ... ,*iov*[*iovcnt*-1] abgelegt. Die Vektorelemente sind Objekte vom Typ *struct iovec*. *iovcnt* gibt die Anzahl der Vektorelemente an.

Die Struktur *iovec* ist in `<sys.uio.h>` wie folgt deklariert:

```
struct iovec
{
    caddr_t iov_base; /* Puffer für Hilfsdaten */
    int iovlen;      /* Pufferlänge */
};
```

Im Parameter *iov* wird die Adresse des Vektors übergeben. Jedes Vektorelement spezifiziert Adresse und Länge eines Speicherbereichs, in den *soc_readv()* die vom Socket *s* empfangenen Daten einliest. *soc_readv()* füllt einen Bereich nach dem anderen mit Daten, wobei *soc_readv()* immer erst dann zum nächsten Bereich übergeht, wenn der aktuelle Bereich vollständig mit Daten gefüllt ist.

Returnwert

≥0:

bei Erfolg (Anzahl der empfangenen Bytes)

-1:

bei Fehler. *errno* wird gesetzt, um den Fehler anzuzeigen.

Fehleranzeige durch *errno*

EBADF

Der Parameter *s* ist kein gültiger Deskriptor.

EIO

Es sind keine Benutzerdaten zum Lesen vorhanden.

ENETDOWN

Die Verbindung zum Netzwerk ist nicht mehr aktiv.

ENOTCONN

Für den Socket besteht keine Verbindung.

EOPNOTSUPP

Der Socket-Typ wird nicht unterstützt. Der Socket ist nicht vom Typ `SOCK_STREAM`.

EWOULDBLOCK

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

EPIPE

Die Verbindung ist abgebaut.

Siehe auch

`connect()`, `getsockopt()`, `recv()`, `select()`, `send()`, `soc_ioctl()`, `soc_read()`, `soc_write()`, `soc_writev()`, `socket()`

soc_wake() - Eine mit select() oder soc-poll() wartende Task wecken

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

Kernighan-Ritchie-C:
int soc_wake(pid);
int *pid;

ANSI-C:
int soc_wake(int* pid);
```

Beschreibung

Mit *soc_wake()* kann der Benutzer die durch **pid* identifizierte Task wecken, wenn diese mit *select()* oder *soc_poll()* wartet. Der Aufruf von *soc_wake()* bewirkt, dass sich *select()* bzw. *soc_poll()* mit Returnwert „-1“ und Fehler EINTR beendet.

soc_wake() kann eine andere Task mit derselben Benutzerkennung wecken. Falls *soc_wake()* aus einer Signal-Routine aufgerufen wird, kann *soc_wake()* auch die eigene Task wecken.

**pid* ist eine Variable, die mit dem Wert der Task Sequence Number (TSN) der wartenden Task versorgt werden muss. Die Task **pid* muss zum Zeitpunkt des Aufrufs von *soc_wake()* bereits existieren und einen Socket geöffnet haben. Falls die Task **pid* nicht mit *select()* wartet, wird das durch *soc_wake()* gesetzte Signal dem nächsten *select()*-Aufruf zugeordnet.

Returnwert

- 0:
bei Erfolg
- 1:
bei Fehler. *errno* wird gesetzt, um den Fehler anzuzeigen.

Fehleranzeige durch errno

ESRCH

Es gibt keine Task mit der Nummer **pid*. Entweder ist **pid* eine ungültige Tasknummer, oder es ist noch kein SOCKETS(BS2000)-Programm geladen.

EACCES

Keine Zugriffsberechtigung. Die Task **pid* hat eine andere Benutzerkennung.

soc_write(), soc_writev() (write, writev) - Nachricht von Socket zu Socket senden

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

Kernighan-Ritchie-C:
int soc_write(s, buf, nbytes);

int s;
char *buf;
int nbytes;

#include <sys.types.h>
#include <sys.uio.h>

int soc_writev(s, iov, iovcnt)

int s;
struct iovec *iov;
int iovcnt;

ANSI-C:
int soc_write(int s, char* buf, int nbytes);
int soc_writev(int s, struct iovec* iov, int iovcnt);
```

Beschreibung

Die Funktionen *soc_write()* und *soc_writev()* unterstützen folgende Wege der Nachrichtenübermittlung:

- Nachrichten von einem Stream-Socket *s* an einen anderen Stream-Socket (AF_INET, AF_INET6)
- Nachrichten von einem „connected“ Datagramm-Socket an einen anderen Socket (AF_INET, AF_INET6)
- Nachrichten von einem Socket *s* der Adressfamilie AF_ISO an einen anderen Socket der Adressfamilie AF_ISO.

soc_write() und *soc_writev()* können nur benutzt werden, wenn zwischen den beiden Sockets eine Verbindung aufgebaut ist.

Bei *soc_write()* zeigt der Parameter *buf* auf das erste Byte des Sendepuffers und *nbytes* spezifiziert die Länge (in Bytes) des Sendepuffers.

Bei `soc_writev()` werden die zu sendenden Daten im Vektor mit den Elementen `iov[0]`, `iov[1]`, ... ,`iov[iovcnt-1]` bereitgestellt. Die Vektorelemente sind Objekte vom Typ `struct iovec`. `iovcnt` gibt die Anzahl der Vektorelemente an.

Die Struktur `iovec` ist in `<sys.uio.h>` wie folgt deklariert:

```
struct iovec{
    caddr_t   iov_base; /* Puffer für Hilfsdaten */
    int       iov_len;  /* Pufferlänge */
};
```

Im Parameter `iov` wird die Adresse des Vektors übergeben. Jedes Vektorelement spezifiziert Adresse und Länge eines Speicherbereichs, aus dem `soc_writev()` Daten ausliest, um sie an den Empfänger-Socket `s` zu senden.

Returnwert

≥0:

bei Erfolg (Anzahl der tatsächlich gesendeten Bytes).

-1:

bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehleranzeige durch `errno`

EBADF

Der Parameter `s` ist kein gültiger Deskriptor.

ECONNRESET

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

EINVAL

Ein Parameter spezifiziert einen ungültigen Wert.

EIO

E/A-Fehler. Die Nachricht konnte nicht an das Transportsystem übergeben werden.

ENETDOWN

Die Verbindung zum Netzwerk ist nicht mehr aktiv.

ENOTCONN

Für den Socket besteht keine Verbindung.

EOPNOTSUPP

Der Socket-Typ wird nicht unterstützt. Der Socket ist nicht vom Typ `SOCK_STREAM`.

EPIPE

Der Socket ist nicht für Schreiben aktiviert, oder der Socket ist verbindungsorientiert und der Partner hat die Verbindung beendet.

EWOULDBLOCK

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

Hinweis

Wird die Verbindung mit einem nicht-blockierenden Socket aufgebaut, kann es beim nächsten Funktionsaufruf zur *errno* EINPROGRESS kommen. Diese Meldung weist darauf hin, dass die Verbindung noch nicht in dem Zustand ist, in dem eine Datentransferphase möglich ist.

Siehe auch

connect(), getsockopt(), recv(), select(), soc_read(), soc_readv(), socket()

socket() - Socket erzeugen

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

```
Kernighan-Ritchie-C:
int socket(domain, type, protocol);

int domain, type, protocol;
```

```
ANSI-C:
int socket(int domain, int type, int protocol);
```

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 Protokolfamilie und somit die 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 Adressfamilien *AF_INET*, *AF_INET6* und *AF_ISO* unterstützt.

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

- *SOCK_STREAM*
- *SOCK_DGRAM*
- *SOCK_RAW*

In den Adressfamilien *AF_INET* und *AF_INET6* werden alle drei Typen unterstützt, in der Adressfamilie *AF_ISO* ist nur der Typ *SOCK_STREAM* definiert.

Der Parameter *protocol* wird nicht ausgewertet.

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 abfragen bzw. setzen.

Adressfamilien AF_INET und AF_INET6

Ein Socket vom Typ `SOCK_STREAM` ermöglicht eine sequenzielle, gesicherte, bidirektionale Datenübertragung über eine Verbindung. Bevor Daten an einen Stream-Socket gesendet oder von ihm empfangen werden können, muss der Stream-Socket mit einem anderen Stream-Socket verbunden sein.

Eine Verbindung zu einem anderen Socket wird hergestellt, indem der Socket die Verbindung zu einem Partner-Socket mit `connect()` anfordert und der Partner die Verbindung mit `accept()` bestätigt. Nach erfolgreichem Verbindungsaufbau können beide Kommunikationspartner Daten übertragen mit `soc_read()` bzw. `soc_readv()` und `soc_write()` bzw. `soc_writev()` oder vergleichbaren Aufrufen wie `send()` und `recv()`.

Ein Socket vom Typ `SOCK_DGRAM` unterstützt die Übertragung von Datagrammen. Datagramme sind verbindungslose, ungesicherte Nachrichten einer festen maximalen Länge. Mit der Funktion `sendto()` werden Datagramme von einem Datagramm-Socket an den im `sendto()`-Aufruf genannten Datagramm-Socket gesendet. Empfangen werden Datagramme mit der Funktion `recvfrom()`. `recvfrom()` liefert das nächste Datagramm zusammen mit der Adresse des Absenders.

Wenn mit der Funktion `connect()` der Kommunikationspartner eines Datagramm-Sockets voreingestellt wurde, können für diesen Datagramm-Socket auch die Funktionen `send()` und `recv()` verwendet werden.

Ein Socket vom Typ `SOCK_RAW` unterstützt die Übertragung von ICMP-/ICMPv6-Nachrichten.

Adressfamilie AF_ISO

Ein Socket vom Typ `SOCK_STREAM` ermöglicht eine satzorientierte, sequenzielle, gesicherte, bidirektionale Datenübertragung über eine Verbindung. Bevor Daten an einen Stream-Socket gesendet oder von ihm empfangen werden können, muss der Stream-Socket mit einem anderen Stream-Socket verbunden sein.

Eine Verbindung zu einem anderen Socket wird hergestellt, indem der Socket die Verbindung zu einem Partner-Socket mit `connect()` anfordert und der Partner die Verbindung mit `accept()` und einer der Funktionen `send()`, `soc_write()` oder `sendmsg()` bestätigt.

Nach erfolgreichem Verbindungsaufbau können beide Kommunikationspartner Daten übertragen mit `soc_read()` bzw. `soc_readv()` und `soc_write()` bzw. `soc_writev()` oder vergleichbaren Aufrufen wie `send()` und `recv()` oder `sendmsg()` und `recvmsg()`.

Returnwert

- ≥0: bei Erfolg. Der Wert bezeichnet einen nicht-negativen Deskriptor.
- 1: bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehleranzeige durch errno

EMFILE

Die Tabelle der Deskriptoren pro Task ist voll; die maximale Anzahl gleichzeitig bearbeitbarer Socket-Deskriptoren ist erreicht. Mit der Funktion *getdtablesize()* kann dieser Maximalwert ermittelt werden.

ENOBUFS

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

EAFNOSUPPORT

Die Adressfamilie wird von dieser Protokollfamilie nicht unterstützt. Die angegebene Adresse ist inkompatibel zum verwendeten Protokoll.

ENOMEM

Speicherengpass. Während der Ausführung der Funktion konnte nicht genügend virtueller Speicherplatz zugewiesen werden.

EPROTONOSUPPORT

Der Socket-Typ wird in dieser Domäne nicht unterstützt.

Siehe auch

`accept()`, `bind()`, `connect()`, `getsockname()`, `getsockopt()`, `listen()`, `recv()`, `recvfrom()`, `select()`, `send()`, `sendto()`, `soc_close()`, `soc_ioctl()`, `soc_read()`, `soc_readv()`, `soc_write()`, `soc_writev()`

7 SOCKETS(BS2000)-Schnittstelle für eine externe Börse

Dieses Kapitel beschreibt die Zusatz-Funktionen der Socket-Schnittstelle für BS2000 im Spezialmodus mit Nutzung einer externen Börse. Hier wurde die Möglichkeit geschaffen, durch die Angabe einer externen Börse mit einem gemeinsamen Wartepunkt verschiedene Ereignisse zu koordinieren.

Der Socketmodus (externer Wartepunkt oder nicht) wird mit dem ersten Socket-Aufruf eingestellt und ist dann statisch. Die Einstellung erfolgt mit einer *setsockopt()*-Subfunktion.

Um die Funktionalität zu erreichen, wird die Zusatzfunktion *soc_getevent()* und die Subfunktion SO_ASYNC für *setsockopt()* zur Verfügung gestellt.

7.1 Beschreibung der Zusatzfunktionen

setsockopt() - Socket-Optionen ändern

```
#include <sys.types.h>
#include <sys.socket.h>
#include <netinet.in.h> /* nur bei AF_INET oder AF_INET6 */
```

Kernighan-Ritchie-C:

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

int s;
int level;
int optname;
char *optval;
int optlen;
```

ANSI-C:

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

Beschreibung

Mit der Funktion *setsockopt()* kann der Benutzer über die Parameter *level*, *optname*, *optval* und *optlen* die Eigenschaften (Optionen) der Socket-Schnittstelle oder eines einzelnen Sockets *s* ändern.

level *SOL_GLOBAL*

In diesem Fall ist der aktuelle Parameterwert für *s* ohne Bedeutung. Deshalb sollte für *s* der Wert 0 angegeben werden.

| <i>optname</i> | <i>*optlen</i> | Wertebereich von <i>optval</i> |
|----------------|----------------|--|
| SO_ASYNC | 4 | Zeiger auf Kurzkenung der Ereigniskennung (Event-ID) |

Die Subfunktion *SO_ASYNC* ist nur für *setsockopt()* zulässig und wirkt nur, wenn es der erste Aufruf des Anwenders auf das Subsystem ist.

Ist das der Fall, wird das Subsystem für diesen Anwender auf einen Betrieb umgeschaltet, der die Koordination mit anderen Ereignissen über einen gemeinsamen Wartepunkt erlaubt. Mit *optval* muss dann die Kurzkenung der Ereigniskennung (Event-ID) der Börse angegeben werden, an die die Sockets die Kommunikations-Ereignisse zustellen sollen. Diese Events können dann mit der Funktion *soc_getevent* abgeholt werden.

Bemerkung

In diesem Betriebsmodus werden die mit *socket()* erzeugten Sockets automatisch im nicht-blockierenden Modus erzeugt.

soc_getevent() - Socket-Event abholen

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

```
ANSI-C:
```

```
int soc_getevent(struct aevent *exb_event);
```

Beschreibung

Die Funktion *soc_getevent()* liefert dem Aufrufer die Information über einen zugestellten Event, der an die vom Aufrufer genutzte Börse signalisiert wurde.

In der Ausgabestruktur *aevent* werden Standarddaten und event-spezifische Daten abgelegt.

Mit dem Signal an die Börse wird ein 2 Wort langer Postcode mit folgendem Aufbau übermittelt:

1. Wort:

Byte 1: Eventcode X'3E'

Byte 2: User Call Indicator

X'14' = IPv4-Event

X'15' = IPv6-Event

Byte 3: Event indicator

C'E' = Standard Event, siehe Liste

C'W' = Event wurde durch ein Wake Ereignis ausgelöst.
User Call Indicator = X'00'.

C'S' = Event wurde durch BCEND ausgelöst.
User Call Indicator = X'00' und 2. Wort = X'00000000'.

Byte 4: X'00'

2. Wort:

nicht definiert

Die Struktur *aevent* ist in `<sys.socket.h>` wie folgt deklariert:

```
struct aevent {
    int    fd;                /* socket filedescriptor */
    int    event;            /* Transportsystem Event */
    int    subevent;        /* Zusatzinformation zum Event */
    int    datalen;         /* An Caller übergebene Datenlänge */
    int    fd_errno;        /* errno */
    int    fd_array_cnt;     /* Anzahl der FD's, wenn Event ECLS */
    int    fd_array[FD_MAX]; /* FD's bei Event ECLS */
}
```

Returnwert

- 0:
bei Erfolg
- 1:
bei Fehler

mögliche Events:

EXB_ECLS

TSAP-Termination-Indication, zwangsweise Schließung des TSAP durch das Transportsystem.

Daten: *fd, event, fd_array_cnt, fd_array*

EXB_ERQQ

Connection-Request-Indication, Verbindungsaufbauwunsch.

Daten: *fd, event*

EXB_ERSP

Connection-Response-Indication, Partnerquittung zur Verbindungsaufbauforderung.

Daten: *fd, event*

EXB_EDIS

Disconnect-Indication, Verbindungsabbauforderung.

Daten: *fd, event*

EXB_EDTA

Data-Indication, TCP-Datenempfang.

Daten: *fd, event, datalen*

EXB_EDTU

Unitdata-Indication, UDP-Datenempfang.

Daten: *fd, event, datalen*

EXB_EERR

Error-Report-Indication
ICMP-Error-Message

EXB_EGOD

Data-Go-Indication, Datentransfer kann fortgesetzt werden.
Daten: *fd, event*

EXB_NOEV

Kein Event vorhanden.
Der Returnwert wird auf 1 gesetzt.

EXB_TRYL

Zurzeit kann kein Event abgeholt werden.
Es kann aber später wieder versucht werden.
Der Returnwert wird auf 1 gesetzt.

EXB_SHUT

Das Transportsystem BCAM ist beendet oder befindet sich gerade
in der Beendigungsphase.
Auch die Anwendung muss beendet werden.
Der Returnwert wird auf 1 gesetzt.

Bei Fehlern wird die *errno* gesetzt.

8 Softwarepaket SOCKETS(BS2000) V2.7

8.1 SOCKETS(BS2000)-Subsysteme

| | |
|---------|--|
| SOC-TP | Subsystem für Systemprogramme |
| SOC6 | Subsystem für Anwenderprogramme auf allen Hardware-Plattformen |
| SOC6-SP | Subsystem für spezielle Programme auf SX-Servern |
| SOC6-X8 | Subsystem für spezielle Programme auf SE Serie mit SU x86 und auf SQ-Servern |

8.2 SOCKETS(BS2000)-Programme

8.2.1 ping4

Eigenständiges Diagnose-Programm zur Ermittlung der Erreichbarkeit eines Host in einem IPv4-Netz. Es wird ein ICMP-Echo-Request versendet und getestet, ob der Host mit einem ICMP-Echo-Reply antwortet.

Online-Hilfe: `ping4 -h`

Beschreibung: siehe Handbuch „[BCAM Band 1/2](#)“

8.2.2 ping6

Eigenständiges Diagnose-Programm zur Ermittlung der Erreichbarkeit eines Host in einem IPv6-Netz. Es wird ein ICMPv6-Echo-Request versendet und getestet, ob der Host mit einem ICMPv6-Echo-Reply antwortet.

Online-Hilfe: `ping6 -h`

Beschreibung: siehe Handbuch „[BCAM Band 1/2](#)“

8.2.3 nslookup

Eigenständiges Programm zur Umsetzung von DNS-Namen in IPv4/IPv6-Adressen (lookup) und umgekehrt (reverse lookup).

Starten von nslookup

```
start-nslookup [-server address | name] address | name
```

oder

```
nslookup [-server address | name] address | name
```

Hilfe: nslookup -h

Usage: [-server address | name] address | name

Mit der Option `-server` können Sie den DNS-Namen oder die IPv4/IPv6-Adresse des Nameservers festlegen, der verwendet werden soll. Ohne Angabe dieser Option verwendet *nslookup* den ersten erreichbaren Nameserver, der in der LWRES-D-Konfigurationsdatei `SYSDAT.LWRES.D.nnn.RESOLV.CONF` eingetragen ist. *nnn* entspricht der LWRES-D-Version, derzeit aktuell ist 013.

Bei der Umsetzung von DNS-Namen in Adressen gibt *nslookup* sowohl IPv4- als auch IPv6-Adressen aus, falls vorhanden.

nslookup kann auch im interaktiven Modus ausgeführt werden. Das kann nützlich sein, wenn mehrere Abfragen gestartet werden sollen, weil nur einmal die Zeit für den Programmstart abgewartet werden muss.

Starten von nslookup im interaktiven Modus

```
start-nslookup oder nslookup
```

Erlaubte Kommandos:

`server:` gibt die in der `sysdat.lwresd.nnn.resolv.conf` eingetragenen nameserver aus.

Ist kein nameserver eingetragen, werden die loopback-Adressen 172.0.0.1 und `::1` ausgegeben.

Es wird grundsätzlich der erste erreichbare nameserver für Abfragen verwendet.

`server name oder adresse:` der angegebene server wird ab jetzt verwendet.

`adresse oder name:` der eingestellte nameserver wird wg. der Umsetzung der Adresse oder des Namens abgefragt.

`help oder ?:` es erfolgt eine usage - Ausgabe

`exit oder end:` nslookup wird beendet

8.3 SOCKETS(BS2000)-DNS-Zugang

Der Zugang zum DNS erfolgt im Rahmen des Software-Pakets openNet Server V3.6, das SOCKETS(BS2000) V2.7 beinhaltet, über das Programm Light Weight RESolver Daemon (LWRESD) auf der Basis des Software-Pakets bind9.x (siehe Handbuch „[BCAM Band 1/2](#)“).

SOCKETS(BS2000) V2.7 stellt die Verbindung zum LWRESD mit einem integrierten LWRES-Client her.

Wenn das DNS für die aufgerufene DNS-Auskunftsfunction keine entsprechende Information zur Verfügung stellt, wird anschließend versucht diese Information vom Transportsystem BCAM zu bekommen.

Das gilt sowohl für die Umsetzung von Rechnernamen, als auch für FQDN (Fully Qualified Domain Name).

In der Standard-Konfiguration ist der LWRESD unter der lokalen Loopback-Adresse und dem Port 921 erreichbar.

Für einen produktionsbedingten erforderlichen Workaround, bzw. für eine zusätzliche spezielle lokale private DNS-Konfiguration kann es erforderlich sein, dass ein zweiter LWRESD im Parallelbetrieb gestartet werden muss.

Das wird mit einem zusätzlichen Kommando-Operanden SCOPE in den entsprechenden Administrationskommandos für den LWRESD (siehe Handbuch „[BCAM Band 1/2](#)“) erreicht.

Die Werte für den Operanden SCOPE sind:

- | | |
|------------|--|
| *STD: | Die Defaulteinstellung; es gibt nur einen LWRESD; bzw. es wird der Standard-LWRESD ausgewählt. |
| *LWRES-NAS | Workaround für einen zweiten LWRESD, der mit der Loopback-Adresse und dem Standard-Port 921 adressiert wird. |
| *LOCAL-DNS | Auswahl eines zweiten LWRESD, der mit einen DNS-Server für ein spezielles lokales Netz kommuniziert (Volle Unterstützung erst in openNet Server > V3.6). |

Sind zwei LWRESD aktiv, dann müssen diese unter unterschiedlichen Adressen und/oder Ports erreichbar sein.

Um neue Konfigurationsmöglichkeiten mit einem LWRESD anbieten zu können, wurde die Auswahl der möglichen Schlüsselworte in der LWRESD-Konfigurationsdatei SYSDAT.LWRESD.*nnn*.RESOLV.CONF erweitert. *nnn* entspricht der LWRESD-Version, derzeit aktuell ist 013.

Die zusätzlichen Schlüsselworte sind:

lwserver IPv4- oder IPv6-Adresse

Unter dieser Adresse wird der Listen-Socket des LWRESD geöffnet, damit er durch den LWRES-Client erreichbar ist.

lwlport Portnummer

Unter dieser Portnummer wird der Listen-Socket des LWRESD eröffnet.

destport Portnummer

Hier kann ein von der Standard-Portnummer 53 abweichender Port für die eingetragenen Nameserver angegeben werden. Damit werden Tests von Nameservern ermöglicht, die zu diesem Zweck unter einer Nicht-Standard Portnummer gestartet wurden.

Änderungen unter *lwserver*, *lwlport* oder *destport* können nicht dynamisch durch ein erneutes Einlesen der LWRESD-Konfigurationsdatei (Kommando RELOAD-LWRESD) aktiviert werden.

Es ist ein LWRESD-Neustart erforderlich, der mit der Kommandofolge STOP-LWRESD und START-LWRESD oder mit dem Kommando RESTART-LWRESD eingeleitet werden kann.

Die IPv4- oder IPv6-Adresse und der UDP-Port, unter dem der ausgewählte LWRESD erreichbar ist, sind in BCAM abgelegt und können mit den BCAM-Kommandos SHOW-DNS-ACCESS und MODIFY-DNS-ACCESS ausgelesen bzw. modifiziert werden. Ab dem nächsten DNS-Zugriff von SOCKETS(BS2000) werden diese Daten verwendet. Ein Neustart von SOCKETS(BS2000) nach einer Modifizierung der Adressen ist nicht mehr erforderlich.

Mit den *getsockopt()* / *setsockopt()* Subfunktionen SO_LWRESINFO; SO_LWADDR; SO_LWADACT wird es einem Sockets-Programm ermöglicht, einen anderen als den konfigurierten LWRESD zu testen.

8.4 SOCKETS(BS2000) - Abfrage an FQDN-Datei

Ab openNet Server V3.5 existiert eine von BCAM verwaltete FQDN-Datei. Mit einem Eintrag in dieser Datei kann ein FQDN auf einen BCAM-Namen umgesetzt werden. Diese Information wird mit berücksichtigt, wenn SOCKETS(BS2000) eine Anfrage an BCAM stellt, weil kein DNS-Server zur Verfügung stand, bzw. weil das DNS keine Information zu einer Anfrage liefern konnte (siehe Handbuch „[BCAM Band 1/2](#)“).

8.5 SOCKETS(BS2000)-Anwenderprogramm produzieren

Das SOCKETS(BS2000) V2.7-Subsystem *SOC6* ist kompatibel zur Vorgängerversion.

Ab SOCKETS(BS2000) V2.1 sind die Include-Dateien der Anwenderprogramm-Bibliothek kompatibel für den Kernighan-Ritchie-, den ANSI-C- und den C++-Modus, d.h. es sind entsprechende Defines in den Include-Dateien implementiert.

8.5.1 Software-Voraussetzungen

Für den Einsatz von SOCKETS(BS2000) V2.7 wird folgende Software benötigt:

- openNet Server V3.6
- BS2000 C-Compiler \geq V3.0

8.5.2 Programmerstellung

- Für den Compilerlauf ist zusätzlich zu den Bibliotheken mit den Private Header Files und den Header Files des C-Laufzeitsystems von SOCKETS(BS2000) nur die Header File-Bibliothek SYSLIB.SOCKETS.027 erforderlich.
- Beim Binden ist von SOCKETS-BS2000 keine Resolve-Bibliothek erforderlich.



Die Subsystem-Entries der genutzten SOCKETS(BS2000)-Funktionen werden vom Binder als Unresolved Externs gemeldet. Zum Ablaufzeitpunkt des Programms werden sie jedoch durch das Sockets-Subsystem aufgelöst.

Literatur

Die Handbücher finden Sie im Internet unter <http://manuals.ts.fujitsu.com>. Handbücher, die mit einer Bestellnummer angezeigt werden, können Sie in auch gedruckter Form bestellen.

C/C++

C/C++-Compiler

Benutzerhandbuch

C/C++

C-Bibliotheksfunktionen

Benutzerhandbuch

openNet Server

BCAM Band 1/2

Benutzerhandbuch

interNet Services

Systemverwalterhandbuch

interNet Services

Benutzerhandbuch

SNMP Management für BS2000

Benutzerhandbuch

RFCs

Umfassende Informationen zu den Request for Comments (RFCs) finden Sie auf der Home Page der Internet Engineering Task Force (IETF):

www.ietf.org

Stichwörter

<arpa.inet.h> 21
<ioctl.h> 21
<iso.h> 21
<net.if.h> 21
<netdb.h> 21, 65
<netinet.in.h> 21
<sys.poll.h> 21
<sys.socket.h> 21
<sys.time.h> 21
<sys.uio.h> 21

A

abbauen
 Verbindung 51, 115
 Voll-Duplex-Verbindung 204
abfragen
 Adresszuordnung 150
 BCAM-Hostnamen 141
 Hostname 145
 Informationen über Protokolle 150, 153
 Lesebereitschaft 194
 Local-Adresse der Sockets-Verbindung 159
 Namen des Kommunikationspartners 153
 Namen/Adresse eines Sockets 159
 Portnummer 157
 Protokollnummer 155
 Rechneradresse 136, 146
 Rechnernamen 136, 146
 Schreibbereitschaft 194
 Service-Namen 157
 Services 136
 Socket auf Verbindungsanforderungen 185
 Socket-Option 77
 Socket-Typ 167

 Statusinformation 208
abfragen Remote-Adresse der Sockets-
 Verbindung 153
abhören (Socket) siehe listen()
abortive disconnect 204
Accept-Socket 82
accept() 38, 49, 74, 88
 Beispiel 38, 39, 49, 92
 Funktionsbeschreibung 123
addrinfo-Struktur 133, 137
 Speicherfreigabe 133
Adress-Struktur 22, 24
 sockaddr_in 25, 31
 sockaddr_in6 26, 31
 sockaddr_iso 28, 31
Adressbereich
 Multicast (IPv4) 75
 Multicast (IPv6) 75
Adresse 22
 automatisch zuordnen 35, 38
 INADDR_ANY 33
 INADDR6_ANY 33
 Internet 66
 lokal 37
 Netz 65, 66
 Protokoll 68
 Socket 24, 31
 umwandeln 65
 Wildcard 33
 zuordnen 31, 126
Adresse siehe auch Name
Adressfamilie 22, 24, 115, 232
 AF_INET 200
 AF_INET6 200
Adressfamilie siehe auch Domäne

- Adressierung, Socket 24
 - Adressierungspaar 37
 - Adressumwandlung bei SOCKETS(BS2000) 65
 - Beispiel 71
 - Adresszuordnung abfragen 150
 - AF_INET 22, 25, 29, 31, 33, 37, 57, 62, 65, 76, 88, 96, 115, 131, 190, 200, 206, 229, 232, 233
 - Adressumwandlung 66
 - Domäne 226
 - Kommunikation 37
 - Verbindungsabbau 51
 - AF_INET-Kommunikationsdomäne 29
 - AF_INET6 22, 26, 29, 30, 31, 33, 37, 62, 65, 76, 88, 96, 115, 131, 190, 200, 229
 - Adressumwandlung 66
 - Domäne 226
 - Kommunikation 37
 - Verbindungsabbau 51
 - AF_INET6-Kommunikationsdomäne 30
 - AF_ISO 22, 28, 29, 30, 32, 49, 63, 92, 100, 115, 131, 190, 200, 206, 226, 229, 233
 - Kommunikation 48
 - Verbindungsabbau 54
 - anfordern
 - Verbindung 37, 48, 88, 115, 130
 - Verbindung (Client-Beispiel) 41
 - annehmen
 - Verbindung 38, 49, 123
 - Verbindung (Server-Beispiel) 42
 - anstehende Verbindungsanforderung
 - überprüfen auf 57, 60
 - Anwenderprogramm produzieren 245
 - asymmetrisch
 - Protokoll 87
 - Verbindung 37
 - aufbauen, Verbindung 37, 115
 - Ausgabe multiplexen 221
 - Ausgabepuffer 209, 224, 225
 - Ausnahmebedingung prüfen (Deskriptor) 194
 - automatische Adresszuordnung 35, 38
- B**
- BCAM-Hostname 28, 167
 - abfragen 141
- beenden, Verbindung siehe Verbindung abbauen
 - Beschreibungsformat, Socket-Funktion 113
 - Bibliothek SOCKETS(BS2000) 65
 - bidirektionale Datenübertragung 22
 - bind() 31, 33
 - Beispiel 31, 88, 92, 100, 104
 - Funktionsbeschreibung 126
 - binden, Socket
 - siehe Namen bzw. Adresse zuordnen
 - Bitmaske 55, 195
 - blockieren 39, 50, 74, 195, 196
 - Broadcast
 - Nachrichten 75
 - Byte-Reihenfolge 128
 - Netz 70, 128, 181
 - Rechner 70, 128, 181
 - umwandeln 70, 128
 - Byteorder-Makros 128
- C**
- C Kernighan/Ritchie 245
 - C-Header-File siehe Include-Datei
 - C-Include-Datei siehe Include-Datei
 - CFRM_DATA 63, 190
 - Character-String siehe String
 - Client 37, 41, 71
 - Task 87
 - Verbindung anfordern (Beispiel) 41
 - Verbindung initiieren 131
 - verbindungslos (Beispiel) 110
 - verbindungsorientiert (Beispiel) 96
 - Client-/Server-Modell 87
 - Kommunikation (Beispiel) 41
 - cmsghdr-Struktur 192, 202
 - CONN_DATA 63, 190
 - connect() 37, 45, 48, 74
 - Beispiel 37, 96, 100
 - Funktionsbeschreibung 130
- D**
- Darstellungsmittel (typografische Gestaltungsmittel) 13
 - Datagramm 233
 - empfangen 187

- empfangen (Beispiel) 45
- senden 198
- Datagramm-Socket 23, 29, 44, 64, 131, 188, 192
 - Eigenschaften 23, 233
 - erzeugen 29, 30, 233
 - siehe auch SOCK_DGRAM
- Datei, Include- 21, 245
- Daten
 - empfangen 116, 187, 226
 - senden 116, 197, 229
- Daten übertragen 40, 50
 - aus Ausgabepuffer 209
 - bidirektional 22
 - gesichert und sequenziell 22
 - verbindungslose Kommunikation 44
 - verbindungsorientierte Kommunikation 40, 50
- Deskriptor 29, 112
 - Ausnahmebedingung prüfen 194
 - Lesebereitschaft prüfen 194
 - Schreibbereitschaft prüfen 194
- Deskriptormenge 55, 195
 - manipulieren 195
- Diagnose-Ausgabe-Level festlegen 163
- DISC_DATA 63, 190
- disconnect
 - abortive 204
 - graceful 204
- DNS-Resolver
 - Funktionalität 243
- Domäne 22, 29, 38, 39
 - AF_INET 22, 29, 31, 37, 57, 62, 65, 88, 96, 115, 131, 190, 206, 226, 229, 233
 - AF_INET6 29, 30, 31, 37, 62, 65, 88, 96, 115, 131, 190, 226, 229
 - AF_ISO 22, 29, 30, 32, 48, 49, 63, 92, 100, 115, 131, 190, 200, 206, 226, 229, 233
 - festlegen 232
- dynamischen Speicher freigeben 176
- E**
- E/A-Multiplexen 55
 - Beispiel 57, 60
 - mit select() 55, 194
- empfangen
 - Datagramm 187
 - Datagramm (Beispiel) 45
 - Daten 116, 187, 226
 - Nachricht 190, 226
- Error-Code getaddrinfo() 135
- erweiterte Socket-Funktionen 73
- erzeugen, Socket 232
- Timeout 195
- Ein-/Ausgabe, nicht-blockierend 198
- Eingabe multiplexen 221
- Eingabepuffer 210, 211
- F**
- FD_CLR 195
- FD_ISSET 56, 195
- FD_SET 195
- Fehler beim Verbindungsaufbau 38, 48
- flags 192
- FQDN
 - Fully Qualified Domain Name 149
- FQDN-Datei 244
- freeaddrinfo()
 - Funktionsbeschreibung 133
- freehostent()
 - Funktionsbeschreibung 134
- freigeben
 - addrinfo-Struktur 133
 - hostent-Struktur 134, 135
- Fully Qualified Domain Name (FQDN) 149
- Funktion siehe Socket-Funktion
- G**
- gai_strerror()
 - Funktionsbeschreibung 135
- gesicherte Datenübertragung 22
- Gestaltungsmittel, typografisch 13
- getaddrinfo()
 - Error-Code 135
 - Funktionsbeschreibung 136
- getbcamhost()
 - Beispiel 92, 100
 - Funktionsbeschreibung 141
- getdtablesize()

Funktionsbeschreibung 142
gethostbyaddr() 66
 Funktionsbeschreibung 143
gethostbyname() 66
 Beispiel 96, 110
 Funktionsbeschreibung 143
gethostname()
 Funktionsbeschreibung 145
getipnodebyaddr() 66
 Funktionsbeschreibung 146
getipnodebyname() 66
 Funktionsbeschreibung 146
getnameinfo()
 Funktionsbeschreibung 150
getpeername()
 Funktionsbeschreibung 153
getprotobyname()
 Funktionsbeschreibung 155
getservbyname() 69
 Anwendungsbeispiel 69
 Funktionsbeschreibung 157
getservbyport() 69
 Funktionsbeschreibung 157
getsockname()
 Funktionsbeschreibung 159
getsockopt() 77
 Beispiel 77
 Funktionsbeschreibung 161, 236
graceful disconnect 204
Grundlagen von SOCKETS(BS2000) 19

H

Handoff 80, 202
Handoff-Prozedur 175
Header-File siehe Include-Datei
Hilfsfunktionen 120
Hoplimit (Multicast) 169, 171
HOST-ALIASING 78
hostent-Struktur 66, 67, 134, 144, 148
 Speicher freigeben 134, 135
Hostname
 siehe auch BCAM-Hostname
Hostnamen abrufen 145
htonl() 70

Beschreibung 128
htons() 70
 Beschreibung 128

I

ICMP

Internet Control Message Protocol 23
ICMP-Echo-Request 85
ICMP-Fehlermeldungen 169, 171
ICMP-Protokoll 85
 Socket 23
ICMP-Protokoll-Header 85
ICMPv6
 Internet Control Message Protocol for
 IPv6 23
ICMPv6-Echo-Request 85
ICMPv6-Protokoll 86
ICMPv6-Protokoll-Header 85
if_freenameindex()
 Funktionsbeschreibung 176
if_indextoname()
 Funktionsbeschreibung 177
if_nameindex()
 Funktionsbeschreibung 178
if_nametoindex()
 Funktionsbeschreibung 179
ifname 179
INADDR_ANY 33
INADDR6_ANY 33
Include-Datei 21, 245
 arpa.inet.h 21
 ioctl.h 21
 iso.h 21
 net.if.h 21
 netdb.h 21, 65
 netinet.in.h 21
 sys.poll.h 21
 sys.socket.h 21
 sys.time.h 21
 sys.uio.h 21
inet_addr()
 Funktionsbeschreibung 180
inet_lnaof()
 Funktionsbeschreibung 180

- inet_makeaddr()
 - Funktionsbeschreibung 180
- inet_netof()
 - Funktionsbeschreibung 180
- inet_network()
 - Funktionsbeschreibung 180
- inet_ntoa()
 - Funktionsbeschreibung 180
- inet_ntop() 66
 - Funktionsbeschreibung 183
- inet_pton() 66
 - Funktionsbeschreibung 183
- installieren, SOCKETS(BS2000) 241
- Interface-Index 177, 178
 - ausgeben 179
- Interface-Namen
 - ausgeben 177
 - Liste ausgeben 178
- Internet Control Message Protocol (ICMP) 23
- Internet Control Message Protocol for IPv6 (ICMPv6) 23
- Internet-Adresse 25, 26, 31, 35, 66
 - automatisch zuordnen 35
 - manipulieren 119, 181, 183
 - mit Wildcard zuordnen 33
 - Punktschreibweise 181
- Internet-Domäne 22, 29, 37, 38, 39, 232
- iovec-Struktur 191, 201, 227, 230
- IP_ADD_MEMBERSHIP 169
- IP_DROP_MEMBERSHIP 169
- IP_MULTICAST_IF 169, 171
- IP_MULTICAST_LOOP 169, 171
- IP_MULTICAST_TTL 169
- IP_RECVERR 169
- IP-Adresse siehe Internet-Adresse
- IPv4-Adresse
 - umwandeln in Rechnernamen 67
- IPv4-Internet-Adresse
 - manipulieren 180
- IPV6_JOIN_GROUP 171
- IPV6_LEAVE_GROUP 171
- IPV6_MULTICAST_HOPS 171
- IPV6_RECVERR 171
- IPV6_V6ONLY 171
- IPv6-Adresse
 - umwandeln in Rechnernamen 66
- ISO 22
- ISO-Kommunikationsdomäne 29, 30
 - siehe auch AF_ISO
- ISO-Transportservice 22, 30
- K**
- Kernighan/Ritchie-C 245
- Kommunikation
 - in AF_INET 37
 - in AF_ISO 48
 - verbindungslos 23, 44, 64
 - verbindungslos (Beispiele) 45
 - verbindungsorientiert 22, 37, 48, 50
 - verbindungsorientiert (Beispiele) 41
- Kommunikationsanwendung 9, 20
- Kommunikationsdomäne siehe Domäne
- Kommunikationsendpunkt 22
 - siehe auch Socket
- Kommunikationspartner, Namen abfragen 153
- Kompatibilität
 - Version 2.7 zu früheren Versionen 14
- L**
- Lesebereitschaft prüfen (Deskriptor) 194
- lesen
 - String aus Eingabepuffer 211
 - Zeichen aus Eingabepuffer 210
- linger-Struktur 166
- Listen-Socket 82
- Listen-Sockets
 - mehrere auf einer Adresse 36
- listen() 38
 - Beispiel 38, 39, 49, 88, 92
 - Funktionsbeschreibung 185
- lokal
 - Adresse 37
 - Name 31
 - Portnummer 37
- M**
- Makro
 - FD_CLR 195

FD_ISSET 56, 195
FD_SET 195
htonl() 70, 128
htons() 70, 128
ntohl() 70, 128
ntohs() 70, 128
manipulieren
 Deskriptormenge 195
 Internet-Adresse 119, 183
 IPv4-Internet-Adresse 180
MSG_PEEK 188
msghdr-Struktur 191, 201
multiplexen, EA
 mit soc_poll 221
Multicast-Gruppe 169, 171
Multicast-Hoplimit 169, 171
Multicast-Nachrichten 75
Multihoming 166
Multihoming Support 36
multiplexen, E/A 55
 Beispiel 57, 60
 mit select() 55, 194
 mit soc_poll() 59, 221

N

Nachricht
 empfangen 187, 190, 226
 senden 198, 200, 229
Nachricht siehe auch Daten
Nagle-Algorithmus 172
Name
 des Kommunikationspartners 153
 eines Sockets abfragen 159
 lokal 31
 Protokoll 68
 Rechner 66
 Service 69
 Socket 24
 zuordnen 31, 33, 38, 126
Name siehe auch Adresse
Netz
 Adresse 65, 66, 182
 Adresse umwandeln 66
 Byte-Reihenfolge 70

 Nummer 182
 Netz-Byte-Reihenfolge 70, 128, 181
 Netzanbindung 20
 Netzselektor NSEL 28, 31, 48, 141
 Netzwerkprogrammierung 20
 nicht-blockierend
 Ein-/Ausgabe 196, 198
 Socket 74, 188, 198
 NSEL 28, 31, 48, 141
 nslookup 242
 ntohl() 70
 Beschreibung 128
 ntohs() 70
 Beschreibung 128
 Nummer eines Protokolls 68, 155

O

Optionen, Socket- 77
OSI-Protokoll 20

P

ping4 241
ping6 241, 242
pollfd-Struktur 222
Portnummer 25, 26, 31
 abfragen 157
 lokal 37
 mit Wildcard zuordnen 35
produzieren, Anwenderprogramm 245
protoent-Struktur 68
Protokoll 87
 asymmetrisch 87
 Namen umwandeln 68
 Nummer 68
 Nummer abfragen 155
 OSI 20
 symmetrisch 87
 TCP 22
 UDP 23, 44, 64
Protokollfamilie 22, 232
prüfen
 Lesebereitschaft 194
 Schreibbereitschaft 194
Punktschreibweise (Internet-Adresse) 181

- Q**
quasi-verbindungsorientiert 64
- R**
Raw-Socket 23, 29, 85
Raw-Socket siehe auch SOCK_RAW
Readme-Datei 11, 15
Rechner
 Adresse abfragen 143
 Byte-Reihenfolge 70, 128, 181
 Informationen über 143
 Namen abfragen 143
 Namen umwandeln 66, 143
Rechneradresse abfragen 136, 146
Rechnernamen abfragen 136, 146
recv() 40
 Beispiel 40, 88, 92
 Funktionsbeschreibung 187
recvfrom() 44
 Beispiel 44, 104
 Funktionsbeschreibung 187
recvmsg() 40, 49
 Beispiel 40, 50
 Funktionsbeschreibung 190
Resolver-Funktionalität 243
- S**
Satzgrenze (der übertragenen Daten) 23
schließen, Socket 51, 206
Schreibbereitschaft prüfen (Deskriptor) 194
schreiben
 String in Ausgabepuffer 225
 Zeichen in Ausgabepuffer 224
select() 55, 57, 59, 228
 Beispiel 55, 57, 59
 Funktionsbeschreibung 194
send() 40, 74
 Beispiel 40, 96, 100
 Funktionsbeschreibung 197
senden 229
 Datagramm 198
 Daten 116, 197, 229
 Nachricht 198, 200, 229
sendmsg() 40, 49
 Beispiel 40, 50, 92
 Funktionsbeschreibung 200
sendto() 44
 Beispiel 44, 110
 Funktionsbeschreibung 197
sequenzielle Datenübertragung 22
servent-Struktur 69, 158
Server 38, 92
 Task 87
 Verbindung annehmen (Beispiel) 42
 verbindungslos (Beispiel) 104
 verbindungsorientiert (Beispiel) 88
Service
 Anforderung 88
 Informationen 157
 Namen abfragen 157
 Namen umwandeln 69
Service-Nummer siehe Portnummer
Services abfragen 136
setsockopt() 77
 Anwendungsbeispiel 77
 Funktionsbeschreibung 161, 236
setzen Socket-Option 161, 236
shutdown()
 Funktionsbeschreibung 204
SO_BROADCAST 164
SO_DEBUG 165
SO_DISHALIAS 165
SO_ERROR 165
SO_KEEPAVIVE 164, 165
SO_LINGER 166
SO_OUTPUTBUFFER 166
SO_RCVBUF 166
SO_REUSEADDR 36, 166
SO_SNDBUF 166
SO_TCP_NODELAY 172
SO_TSTIPAD 166
SO_TYPE 167
SO_VHOSTANY 167
soc_clearerr()
 Funktionsbeschreibung 208
soc_close()
 Beispiel 88, 92, 96, 100, 104, 110
 Funktionsbeschreibung 206

- soc_eof()
 - Funktionsbeschreibung [208](#)
- soc_error()
 - Funktionsbeschreibung [208](#)
- soc_flush()
 - Funktionsbeschreibung [209](#)
- soc_getc()
 - Funktionsbeschreibung [210](#)
- soc_gets()
 - Funktionsbeschreibung [211](#)
- soc_ioctl() [74](#), [196](#)
 - Funktionsbeschreibung [212](#)
- soc_poll() [60](#)
 - Beispiel [60](#)
 - Funktionsbeschreibung [221](#)
- soc_putc()
 - Funktionsbeschreibung [224](#)
- soc_puts()
 - Funktionsbeschreibung [225](#)
- soc_read()
 - Beispiel [40](#)
 - Funktionsbeschreibung [226](#)
- soc_readv()
 - Funktionsbeschreibung [226](#)
- soc_wake()
 - Funktionsbeschreibung [228](#)
- soc_write() [40](#)
 - Beispiel [40](#)
 - Funktionsbeschreibung [229](#)
- soc_writenv()
 - Funktionsbeschreibung [229](#)
- SOCK_DGRAM [44](#), [64](#), [131](#)
 - siehe auch Datagramm-Socket
- SOCK_RAW [29](#), [162](#), [232](#), [233](#)
- SOCK_STREAM [131](#), [185](#), [210](#), [211](#), [224](#), [225](#), [229](#)
- sockaddr_in-Struktur [25](#), [31](#)
- sockaddr_in6-Struktur [26](#), [31](#)
- sockaddr_iso-Struktur [28](#), [31](#)
- Socket [226](#)
 - abhören [38](#), [49](#), [185](#)
 - Adresse [24](#)
 - Adressierung [24](#)
 - auf anstehende Verbindungen
 - überprüfen [185](#)
 - Ausnahmebedingung prüfen [194](#)
 - Datagramm- [23](#)
 - Definition [22](#)
 - erzeugen [29](#), [232](#)
 - ICMP-Protokoll
 - Lesebereitschaft prüfen [194](#)
 - Nachricht empfangen [187](#), [190](#), [226](#)
 - Nachricht senden [198](#), [200](#), [229](#)
 - Namen abfragen [159](#)
 - Namen zuordnen [31](#), [126](#)
 - nicht-blockierend [74](#), [188](#), [198](#)
 - Optionen [77](#), [161](#), [236](#)
 - Raw-Socket [85](#)
 - schließen [51](#), [206](#)
 - Schreibbereitschaft prüfen [194](#)
 - Steuerfunktionen [212](#)
 - Stream- [22](#), [210](#), [211](#), [224](#), [225](#)
 - verbindungslos [23](#), [29](#), [44](#), [64](#)
 - verbindungsorientiert [22](#), [29](#), [38](#), [49](#)
- Socket-Anwenderprogramm
 - siehe Anwenderprogramm
- Socket-Bibliothek [65](#)
- Socket-Deskriptor siehe Deskriptor
- Socket-Funktion [9](#)
 - accept() [38](#), [49](#), [74](#), [88](#), [92](#), [123](#)
 - bind() [31](#), [33](#), [88](#), [92](#), [100](#), [104](#), [126](#)
 - connect() [37](#), [45](#), [48](#), [74](#), [96](#), [100](#), [130](#)
 - freeaddrinfo() [133](#)
 - freehostent() [134](#)
 - für Adressumwandlung [65](#)
 - gai_strerror() [135](#)
 - getaddrinfo() [136](#)
 - getbcamhost() [92](#), [100](#), [141](#)
 - getdtablesize() [142](#)
 - gethostbyaddr() [66](#), [143](#)
 - gethostbyname() [66](#), [96](#), [110](#), [143](#)
 - gethostname() [145](#)
 - getipnodebyaddr() [66](#), [146](#)
 - getipnodebyname() [66](#), [146](#)
 - getnameinfo() [150](#)
 - getpeername() [153](#)
 - getprotobyname() [155](#)

- getservbyname() 69, 157
- getservbyport() 69, 157
- getsockname() 159
- getsockopt() 77, 161, 236
- Hilfsfunktionen 120, 128
- if_freenameindex() 176
- if_indextoname() 177
- if_nameindex() 178
- if_nametoindex() 179
- inet_addr() 180
- inet_lnaof() 180
- inet_makeaddr() 180
- inet_netof() 180
- inet_network() 180
- inet_ntoa() 180
- inet_ntop() 66, 183
- inet_pton() 66, 183
- listen() 38, 88, 92, 185
- recv() 40, 88, 92, 187
- recvfrom() 44, 104, 187
- recvmsg() 40, 49, 50, 190
- select() 55, 57, 59, 194, 228
- send() 40, 74, 96, 100, 197
- sendmsg() 40, 49, 50, 92, 200
- sendto() 44, 110, 197
- setsockopt() 77, 161, 236
- shutdown() 204
- soc_clearerr() 208
- soc_close() 88, 92, 96, 100, 104, 110, 206
- soc_eof() 208
- soc_error() 208
- soc_flush() 209
- soc_getc() 210
- soc_gets() 211
- soc_ioctl() 74, 196, 212
- soc_poll() 60, 221
- soc_putc() 224
- soc_puts() 225
- soc_read() 40, 226
- soc_readv() 226
- soc_wake() 228
- soc_write() 40, 229
- soc_writev() 229
- socket() 29, 88, 92, 96, 100, 104, 110, 232
- Übersicht 115
- Zusammenspiel 62
- Socket-Hostname 145
- Socket-Name siehe auch Name
- Socket-Option
 - abfragen 77, 161, 236
 - IP_AD_MEMBERSHIP 169
 - IP_DROP_MEMBERSHIP 169
 - IP_MULTICAST_IF 169, 171
 - IP_MULTICAST_LOOP 169, 171
 - IP_MULTICAST_TTL 169
 - IPV6_JOIN_GROUP 171
 - IPV6_LEAVE_GROUP 171
 - IPV6_MULTICAST_HOPS 171
 - setzen 77, 161, 236
 - SO_BROADCAST 164
 - SO_DEBUG 165
 - SO_DISHALIAS 165
 - SO_ERROR 165
 - SO_KEEPALIVE 164, 165
 - SO_LINGER 166
 - SO_OUTPUTBUFFER 166
 - SO_RCVBUF 166
 - SO_REUSEADDR 166
 - SO_SNDBUF 166
 - SO_TCP_NODELAY 172
 - SO_TESTIPAD 166
 - SO_TSTIPAD 166
 - SO_TYPE 167
 - TPOPT_CFRM_DATA 174
 - TPOPT_CONN_DATA 174
 - TPOPT_DISC_DATA 174
- Socket-Schnittstelle 9, 20
- Socket-Typ 22
 - abfragen 167
 - Datagramm-Socket siehe SOCK_DGRAM
 - Raw-Socket 23
 - Raw-Socket siehe SOCK_RAW
 - SOCK_DGRAM 23, 29, 44, 64, 131, 233
 - SOCK_RAW 29, 162, 232, 233
 - SOCK_STREAM 22, 29, 131, 185, 229, 233
 - Stream-Socket siehe SOCK_STREAM
- socket() 29
 - Beispiel 29, 30, 88, 92, 96, 100, 104, 110

- Funktionsbeschreibung [232](#)
- SOCKETS(BS2000) [9](#)
 - Anwenderprogramm produzieren [245](#)
 - installieren [241](#)
- SOCKETS(BS2000)-Subsysteme [241](#)
- Software-Voraussetzungen [245](#)
- SOL_SOCKET [77](#)
- Speicherfreigabe
 - addrinfo-Struktur [133](#)
 - hostent-Struktur [134](#)
- Starten
 - nslookup [242](#)
- Statusinformation abfragen [208](#)
- Steuerfunktion (für Sockets) [120](#), [212](#)
- Stream-Socket [131](#), [188](#), [210](#), [211](#), [224](#), [225](#), [229](#)
 - Eigenschaften [22](#), [233](#)
 - erzeugen [29](#), [30](#)
- Stream-Socket siehe auch SOCK_STREAM
- Streams-Verbindung
 - annehmen (Beispiel) [42](#)
 - veranlassen (Beispiel) [41](#)
- String
 - lesen aus Eingabepuffer [211](#)
 - schreiben in Ausgabepuffer [225](#)
- Struktur
 - addrinfo [133](#), [137](#)
 - cmsghdr [192](#), [202](#)
 - hostent [66](#), [67](#), [134](#), [144](#), [148](#)
 - iovec [191](#), [201](#), [227](#), [230](#)
 - linger [166](#)
 - msghdr [191](#), [201](#)
 - pollfd [222](#)
 - protoent [68](#)
 - servent [69](#), [158](#)
 - sockaddr_in [25](#), [31](#)
 - sockaddr_in6 [26](#), [31](#)
 - sockaddr_iso [28](#), [31](#)
- Subsysteme
 - siehe SOCKETS(BS2000)-Subsysteme
- symmetrisch, Protokoll [87](#)

T

- Task
 - Client- [87](#)

- Server- [87](#)
 - wecken [228](#)
- TCP [22](#)
- TCP_DELAY [172](#), [173](#)
- TCP_NODELAY [172](#)
- TCP/IP [9](#), [20](#)
- Testmakros für AF_INET6 [121](#)
- Timeout (E/A-Multiplexen) [56](#), [195](#)
- TPOPT_CFRM_DATA [63](#), [174](#), [202](#)
- TPOPT_CONN_DATA [63](#), [174](#), [202](#)
- TPOPT_DISC_DATA [63](#), [174](#), [202](#)
- TPOPT_REDI_CALL [175](#)
- Transportselektor TSEL [28](#), [48](#)
- TSEL [28](#), [48](#)
- typografische Gestaltungsmittel [13](#)

U

- überprüfen siehe abfragen
- Übersicht, Socket-Funktionen [115](#)
- übertragen
 - Daten [40](#), [44](#), [50](#)
 - Daten aus Ausgabepuffer [209](#)
- übertragen siehe auch senden/empfangen
- UDP [23](#), [44](#), [64](#)
- umwandeln
 - Adresse [65](#)
 - Adresse (Beispiel) [71](#)
 - Byte-Reihenfolge [70](#), [128](#)
 - Protokollname [68](#)
 - Rechnername [66](#)
 - Service-Name [69](#)

V

- Verbindung [115](#)
 - abbauen [51](#), [115](#)
 - abbauen (AF_INET) [51](#)
 - abbauen (AF_INET6) [51](#)
 - abbauen (AF_ISO) [54](#)
 - anfordern [37](#), [48](#), [88](#), [115](#), [130](#)
 - anfordern (Client-Beispiel) [41](#)
 - annehmen [38](#), [49](#), [115](#), [123](#)
 - annehmen (Server-Beispiel) [42](#)
 - anstehend siehe Verbindungsanforderung
 - asymmetrisch [37](#)

- aufbauen 37, 115
 - initiiieren siehe Verbindung anfordern
 - Verbindung beenden siehe Verbindung abbauen
 - Verbindungsabbau (abortive) 53, 54
 - Verbindungsabbau (graceful) 51
 - Verbindungsanforderung 38, 49, 115, 130
 - anstehend 185
 - senden 131
 - warten auf 185
 - Verbindungsdaten 190, 200
 - Verbindungsdatentyp
 - CFRM_DATA 63, 190
 - CONN_DATA 63, 190
 - DISC_DATA 63, 190
 - TPOPT_CFRM_DATA 63, 202
 - TPOPT_CONN_DATA 63, 202
 - TPOPT_DISC_DATA 63, 202
 - verbindungslos
 - Client (Beispiel) 110
 - Kommunikation 23, 44, 64
 - Kommunikation (Beispiele) 45
 - Server (Beispiel) 104
 - Socket 23, 29, 44, 64
 - verbindungsorientiert 22
 - Client (Beispiel) 96
 - Kommunikation 22, 37, 48, 50
 - Kommunikation (Beispiele) 41
 - Server (Beispiel) 88
 - Socket 22
 - Verzögerungsintervall 166
 - virtueller Host 78, 167
 - Voll-Duplex-Verbindung
 - abbauen 204
 - Voraussetzungen Software 245
- W**
- wecken, Task 228
 - Wildcard
 - Adresse 33
 - Portnummer 35
- Z**
- Zeichen
 - lesen aus Eingabepuffer 210
 - schreiben in Ausgabepuffer 224
 - Zeichenkette 181
 - Zeichenkette siehe auch String
 - zuordnen
 - Adresse/Namen 33, 38, 126
 - Zusammenspiel
 - Socket-Funktionen (quasi-
verbindungsorientiert) 64
 - Socket-Funktionen (verbindungslos) 64
 - Socket-Funktionen
(verbindungsorientiert) 62

