Einleitung

Die in diesem Handbuch beschriebene C-Programmierschnittstelle besteht aus über 500 Funktionen, Makros und externen Variablen. Dazu gehören alle im ANSI-Standard definierten und alle vom X/Open Portability Guide Issue 4 Version 2 - kurz **XPG4 Version 2** genannt - geforderten Funktionen. Zusätzlich wird die optionale Funktionsgruppe „Encryption“ des XPG4 unterstützt sowie zahlreiche Erweiterungen.

Diese C-Programmierschnittstelle ist Bestandteil der C-Laufzeitbibliothek (BS2000) V2.5. Die C-Laufzeitbibliothek ist Bestandteil des Common Runtime Environment **CRTE** V2.3A und wird ab der Betriebssystemversion BS2000/OSD V3.0 bzw. OSD-SVP V2.0 freigegeben. Das POSIX-Subsystem muss geladen sein, um die volle Funktionalität der in diesem Handbuch beschriebenen C-Bibliotheksfunktionen zu erhalten.

Die C-Bibliotheksfunktionen erlauben die komfortable Programmierung vieler Aufgaben, für die die Sprache C selbst keine höheren Sprachmittel vorsieht. Beispiele für solche Programmieraufgaben sind:

– Verarbeitung von Dateien (Öffnen, Schließen, Positionieren, Lesen, Schreiben etc.)

– Verarbeitung von einzelnen Zeichen oder Zeichenketten (Suchen, Ändern, Kopieren, Löschen etc.)

– Dynamische Speicherverwaltung (Speicherbereiche anlegen, freigeben etc.)

– Zugriff auf das Betriebssystem

– Einsatz mathematischer Funktionen

Alle Funktionen im Nachschlageteil „Funktionen und Variablen alphabetisch“, die nicht in der Überschrift als Erweiterungen gekennzeichnet sind (siehe nächster Abschnitt), verhalten sich konform zu den oben genannten Standards. Funktionalitätserweiterungen einzelner Funktionen oder bis zum Branding noch vorhandene Einschränkungen werden in der Beschreibung explizit gekennzeichnet.

Erweiterungen

Zusätzlich zu den erwähnten internationalen Standards unterstützt die C-Bibliothek die Funktionen der C-Laufzeitbibliothek (BS2000) bis V2.4 (siehe auch Handbuch „C-Bibliotheksfunktionen (BS2000)“ und zusätzliche Erweiterungen, die auf vielen UNIX-Systemen - z.B. auch unter SINIX - unterstützt werden. Die Erweiterungen der bisherigen C-Bibliothek (BS2000) sind in den Überschriften des Nachschlageteils mit *BS2000* gekennzeichnet. Die neu hinzugekommenen Erweiterungen sind in den Überschriften des Nachschlageteils mit *Erweiterung* gekennzeichnet. Diese explizite Kennzeichnung der Erweiterungen soll die Programmierung von portablen Programmen unterstützen.

Funktionen für Ein-/Ausgabe, Signalbehandlung und Lokalität unterstützen Erweiterungen, die mit den C-Laufzeitbibliothek-Vorgängern kompatibel sind. Insbesondere ist der Zugriff sowohl auf das Datenverwaltungssystem von BS2000/OSD (DVS) als auch auf das XPG4 Version 2-konforme POSIX-Dateisystem möglich (siehe Handbuch „POSIX Grundlagen (BS2000/OSD)“).

Als zusätzliche Erweiterungen stehen zur Verfügung:

– 64-Bit-Funktionen zur NFS V3.0-Unterstützung
– Funktionen zur POSIX-Thread-Unterstützung in der C-Laufzeit-Bibliothek

Einschränkungen

In dieser Version der C-Laufzeitbibliothek gibt es folgende Einschränkung gegenüber XPG4 Version 2:

Wenn die Umgebung (externe Variable `environ`) durch `putenv()` neu eingerichtet wird, ist als Dateisystem das DVS voreingestellt. Der Anwender muss `PROGRAM-ENVIRONMENT` explizit auf `SHELL` setzen (siehe Abschnitt „Umfang der unterstützten C-Bibliothek“ auf Seite 18 und die C- und C++-Benutzerhandbücher).

Weitere Detail-Einschränkungen sind in den Funktionsbeschreibungen ausgewiesen.

Zielgruppe des Handbuchs

Das Handbuch richtet sich an C-Programmierer, die folgende Aufgaben erledigen wollen:

– C-Programme von UNIX oder SINIX-Plattformen ins POSIX-Subsystem portieren

– C-Programme für XPG4 Version 2-konforme Umgebungen (POSIX-Subsystem) im BS2000/OSD schreiben

– C-Programme erstellen, die sowohl auf ein XPG4 Version 2-konformes Dateisystem als auch auf das DVS zugreifen können

Voraussetzungen für die Arbeit mit diesem Handbuch sind Kenntnisse der Programmiersprache C und der Betriebssysteme POSIX-Subsystem und BS2000/OSD.

Konzept des Handbuchs

Das Handbuch gliedert sich in einen konzeptionellen Teil, einen Nachschlageteil und einen Verzeichnisteil.

Der konzeptionelle Teil umfasst anschließend an die Einleitung folgende Kapitel:

– eine allgemeine Beschreibung der wichtigsten Eigenschaften der C-Bibliothek und grundsätzliche Besonderheiten über die Interaktionen zwischen den Betriebssystemen

– eine thematische Gliederung der im Nachschlageteil aufgeführten Funktionen, Makros und externen Variablen

Der Nachschlageteil enthält die Beschreibungen der einzelnen, alphabetisch sortierten Funktionen, Makros und Variablen und die Beschreibung der ebenfalls alphabetisch aufgeführten Include-Dateien.

Der Verzeichnisteil enthält neben dem Stichwortverzeichnis ein Fachwort- und ein Literaturverzeichnis.

Dokumentation des CRTE und des C-Entwicklungssystems

In den C- und C++-Benutzerhandbüchern ist ausführlich dargestellt, wie beim Übersetzen, beim Binden und beim Ablauf eines C/C++-Programms auf die CRTE-Bibliothek zugegriffen wird.

Im Benutzerhandbuch „CRTE (BS2000/OSD)“ finden Sie allgemeine Hinweise und Bindebeispiele für die gemeinsame Laufzeitumgebung von C, C++ und COBOL85/COBOL2000.

Konzept der POSIX-Dokumentation

Für das Kennenlernen und Arbeiten mit dem POSIX-Subsystem im BS2000/OSD steht Ihnen folgende Dokumentation zur Verfügung:

- Ein Überblick über die Strategie und die Ziele von POSIX im BS2000/OSD in der Broschüre „POSIX im BS2000/OSD“.
- Eine Einführung in das Arbeiten mit dem POSIX-Subsystem im Handbuch „POSIX - Grundlagen für Anwender und Systemverwalter“. Darüber hinaus werden die Verwaltungsaufgaben beschrieben, die im Zusammenhang mit dem POSIX-Subsystem anfallen. Des Weiteren erfahren Sie, mit welchen BS2000/OSD-Softwareprodukten Sie das POSIX-Subsystem nutzen können.
- Eine vollständige Beschreibung der POSIX-Kommandos, mit denen Sie in der POSIX-Shell arbeiten können, finden Sie im Handbuch „POSIX-Kommandos“.
- Das Handbuch „POSIX-Kommandos des C- und C++-Compilers“ liefert eine Einführung in die C-/C++-Programmentwicklung in POSIX-Shell-Umgebung, beschreibt das Übersetzen und Binden von C- und C++-Programmen mit den POSIX-Kommandos cc, c89 und CC und zeigt, wie Sie den globalen C- und C++-Listengenerator mit dem POSIX-Kommando ccxref steuern.
- Das Handbuch „POSIX V1.1A Sockets/XTI für POSIX“ wendet sich an C- und C++-Programmierer, die mit SOCKETS- bzw. XTI-Funktionen Kommunikationsanwendungen auf der Basis der POSIX-Schnittstelle entwickeln.
- „NFS V3.0 / NFS V1.2C Network File System“

POSIX-Dokumentation im BS2000/OSD-Umfeld

Im BS2000/OSD werden Softwareprodukte funktionell erweitert, so dass Sie auch mit diesen Produkten die POSIX-Funktionalität nutzen können.

Eine Reihe von Dienstprogrammen ermöglichen den Zugriff auf das POSIX-Dateisystem. So können Sie z.B. mit EDT V16.5 Dateien des POSIX-Dateisystems bearbeiten.

Durch die Erweiterung des CRTE (Common Runtime Environment) gemäß dem XPG4 Version 2-Standard können Sie mit den C-Bibliotheksfunktionen V2.5 unabhängig vom ausführenden Betriebssystem portable C-Programme schreiben.

Als Grundlage für den Zugriff auf die POSIX-Funktionalität aus anderen Softwareprodukten wird die Kenntnis des Handbuchs „POSIX - Grundlagen für Anwender und Systemverwalter“ vorausgesetzt.

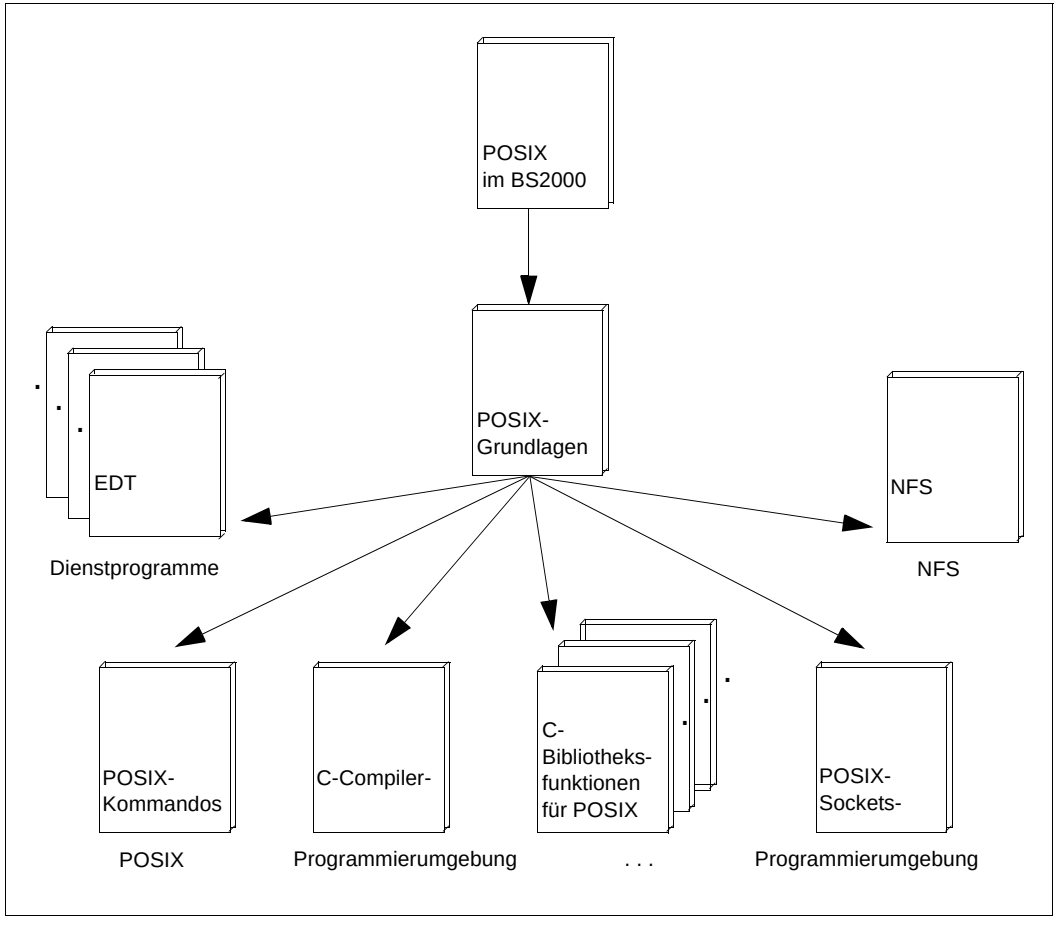


Bild 1: POSIX-Dokumentation im BS2000/OSD-Umfeld

Darstellungsmittel

Die Darstellung der Anweisungsformate und Benutzereingaben in diesem Handbuch richtet sich nach folgenden Regeln:

`dicktengleich`	Kennzeichnung von Namen, die zum Sprachumfang von C und der C-Bibliothek gehören. Kennzeichnung der Querverweise unter „Fachwörter“ auf andere Fachwörter. Darstellung von Beispielen, Ein- und Ausgaben.
`GROSS`	Kennzeichnung von nicht implementierungsabhängigen symbolischen Konstanten (z.B. `HUGE_VAL`), symbolischen Namen von Signalen (z.B. `SIGABRT`) und Fehlernummern (z.B. `EDOM`).
`{GROSS}`	Kennzeichnung von implementierungsabhängigen symbolischen Konstanten, die in der Include-Datei `limits.h` definiert sind (z.B. `{INT_MAX}`).
kursiv	Kennzeichnung von beispielhaften Namen für Parameter in Benutzereingaben.
`[ ]`	Kennzeichnung von syntaktischen Einheiten, die Sie verwenden können, aber nicht müssen. Die Klammern selbst dürfen Sie nicht angeben.
...	In Syntaxangaben Kennzeichnung für die Wiederholung der vorhergehenden syntaktischen Einheit. In Beispielen Kennzeichnung für eine Auslassung von Programmcode.
␣	Dieses Zeichen wird verwendet, um auf ein zwingendes Leerzeichen explizit hinzuweisen und damit Missverständnisse zu vermeiden. Im Allgemeinen gilt ein Zwischenraum als Leerzeichen.
|	Trennzeichen für alternative Angaben. Sie müssen sich für eine der nebenstehenden Angaben entscheiden. Den senkrechten Strich selbst dürfen Sie nicht angeben.
Taste	Tastendarstellung.
Taste1 + Taste2	Darstellung von Tasten, die gleichzeitig gedrückt werden müssen.
Taste1 Taste2	Darstellung von Tasten, die hintereinander gedrückt werden müssen.

Strukturmittel, die im Nachschlageteil eingesetzt werden, um die Funktionsbeschreibungen zu gliedern, finden Sie zu Beginn des Kapitels „Funktionen und Variablen alphabetisch (a - m)“ auf Seite 165 beschrieben.

Änderungen gegenüber dem Vorgängerhandbuch

Gegenüber dem Vorgängerhandbuch wurde die Beschreibung um die folgenden Gruppen von Funktionen erweitert:

- C-Bibliotheksfunktionen (BS2000) aus den Versionen V2.3B und V2.4A der C-Laufzeitbibliothek (Einzelheiten siehe nachfolgende Abschnitte).
- 64-Bit-Funktionen zur Unterstützung von großen Dateien (> 2GB)
- Funktionen zur Unterstützung von POSIX-Threads

Neuerungen der C-Laufzeitbibliothek V2.3B

Die folgenden Änderungen sind auf wesentliche Neuerungen der C-Laufzeitbibliothek zurückzuführen, die bereits in der Version 2.3B realisiert wurden:

- Unterstützung der neuen Datentypen `wint_t`, `wctrans_t` und `mbstate_t`
- neue Include-Datei `<wctype.h>` mit Definitionen für Langzeichen und Multibyte-Zeichen
- neue Include-Datei `<iso646.h>` mit der Definition von C-Schlüsselwortoperatoren
- Unterstützung aller Funktionen aus dem Amendment 1 zum Standard ISO/IEC 9899:1990

  Die Beschreibung der nachfolgend aufgelisteten Funktionen wurde neu in das Handbuch aufgenommen.

  In **<wchar.h>** deklarierte Funktionen:

btowc	swprintf	wctob
fwide	vwfprintf	wmemchr
fwprintf	vswprintf	wmemcmp
fwscanf	vwprintf	wmemcpy
mbrlen	wcrtomb	wmemmove
mbrtowc	wcsrtombs	wmemset
mbsinit	wcstoll	wprintf
mbsrtowcs	wcstoul	wscanf
swscanf	wcsstr	

  In **<wctype.h>** deklarierte Funktionen::

  towctrans
  wctrans

Neuerungen der C-Laufzeitbibliothek V2.4A

Die folgenden Änderungen sind auf wesentliche Neuerungen der C-Laufzeitbibliothek zurückzuführen, die bereits in der Version 2.4A realisiert wurden:

- Die der IO_CONVERSION zugrundeliegenden Konvertierungstabellen wurden kompatibel auf einen 8-Bit-Code erweitert.
- Zwei neue Lokalitäten wurden zur Unterstützung des Euro aufgenommen:
  De.EDF04F und
  De.EDF04F@euro.

  Diese beiden Lokalitäten unterscheiden sich nur in der Kategorie LC_MONETARY.
  Beiden Lokalitäten liegt die neue 8-Bit-Code-Tabelle zugrunde.
  Monetärer Dezimalpunkt (mon_dezimal_point) ist in beiden Lokalitäten das Zeichen "," (Komma).
- Es wurde ein zusätzlicher Bindeschalter (Bibliothek SYSLNK.CRTE.TIME bzw. SRULNK.CRTE.TIME) aufgenommen, der die Nutzung der POSIX-Zeitfunktionen gestattet, ohne dass sich das Programm mit einem eventuell vorgeladenen POSIX-Subsystem konnektiert.
  Für die generelle Einstellung der Zeitzone der POSIX-Zeitfunktionen steht Ihnen die Prozedur ICXTZ in der Bibliothek SINPRC.CRTE.023 zur Verfügung.
- Es wurden folgende Funktionen zur Unterstützung des Datentyps long long integer aufgenommen (da es sich bei den Rundungsfunktionen um eine Funktionsgruppe handelte, wurden auch die Funktionen für die anderen Datentypen aufgenommen):
  - Rundungsfunktionen

```
double rint                 float round
float rintf                 float roundf
long double rintl           long double roundl
long int lrint              long int lround
long int lrintf             long int lroundf
long int lrintl             long int lroundl
long long int llrint        long long int llround
long long int llrintf       long long int llroundf
long long int llrintl       long long int llroundl
```

  - Absolutbetrag und Division

```
long long int llabs         lldiv_t lldiv
```

– Konvertierungsfunktionen

```
long long int atoll          long long int wcstoll
long long int strtoll        long long int wcstoull
long long int strtoull
```

- Erweiterung der Funktionen zur formatierten Ein-/Ausgabe um die Formatangabe `ll` für Werte vom Typ `long long int`.
- Betroffen sind die Ausgabe-Funktionen `printf`, `fprintf`, `sprintf`, `vprintf`, `vfprintf` und `vsprintf` und die entsprechenden Langzeichenfunktionen `wprintf`, `fwprintf`, `swprintf`, `vwprintf`, `vfwprintf` und `vswprintf` sowie die Eingabe-Funktionen `scanf`, `fscanf`, `sscanf`, `vscanf`, `vfscanf` und `vsscanf` und die entsprechenden Langzeichen-Funktionen w`scanf`, `fwscanf`, `swscanf`, `vwscanf`, `vfwscanf` und `vswscanf`.
- Das Präprozessor-Define `XPG_IV_SPECIAL` wurde durch `_XOPEN_SOURCE_EXTENDED` ersetzt.

Änderungen gegenüber dem C-Bibliothekshandbuch V2.2A

Die folgenden Änderungen sind auf wesentliche Neuerungen der C-Laufzeitbibliothek zurückzuführen, die bereits in der Version 2.3B realisiert wurden:

– neue Include-Dateien <`wchar.h`> und <`wctype.h`> mit Definitionen für Langzeichen und Multibyte-Zeichen.

– neue Include-Datei <`iso646.h`> mit der Definition von C-Schlüsselwortoperatoren (siehe Seite 13).

Die C-Programmierschnittstelle

In diesem Kapitel wird beschrieben, welche Systemvoraussetzungen die C-Programmierschnittstelle benötigt, welche Bestandteile sie hat und was bei ihrem Einsatz zu beachten ist.

Systemvoraussetzungen

Die folgende Tabelle listet die Softwareprodukte auf, die für die Unterstützung der vollständigen Funktionalität der C-Bibliothek notwendig sind, wie sie mit CRTE V2.3A zur Verfügung gestellt wird und in diesem Handbuch beschrieben ist.

Produkt	**Relevante Bestandteile**
BS2000/OSD-BC V3.0	– Betriebssystem – Include-Dateien für POSIX-Funktionen
C/C++ V3.0	C- und C++-Compiler für POSIX-Subsystem und BS2000/OSD
CRTE V2.3A	– Include-Dateien für BS2000-Funktionen – Laufzeitmodule der C-Bibliotheksfunktionen
POSIX-BC V3.0	– POSIX-Dateisystem – Basis-Shell – POSIX-HEADER V1.3
SDF-P	Variablenstruktur `SYSPOSIX` für die Initialisierung der Laufzeitumgebung

Die Kommandos des **POSIX-Subsystems**, das die Produkte POSIX-BC und POSIX-SH umfasst, sind im Handbuch „POSIX Kommandos (BS2000/OSD)“ beschrieben. Die Kommandos des Produktes POSIX-SH erhöhen den Komfort der Arbeit in der POSIX-Shell. Sie sind jedoch nicht Systemvoraussetzung für das Übersetzen, Binden und Starten von C-Programmen.

Bestandteile der C-Bibliothek

Die Programmschnittstelle der C-Laufzeitbibiliothek unterstützt über 500 vordefinierte Funktionen (siehe auch Tabelle auf Seite 19ff.). Diese Funktionen liegen entweder als Quellprogrammteile (Makros) oder als bereits übersetzte Programmteile (Module) vor. Die Deklarationen der Funktionen, die Definitionen von Konstanten, Datentypen und Makros sowie die Funktionsmakros selbst stehen in den Include-Dateien.

Include-Dateien

Die Include-Dateien für die C-Programmierschnittstelle werden mit zwei verschiedenen Produkten ausgeliefert:

Include-Dateien für POSIX-Funktionen werden als Komponente POSIX HEADER mit dem Produkt POSIX-BC ausgeliefert, Include-Dateien für BS2000-Funktionen mit CRTE V2.3A (siehe auch Tabelle auf Seite 11).

Sie werden bei der Übersetzung auf Grund der Präprozessoranweisung `#include` in das Programm kopiert. Wie dies geschieht, ist ausführlich in den C- und C++-Benutzerhandbüchern dargestellt.

In einer Include-Datei sind deklariert bzw. definiert:

– Funktionen bzw. entsprechende Makros

– externe Variablen

– symbolische Konstanten und Datentypen

Die XPG4 Version 2-konformen Include-Dateien sind im Kapitel „Include-Dateien“ beschrieben.

Für den Aufruf von C-Bibliotheksfunktionen aus C++-Quellen enthalten die Include-Dateien für alle Funktionen und Daten `extern "C"`-Deklarationen.

Im POSIX-Subsystem sind die Include-Dateien in den Standard-Dateiverzeichnissen `/usr/include` und `/usr/include/sys` abgelegt.

Im BS2000 stehen die Include-Dateien als PLAM-Bibliothekselemente vom Typ S in den Bibliotheken `$.SYSLIB.CRTE` (für BS2000-Funktionen) und `$.SYSLIB.POSIX-HEADER` (für POSIX-Funktionen).

`Include`-Anweisungen, in denen die Namen der Include-Elemente Schrägstriche (/) für Verzeichnisse enthalten, werden vom Compiler auch im Falle von PLAM-Bibliothekselementen akzeptiert. Jeder Schrägstrich in Namen von benutzereigenen und Standard-Include-Elementen wird intern zur Suche in PLAM-Bibliotheken in einen Punkt (.) umgewandelt.

In Quellprogrammen, die z.B. aus dem POSIX oder SINIX ins BS2000 portiert werden, müssen die Schrägstriche nicht in Punkte umgewandelt werden.

In Quellprogrammen, die aus der BS2000-Umgebung in das POSIX-Filesystem kopiert werden, müssen die Punkte nicht in Schrägstriche umgewandelt werden. Dies gilt jedoch nur für die Standard-Include-Elemente und nicht für benutzerdefinierte Include-Dateien.

Include-Datei iso646.h

Die Include-Datei `iso646.h` enthält die folgenden 11 Makros, die zu den jeweils dahinterstehenden Schreibweisen expandiert werden und damit alternative Schreibweisen für die Operatoren darstellen:

and	&&	compl	~	or_eq	|=
and_eq	&=	not	!	xor	^
bitand	&	not_eq	!=	xor_eq	^=
bitor	|	or	| |		

Funktionen und Makros

Die meisten Bibliotheksfunktionen sind als C-Funktionen, manche als Makros realisiert. Einige Bibliotheksfunktionen sind sowohl als Funktion als auch als Makro realisiert.

Gibt es eine Bibliotheksfunktion in beiden Varianten, wird für den Aufruf standardmäßig die Makrovariante generiert. Ein Funktionsaufruf wird dann generiert, wenn der Name in Klammern `()` eingeschlossen oder mit der `#undef`-Anweisung aufgehoben wird. Welche Ausführung Sie jeweils wählen, hängt davon ab, ob und welche Aspekte (Performance, Programmgröße, Einschränkungen) jeweils für das Programm relevant sind.

Eine **Funktion** ist ein nur einmal vorhandener, übersetzter Programmteil (Modul) und wird zum Ablaufzeitpunkt wie ein externes Unterprogramm behandelt. Für jeden Funktionsaufruf ist bei Programmablauf ein Organisationsaufwand notwendig; z.B. das Verwalten der lokalen, dynamischen Daten einer Funktion im Laufzeitstack, Sichern der Registerinhalte, Rücksprungadressen etc.

Gesteuert über die Compileroption `OPTIMIZATION` können einige Bibliotheksfunktionen inline generiert werden. In diesem Fall wird der Funktionscode direkt in die Aufrufstelle eingesetzt, und es entfallen die o.g. Verwaltungsaktivitäten.

Derzeit können folgende Funktionen inline generiert werden: `strcpy()`, `strcmp()`, `strlen()`, `strcat()`, `memcpy()`, `memcmp()`, `memset()`, `abs()`, `fabs()`, `labs()` (siehe auch C- und C++-Benutzerhandbücher).

Ein **Makro** ist ein mit der `#define`-Anweisung definierter Quellprogrammteil. Mit jedem Makro-Aufruf wird bei der Übersetzung der Makro-Name im Programm durch den Inhalt des aufgerufenen Makros ersetzt.

Die Benutzung eines Makros kann zu einer besseren Performance bei Programmablauf führen, da die Verwaltungsaktivitäten des Laufzeitsystems (siehe Funktion) entfallen. Andererseits wird das übersetzte Programm durch die Makroauflösungen größer.

Bei der Benutzung eines Makros ist außerdem auf Folgendes zu achten:

– Makronamen können anderen Funktionen nicht als Argument übergeben werden, wenn diese einen Zeiger auf eine Funktion als Argument verlangen.

– Inkrement-/Dekrement- oder zusammengesetzte Zuweisungsoperatoren für Makro-Argumente können zu unerwünschten Nebeneffekten führen.

– Die Include-Datei, die die Makrodefinition enthält, muss auf jeden Fall in das Programm eingefügt werden.

Unterstützung von NFS V3.0 durch 64-Bit-Funktionen

Mit NFS V3.0 können Dateisysteme bearbeitet werden, die Dateien mit einer Größe von mehr als 2 Gigabyte (GB) enthalten. Dazu wurden zahlreiche I/O-Funktionen um eine neue Variante ergänzt, die das Suffix 64 enthält. So wurde z.B. dem 32-Bit-Datentyp `off_t` der Datentyp `off64_t` hinzugefügt.

Die Übersetzungsumgebung stellt alle diese expliziten 64-Bit-Funktionen und Typen zusätzlich zu den 32-Bit-Funktionen und Typen zur Verfügung. Damit kann ein Programm, je nach Bedarf, beide Schnittstellen verwenden. Um große Dateien (> 2 GB) zu handhaben, muss eine Anwendung die 64-Bit-Schnittstelle benutzen.

Die Verwendung dieser Funktionen macht nur im Zusammenhang mit POSIX Sinn, da große Dateisysteme bzw. große Dateien nur über NFS erreicht werden können.

Da die meisten Namen der 64-Bit-Funktionen auf 8 Zeichen gekürzt CRTE-weit nicht mehr eindeutig sind, müssen Sourcen, die 64-Bit-Funktionen nutzen wollen, als LLMs generiert werden.

Die neuen Funktionen können nur genutzt werden, wenn vorher das Define `_LARGEFILE64_SOURCE=1` gesetzt wird (Prototyp-Generierung und weitere Defines). Außerdem muss in der POSIX-Parameterdatei für den Parameter `FILESIZE` der Wert `UNLIMITED64` gesetzt sein.

POSIX-Thread-Unterstützung in der C-Laufzeitbibliothek

CRTE V2.3A bietet Unterstützung von POSIX-Threads durch neue Header-Dateien und Funktionen. In diesem Handbuch wird die neue Funktionalität beschrieben, die sich aus der POSIX-Thread-Unterstützung ergibt.

Langzeichen und Multibyte-Zeichen

Langzeichen und Multibyte-Zeichen wurden definiert, um das ursprüngliche „Zeichen“-Konzept der Computersprachen zu erweitern, das die Zuordnung eines Zeichens zu einem Byte Speicherplatz vorsah. Diese Zuordnung reichte jedoch für Sprachen wie zum Beispiel Japanisch nicht aus, da die Darstellung eines Zeichens in diesen Sprachen mehr als ein Byte Speicherplatz erfordert. Aus diesem Grunde wurde das Zeichen-Konzept um Multibyte-Zeichen und Langzeichen erweitert. Multibyte-Zeichen stellen Zeichen des erweiterten Zeichensatzes in zwei, drei oder mehr Bytes dar.
Multibyte-Zeichenketten können „Shift-Sequenzen“ enthalten, die die Bedeutung der nachfolgenden Multibyte-Codes verändern. Shift-Sequenzen können zum Beispiel umschalten zwischen verschiedenen Interpretationsmodi: Die ein-byte Shift-Sequenz `0200` kann festlegen, dass nachfolgende Byte-Paare als japanische Zeichen interpretiert werden, bzw. die Shift-Sequenz `0201`, dass nachfolgende Byte-Paare als Zeichen des ISO-Latin-1-Zeichensatzes interpretiert werden.

Programmier-Modell

Programme, die mit Multibyte-Zeichen arbeiten, können mit Hilfe der Amendment 1-Funktionen ebenso leicht realisiert werden, wie Programme, die das traditionelle Zeichenkonzept verwenden.

Dabei werden Multibyte-Zeichen oder -Zeichenketten, die aus einer externen Datei eingelesen werden, intern in ein `wchar_t`-Objekt oder ein Feld vom Typ `wchar_t` eingelesen. Bei dieser Leseoperation werden die Multibyte-Zeichen in das entsprechende Langzeichen konvertiert.
Die `wchar_t`-Objekte können anschließend durch `isw`*xxx*-Funktionen, `wcstod`, `wmemcmp` usw. bearbeitet werden.
Die resultierenden `wchar_t`-Objekte werden dann durch Ausgabefunktionen wie `putwchar`, `fputws` usw. ausgegeben.
Bei der Ausgabe werden die Langzeichen in die entsprechenden Multibyte-Zeichen konvertiert.

Hinweise zu Langzeichen

Ein Langzeichen ist definiert als der Codewert eines Objekts vom Typ `wchar_t` (binär kodierter Integerwert), der einem Element des erweiterten Character-Sets entspricht.
Das Null-Langzeichen hat den Codewert Null.

Das Dateiende-Kriterium in Langzeichen-Dateien ist `WEOF`.

Langzeichenkonstanten werden in der Form L“*langzeichenkette*“ geschrieben.

Hinweise zu dieser Implementierung

In dieser Version der C-Laufzeitbibliothek werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t`, der intern auf den Typ `long` abgebildet wird.
Multibyte-Zeichen haben entsprechend auch immer die Länge 1 Byte.

Zeitfunktionen

Die Zeitfunktionen, die beim Einsatz der C-Bibliotheksfunktionen ohne POSIX verwendet werden, d.h. wenn der POSIX-Bindeschalter nicht eingebunden ist, unterscheiden sich in drei wesentlichen Punkten von den Zeitfunktionen, die im POSIX/UNIX-Bereich verwendet werden:

– Stichtag für die Umrechnung in Sekunden (Epoche) ist der 1.1.1950 00:00:00 (bei POSIX der 1.1.1970).

– Zeitangaben sind streng lokalzeitbezogen; bei der Umstellung auf Sommer- bzw. Winterzeit „springen“ die Zeitangaben. Speziell durch den Rücksprung bei der Winterzeitumstellung können negative Zahlen und Zeitdifferenzen auftreten, die bei der Weiterverarbeitung zu unerwarteten Resultaten führen.

– Die Funktion `gmtime` ist wie `localtime` implementiert.

Die Verwendung des Stichtags 1.1.1950 hat zur Folge, dass die entsprechenden Zeitfunktionen ab dem Jahr 2018 nicht mehr funktionieren. Das Jahr 2018 stellt somit für C-Anwendungen im BS2000 ein größeres Problem dar als der Jahrtausendwechsel.
Bei Verwendung der POSIX-Zeitfunktionen tritt das Problem erst zwanzig Jahre später auf; zudem ist die Problematik in den entsprechenden Fachkreisen bekannt, an ihrer Lösung wird gearbeitet.

Aus diesen Gründen wird den Anwendern empfohlen, ihre Programme auf die POSIX-Zeitfunktionen umzustellen.

Die POSIX-Zeitfunktionen werden automatisch verwendet, wenn der POSIX-Bindeschalter eingebunden wird; für ihre Verwendung muss kein POSIX-Subsystem vorhanden sein.
Ist allerdings das POSIX-Subsystem vorgeladen, bewirkt das Einbinden des POSIX-Bindeschalters, dass sich das Programm mit dem POSIX-Subsystem konnektiert.

Wenn Sie den POSIX-Bindeschalter einbinden, werden zudem die POSIX-Funktionen verwendet, wie sie im Handbuch „C-Bibliotheksfunktionen für POSIX“ beschrieben sind, z. B. auch bei Ein-/Ausgabefunktionen; insbesondere werden Dateinamen, die nicht explizit als BS2000-Dateinamen gekennzeichnet sind, als POSIX-UFS-Dateinamen interpretiert.

Wenn Ihr Programm nur die POSIX-Zeitfunktionen benutzen soll, müssen Sie den TIME-Bindeschalter verwenden.

Zum Inkludieren steht Ihnen dazu die Bibliothek

– SYSLNK.CRTE.TIME bzw.

- SRULNK.CRTE.TIME

zur Verfügung.

Wenn Sie den TIME-Bindeschalter nicht verwenden, verhalten sich alle bestehenden Programme und Prozeduren wie bisher.

Einstellen der Zeitzone für POSIX-Zeitfunktionen

Die POSIX-Zeitfunktionen werten für die Bestimmung der Zeitzone die Variable TZ aus.

Sie können die Zeitzone vor dem Programmstart über die SYSPOSIX-Variable setzen.
Ist die Variable beim Programmstart nicht gesetzt, dann initialisiert die C-Laufzeitbibliothek die Variable mit der für Deutschland gültigen Zeitzone, indem sie TZ auf den Wert MET-1DST,M3.5.0/02:00:00,M10.5.0/03:00:00 setzt.

Wollen Sie für die Installation generell eine andere als die deutsche Zeitzone einstellen, bietet Ihnen CRTE die Prozedur ICXTZ in der Bibliothek SINPRC.CRTE.023 an:

ICXTZ,(TZ='*zeitzonenangaben*')

Umfang der unterstützten C-Bibliothek

Die folgende Tabelle gibt eine Übersicht über die unterstützten C-Bibliotheksfunktionen.

Legende

In der Spalte „XPG5“ bedeuten:

x Funktion, die von XPG5 gefordert wird, die nur mit POSIX-BC ablauffähig und portabel in Hinblick auf XPG5-konforme Systeme ist.

xx Funktion, die mit derselben Funktionalität schon in der bisherigen BS2000-Bibliothek vorhanden war.

d Funktion, die zusätzlich außer POSIX-Dateien auch BS2000-Dateien bearbeiten kann.

a Funktion, die zusätzlich um die Funktionalität der bisherigen C-(BS2000)-Bibliotheksfunktion erweitert ist.

y Funktion, die von XPG5 gefordert wird und die portabel in Hinblick auf XPG5-konforme Systeme ist. Funktion ist auch ohne POSIX-BC ablauffähig.

In der Spalte „XPG4 Version 2“ bedeuten:

x Funktion, die von XPG4 Version 2 gefordert wird, die nur mit POSIX-BC ablauffähig und portabel in Hinblick auf XPG4 Version 2-konforme Systeme ist.

xx Funktion, die mit derselben Funktionalität schon in der bisherigen BS2000-Bibliothek vorhanden war.

d Funktion, die zusätzlich außer POSIX-Dateien auch BS2000-Dateien bearbeiten kann.

a Funktion, die zusätzlich um die Funktionalität der bisherigen C-(BS2000)-Bibliotheksfunktion erweitert ist.

y Funktion, die von XPG4 Version 2 gefordert wird und die portabel in Hinblick auf XPG4 Version 2-konforme Systeme ist. Funktion ist auch ohne POSIX-BC ablauffähig.

x in der Spalte „*Erweiterung*“ oder „*BS2000*“ bedeutet:
Erweiterung, die nur mit POSIX-BC ablauffähig ist (*Erweiterung*) oder die es schon in der bisherigen C-Bibliothek (*BS2000*) gab.

y in Spalte „XPG4 Version 2“ bedeutet:
Funktionen, die von XPG4 Version 2 gefordert wird und die portabel im Hinblick auf XPG4 Version 2-konforme Systeme ist. Die Funktion ist auch ohne POSIX-BC ablauffähig.

x in Spalte „ANSI“ bedeutet

Funktionen, die weder in XPG4 Version 2 enthalten sind, noch eine BS2000-Erweiterung darstellen, sondern gemäß dem ANSI-C-Standard (__STDC_VERSION 199901L) implementiert wurden.

Funktion	XPG5	XPG4 Version 2	Erweiterungen		ANSI
			Erweiterung	*BS2000*	
a641()	y	y			
abort()	xa	xa			
abs()	xx	xx			
access()	x	x			
acos()	xx	xx			
acosh()	x	x			
advance()	x	x			
alarm()	xa	xa			
altzone			x		
ascii_to_ebcdic()			y		
asctime()	xx	xx			
asctime_r()	y				
asin()	xx	xx			
asinh()	x	x			
assert()	xx	xx			
atan()	xx	xx			
atan2()	xx	xx			
atanh()	x	x			
atexit()	xx	xx			
atof()	xx	xx			
atoi()	xx	xx			
atol()	xx	xx			
atoll()					x
basename()	x	x			
bcmp()	x	x			
bcopy()	x	x			
brk()	x	x			
bs2exit()				x	
bs2fstat()				x	

Funktion	XPG5	XPG4 Version 2	Erweiterungen		ANSI
			Erweiterung	*BS2000*	
bs2system()			x	(system())	
bsd_signal()	x	x			
bsearch()	xx	xx			
btowc()	y				
bzero()	x	x			
cabs()				x	
calloc()	xx	xx			
catclose()	x	x			
catgets()	x	x			
catopen()	x	x			
cbrt()	x	x			
cdisco()				x	
ceil()	xx	xx			
ceilf()					x
ceill()					x
cenaco()				x	
cfgetispeed()	x	x			
cfgetospeed()	x	x			
cfsetispeed()	x	x			
cfsetospeed()	x	x			
chdir()	x	x			
chmod()	x	x			
chown()	x	x			
chroot()	x	x			
clearerr()	xxd	xxd			
clock()	xa	xa			
close()	xxd	xxd			
closedir()	x	x			
closelog()	x	x			
compile()	x	x			
confstr()	x	x			
cos()	x	xx			

Funktion	XPG5	XPG4 Version 2	Erweiterungen		ANSI
			Erweiterung	*BS2000*	
cosh()	xx	xx			
cputime()				x	
creat()	xxd	xxd			
crypt()	x	x			
cstxit()				x	
ctermid()	x	x			
ctime()	xa	xa			
ctime_r()	y				
cuserid()	x	x			
__DATE__		xx			
daylight	x	x			
dbm_clearerr()	x	x			
dbm_close()	x	x			
dbm_delete()	x	x			
dbm_error()	x	x			
dbm_fetch()	x	x			
dbm_firstkey()	x	x			
dbm_nextkey()	x	x			
dbm_open()	x	x			
dbm_store()	x	x			
difftime()	xx	xx			
dirname()	x	x			
div()	xx	xx			
drand48()	x	x			
dup()	x	x			
dup2()	x	x			
ebcdic_to_ascii()			y		
ecvt()	xx	xx			
_edt()				x	
encrypt()	x	x			
endgrent()	x	x			
endpwent()	x	x			

Funktion	XPG5	XPG4 Version 2	Erweiterungen		ANSI
			Erweiterung	*BS2000*	
endutxent()	x	x			
environ	x	x			
erand48()	x	x			
erf()	xx	xx			
erfc()	xx	xx			
errno	xx	xx			
execl()	x	x			
execle()	x	x			
execlp()	x	x			
execv()	x	x			
execve()	x	x			
execvp()	x	x			
_exit()	xx	xx			
exit()	xx	xx			
exp()	xx	xx			
expm1()	y	y			
fabs()	xx	xx			
fattach()	x	x			
fchdir()	x	x			
fchmod()	x	x			
fchown()	x	x			
fclose()	xxd	xxd			
fcntl()	x	x			
fcvt()	xx	xx			
fdelrec()				x	
fdetach()	x	x			
fdopen()	xxd	xxd			
feof()	xxd	xxd			
ferror()	xxd	xxd			
fflush()	xxd	xxd			
ffs()	x	x			
fgetc()	xxd	xxd			

Funktion	XPG5	XPG4 Version 2	Erweiterungen		ANSI
			Erweiterung	*BS2000*	
fgetpos()	xxd	xxd			
fgets()	xxd	xxd			
fgetwc()	yd	yd			
fgetws()	yd	yd			
__FILE__		xx			
fileno()	xxd	xxd			
flocate()				x	
flockfile()	y				
floor()	xx	xx			
floorf()					x
floorl()					x
fmod()	xx	xx			
fmtmsg()	x	x			
fopen()	xxd	xxd			
fork()	x	x			
fpathconf()	x	x			
fprintf()	xxd	xxd			
fputc()	xxd	xxd			
fputs()	xxd	xxd			
fputwc()	yd	yd			
fputws()	yd	yd			
fread()	xxd	xxd			
free()	xx	xx			
freopen()	xxd	xxd			
frexp()	xx	xx			
fscanf()	xxd	xxd			
fseek()	xxd	xxd			
fsetpos()	xxd	xxd			
fstat()	xd	xd			
fstatvfs()	x	x			
fsync()	x	x			
ftell()	xxd	xxd			

Funktion	XPG5	XPG4 Version 2	Erweiterungen		ANSI
			Erweiterung	*BS2000*	
ftello()	yd	yd			
ftime()	xa	xa			
ftok()	x	x			
ftruncate()	x	x			
ftrylockfile()	y				
ftw()	x	x			
funlockfile()	y				
fwide()	y				x
fwprintf()	y				x
fwrite()	xxd	xxd			x
fwscanf()	y				
FD_CLR()		x			
FD_ISSET()		x			
FD_SET()		x			
FD_ZERO()		x			
gamma()	xx	xx			
garbcoll()				x	
gcvt()	xx	xx			
getc()	xxd	xxd			
getc_unlocked()	yd				
getchar()	xxd	xxd			
getchar_unlocked()	yd				
getcontext()	x	x			
getcwd()	x	x			
getdate()	x	x			
getdents()			x		
getdtablesize()	x	x			
getegid()	x	x			
getenv()	xx	xx			
geteuid()	x	x			
getgid()	x	x			
getgrent()	x	x			

Funktion	XPG5	XPG4 Version 2	Erweiterungen		ANSI
			Erweiterung	*BS2000*	
getgrgid()	x	x			
getgrgid_r()	x				
getgrnam()	x	x			
getgrnam_r()	x				
getgroups()	x	x			
gethostid()	y	y			
getitimer()	x	x			
getlogin()	xx	xx			
getlogin_r()	y				
getmsg()	x	x			
getopt()	x	x			
getpagesize()	x	x			
getpass()	x	x			
getpgit()	x	x			
getpid()	x	x			
getpgmname()				x	
getpgrp()	x	x			
getpmsg()	x	x			
getppid()	x	x			
getpriority()	x	x			
getpwent()	x	x			
getpwnam()	x	x			
getpwnam_r()	x				
getpwuid()	x	x			
getpwuid_r()	x				
getrlimit()	x	x			
getrusage()	x	x			
gets()	xxd	xxd			
getsid()	x	x			
getsubopt()	x	x			
gettimeofday()	x	x			
gettsn()				x	

Funktion	XPG5	XPG4 Version 2	Erweiterungen		ANSI
			Erweiterung	*BS2000*	
getuid()	x	x			
getutxent()	x	x			
getutxid()	x	x			
getutxline()	x	x			
getw()	xxd	xxd			
getwc	yd	yd			
getwd()	x	x			
getwchar()	yd	yd			
gmatch()			x		
gmtime()	xa	xa			
gmtime_r()	xa				
grantpt()	x	x			
hcreate()	x	x			
hdestroy()	x	x			
hsearch()	x	x			
hypot()	xx	xx			
iconv()	x	x			
iconv_close()	x	x			
iconv_open()	x	x			
ilogb()	y	y			
index()	xx	xx			
initstate()	x	x			
insque()	x	x			
ioctl()	x	x			
isalnum()	xx	xx			
isalpha()	xx	xx			
isascii()	xx	xx			
isastream()	x	x			
isatty()	x	x			
iscntrl()	xx	xx			
isdigit()	xx	xx			
isebcdic()				x	

Funktion	XPG5	XPG4 Version 2	Erweiterungen		ANSI
			Erweiterung	*BS2000*	
isgraph()	xx	xx			
islower()	xx	xx			
isnan()	x	x			
isprint()	xx	xx			
ispunct()	xx	xx			
isspace()	xx	xx			
isupper()	xx	xx			
iswalnum()	x	x			
iswalpha()	x	x			
iswcntrl()	x	x			
iswctype()	x	x			
iswdigit()	x	x			
iswgraph()	x	x			
iswlower()	x	x			
iswprint()	x	x			
iswpunct()	x	x			
iswspace()	x	x			
iswupper()	x	x			
iswxdigit()	x	x			
isxdigit()	xx	xx			
j0()	xx	xx			
j1()	xx	xx			
jn()	xx	xx			
jrand48()	x	x			
kill()	xa	xa			
killpg()	x	x			
_longjmp()	y	y			
l64a()	y	y			
labs()	xx	xx			
lchown()	x	x			
lcong48()	x	x			
ldexp()	xx	xx			

Funktion	XPG5	XPG4 Version 2	Erweiterungen		ANSI
			Erweiterung	*BS2000*	
ldiv()	xx	xx			
lfind()	x	x			
lgamma()	x	x			
__LINE__		xx			
link()	x	x			
llabs()					x
lldiv()					x
llrint()					x
llrintf()					x
llrintl()					x
llround()					x
llroundf()					x
llroundl()					x
loc1	x	x			
loc2	x	x			
localeconv()	xx	xx			
localtime()	xa	xa			
localtime_r()	xa				
lockf()	x	x			
locs	x	x			
log()	xx	xx			
log10()	xx	xx			
log1p()	y	y			
logb()	y	y			
longjmp()	xx	xx			
lrand48()	x	x			
lrint()					x
lrintf()					x
lrintl()					x
lround()					x
lroundf()					x
lroundl()					x

Funktion	XPG5	XPG4 Version 2	Erweiterungen		ANSI
			Erweiterung	*BS2000*	
lsearch()	x	x			
lseek()	xxd	xxd			
lstat()	x	x			
major()			x		
makecontext()	x	x			
makedev()			x		
malloc()	xx	xx			
mblen()	xx	xx			
mbrlen()	y				x
mbrtowc()	y				x
mbsinit()	y				x
mbsrtowcs()	y				x
mbstowcs()	xx	xx			
mbtowc()	xx	xx			
memalloc()				x	
memccpy()	x	x			
memchr()	xx	xx			
memcmp()	xx	xx			
memcpy()	xx	xx			
memfree()				x	
memmove()	xx	xx			
memset()	xx	xx			
minor()			x		
mkdir()	x	x			
mkfifo()	x	x			
mknod()	x	x			
mkstemp()	x	x			
mktemp()	xa	xa			
mktime()	xa	xa			
mmap()	x	x			
modf()	xx	xx			
mount()			x		

Funktion	XPG5	XPG4 Version 2	Erweiterungen		ANSI
			Erweiterung	*BS2000*	
mprotect()	x	x			
mrand48()	x	x			
msgctl()	x	x			
msgget()	x	x			
msgrcv()	x	x			
msgsnd()	x	x			
msync()	x	x			
munmap()	x	x			
nanosleep()	y				
nextafter()	y	y			
nftw()	x	x			
nice()	x	x			
nl_langinfo()	x	x			
nrand48()	x	x			
offsetof()				x	
open()	xxd	xxd			
opendir()	x	x			
openlog()	x	x			
optarg	x	x			
opterr	x	x			
optint	x	x			
optopt	x	x			
pathconf()	x	x			
pause()	x	x			
pclose()	x	x			
perror()	xxd	xxd			
pipe()	x	x			
poll()	x	x			
popen()	x	x			
pow()	xx	xx			
printf()	xxd	xxd			
ptsname()	x	x			

Funktion	XPG5	XPG4 Version 2	Erweiterungen		ANSI
			Erweiterung	*BS2000*	
putc()	xxd	xxd			
putc_unlocked()	yd				
putchar()	xxd	xxd			
putchar_unlocked()	yd				
putenv()	x	x			
putmsg()	x	x			
putpmsg()	x	x			
putpwent()			x		
puts()	xxd	xxd			
pututxline()	x	x			
putw()	xxd	xxd			
putwc()	yd	yd			
putwchar()	yd	yd			
qsort()	xx	xx			
raise()	xa	xa			
rand()	xx	xx			
rand_r()	y				
random()	x	x			
re_cmp()	x	x			
re_exec()	x	x			
read()	xxd	xxd			
readdir()	x	x			
readdir_r()	x				
readlink()	x	x			
readv()	x	x			
realloc()	xx	xx			
realpath()	x	x			
regcmp()	x	x			
regex()	x	x			
remainder()	y	y			
remove()	xxd	xxd			
remque()	x	x			

Funktion	XPG5	XPG4 Version 2	Erweiterungen		ANSI
			Erweiterung	*BS2000*	
rename()	xxd	xxd			
rewind()	xxd	xxd			
rewinddir()	x	x			
rindex()	xx	xx			
rint()	y	y			
rintf()					x
rintl()					x
rmdir()	x	x			
round()					x
roundf()					x
roundl()					x
_setjmp	y	y			
sbrk()	x	x			
scalb()	y	y			
scanf()	xxd	xxd			
seed48()	x	x			
seekdir()	x	x			
select()	x	x			
semctl()	x	x			
semget()	x	x			
semop()	x	x			
setbuf()	xxd	xxd			
setcontext()	x	x			
setgid()	x	x			
setgrent()	x	x			
setitimer()	x	x			
setjmp()	xx	xx			
setkey()	x	x			
setlocale()	xa	xa			
setlogmask()	x	x			
setpgid()	x	x			
setpgrp()	x	x			

Funktion	XPG5	XPG4 Version 2	Erweiterungen		ANSI
			Erweiterung	*BS2000*	
setpriority()	x	x			
setpwent()	x	x			
setregid()	x	x			
setreuid()	x	x			
setrlimit()	x	x			
setsid()	x	x			
setstate()	x	x			
setuid()	x	x			
setutxent()	x	x			
setvbuf()	xxd	xxd			
shmat()	x	x			
shmctl()	x	x			
shmdt()	x	x			
shmget()	x	x			
sigaction()	x	x			
sigaddset()	x	x			
sigdelset()	x	x			
sigemptyset()	x	x			
sigfillset()	x	x			
sighold()	x	x			
sigignore()	x	x			
siginterrupt()	x	x			
sigismember()	x	x			
siglongjmp()	x	x			
signal()	xa	xa			
signalstack()	x	x			
sigpause()	x	x			
signgam	x	x			
sigpending()	x	x			
sigprocmask()	x	x			
sigrelse()	x	x			
sigset()	x	x			

Funktion	XPG5	XPG4 Version 2	Erweiterungen		ANSI
			Erweiterung	*BS2000*	
sigsetjmp()	x	x			
sigstack()	x	x			
sigsuspend()	x	x			
sin()	xx	xx			
sinh()	xx	xx			
sleep()	xa	xa			
sprintf()	xx	xx			
sqrt()	xx	xx			
srand()	xx	xx			
srand48()	x	x			
srandom()	x	x			
sscanf()	xx	xx			
stat()	xd	xd			
statvfs()	x	x			
__STDC__		xx			
__STDC_VERSION__					x
stderr	xx	xx			
stdin	xx	xx			
stdout	xx	xx			
step()	x	x			
strcasecmp()	x	x			
strcat()	xx	xx			
strchr()	xx	xx			
strcmp()	xx	xx			
strcoll()	xa	xa			
strcpy()	xx	xx			
strcspn()	xx	xx			
strdup()	x	x			
strerror()	xx	xx			
strfill()				x	
strftime()	xa	xa			
strlen()	xx	xx			

Funktion	XPG5	XPG4 Version 2	Erweiterungen		ANSI
			Erweiterung	*BS2000*	
strlower()				x	
strncasecmp()	x	x			
strncat()	xx	xx			
strncmp()	xx	xx			
strncpy()	xx	xx			
strpbrk()	xx	xx			
strrchr()	xx	xx			
strspn()	xx	xx			
strstr()	xx	xx			
strtod()	xa	xa			
strtok()	xx	xx			
strtok_r()	y				
strtol()	xx	xx			
strtoll()					x
strtoul()	xx	xx			
strtoull()					x
strupper()				x	
strxfrm()	xa	xa			
swab()	x	x			
swapcontext()	x	x			
swprintf()	y				x
swscanf()	y				x
symlink()	x	x			
sync()	x	x			
sysconf()	x	x			
sysfs()			x		
syslog()	x	x			
system()	xa	xa			
tan()	xx	xx			
tanh()	xx	xx			
tcdrain()	x	x			
tcflow()	x	x			

Funktion	XPG5	XPG4 Version 2	Erweiterungen		ANSI
			Erweiterung	*BS2000*	
tcflush()	x	x			
tcgetattr()	x	x			
tcgetpgrp()	x	x			
tcgetsid()	x	x			
tcsendbreak()	x	x			
tcsetattr()	x	x			
tcsetpgrp()	x	x			
tdelete()	x	x			
tell()				x	
telldir()	x	x			
tempnam()	x	x			
tfind()	x	x			
__TIME__		xx			
time()	xa	xa			
times()	x	x			
timezone	x	x			
tmpfile()	xxd	xxd			
tmpnam()	xxd	xxd			
toascii()	xx	xx			
toebcdic()				x	
_tolower()	x	x			
tolower()	xa	xa			
_toupper()	x	x			
toupper()	xa	xa			
towctrans()	x				x
towlower()	x	x			
towupper()	x	x			
truncate()	x	x			
tsearch()	x	x			
ttyname()	x	x			
ttyname_r()	x				
ttyslot()	x	x			

Funktion	XPG5	XPG4 Version 2	Erweiterungen		ANSI
			Erweiterung	*BS2000*	
twalk()	x	x			
tzname	x	x			
tzset()	x	x			
ualarm()	x	x			
ulimit()	x	x			
umask()	x	x			
umount()			x		
uname()	x	x			
ungetc()	xxd	xxd			
ungetwc()	x	x			
unlink()	xxd	xxd			
unlockpt()	x	x			
usleep()	x	x			
utime()	x	x			
utimes()	x	x			
va_arg()	xx	xx			
va_end()	xx	xx			
va_start()	xx	xx			
valloc()	y	y			
vfork()	x	x			
vfprintf()	xxd	xxd			
vfwprintf	y				x
vprintf()	xxd	xxd			
vsprintf()	xx	xx			
vswprintf()	y				x
vwprintf()	y				x
wait()	x	x			
wait3()	x	x			
waitid()	x	x			
waitpid()	x	x			
wcrtomb()	y				
wcscat()		x			

Funktion	XPG5	XPG4 Version 2	Erweiterungen		ANSI
			Erweiterung	*BS2000*	
wcschr()		x			
wcscmp()		x			
wcscoll	x	x			
wcscpy()	x	x			
wcscspn()	x	x			
wcsftime	x	x			
wcslen()	x	x			
wcsncat()	x	x			
wcsncmp()	x	x			
wcsncpy()	x	x			
wcspbrk()	x	x			x
wcsrchr()	x	x			x
wcsrtombs()	x				x
wcsspn()	x	x			
wcsstr()	x				
wcstod()	x	x			x
wcstok()	x	x			
wcstol()	x	x			
wcstoll()					x
wcstombs()	xx	xx			
wcstoul()	x	x			
wcstoull()					x
wcswcs()	x	x			
wcswidth()	x	x			
wcsxfrm	x	x			
wctob()	y				x
wctomb()	xx	xx			
wctrans()	y				x
wctype()	x	x			
wcwidth()	x	x			
wmemchr()	y				x
wmemcmp()	y				x

Funktion	XPG5	XPG4 Version 2	Erweiterungen		ANSI
			Erweiterung	*BS2000*	
wmemcpy()	y				x
wmemmove()	y				x
wmemset()	y				x
wprintf()	y				x
write()	xxd	xxd			
writev()	x	x			
wscanf()	y				x
y0()	xx	xx			
y1()	xx	xx			
yn()	xx	xx			

Wahl der Funktionalität

Es besteht die Möglichkeit, zwischen den unterschiedlichen Funktionsumfängen zu wählen. Dabei wird im Folgenden zwischen dem um die POSIX-Funktionalität erweiterten Funktionsumfang und dem im BS2000 (ohne POSIX) zur Verfügung stehenden Funktionsumfang unterschieden, der die BS2000-Funktionalität darstellt.

Diejenigen C-Bibliotheksfunktionen, die die BS2000-Funktionalität zur Verfügung stellen, bilden die Basis der Bibliothek.
Zusätzlich stellen Ihnen die weiteren Funktionen der C-Bibliothek die POSIX-Funktionalität zur Verfügung. Sie können also bei Wahl der erweiterten Funktionalität alle Funktionen der Bibliothek nutzen, d.h. sowohl die BS2000- als auch die zusätzlichen XPG4 Version 2-konformen Funktionen.

Für eine kleine Zahl von Funktionen gibt es im BS2000 und in POSIX unterschiedliche Ausprägungen. Dabei handelt es sich zum einen um Funktionen für die Ein-/Ausgabe und Dateizugriffe (eine Auflistung dieser Funktionen finden Sie auf Seite 75), zum anderen um Zeitfunktionen, Signalbearbeitungs- und Unterbrechungs-Funktionen sowie die Funktionen `clock()` und `system()`.
Im folgenden ist für beide Varianten des Funktionsumfanges beschrieben, welche Ausprägung jeweils verwendet wird.

Um POSIX-Funktionalität erweiterter Funktionsumfang

Wenn Sie ein Programm in der **POSIX-Shell** übersetzen, binden und starten (siehe auch Handbuch „C/C++ POSIX-Kommandos des C- und C++- bzw. C/C++-Compilers“), stehen alle Funktionen der C-Bibliothek zur Verfügung, wie sie im Folgenden aufgelistet sind. Dieser Funktionsumfang wird im Folgenden um **POSIX-Funktionalität** erweiterter Funktionsumfang genannt:

- alle XPG4 Version 2-konformen Funktionen (in der Tabelle auf Seite 19 ff. in der Spalte „XPG4 Version 2“ mit x, y und xx gekennzeichnet)
- alle Funktionen, die als Erweiterung gekennzeichnet sind (in der Tabelle auf Seite 19 ff. in den Spalten „Erweiterung“ und „BS2000“ mit x gekennzeichnet)
- bei den Funktionen, die mit xa gekennzeichnet sind, wird die XPG4 Version 2-konforme Funktionalität zur Verfügung gestellt. Dabei handelt es sich um folgende Funktionsgruppen:
  - die Zeitfunktionen `clock()`, `ctime()`, `ctime_r()`, `ftime()`, `gmtime()`, `localtime()`, `mktime()`, `time()`
  - die Funktionen zur Prozesssteuerung `abort()`, `alarm()`, `_exit()`, `kill()`, `raise()`, `signal()`

- bei den Funktionen, die in der Tabelle mit xd gekennzeichnet sind, besteht die Möglichkeit, im Einzelfall auf BS2000- oder auf POSIX-Dateien zuzugreifen. Dies kann gesteuert werden, wie es im Abschnitt „Wahl des Dateisystems und der Systemumgebung“ auf Seite 42 beschrieben ist. Zu dieser Gruppe gehört auch die Funktion `system()`, da sie wie die Dateizugriffunktionen auf Quellprogrammebene gesteuert werden kann.

Intern wird ein Programm, das in der POSIX-Shell gestartet wurde, mit `fork()` und einer `exec()`-Funktion gestartet und hat damit einen Vaterprozess.

Sie erhalten den um die POSIX-Funktionalität erweiterten Funktionsumfang ebenfalls, wenn Sie das Programm in der **BS2000-Kommandoebene** übersetzen, binden und starten. Dabei müssen Sie jedoch Folgendes beachten:

1. Beim Übersetzen muss Folgendes beachtet werden:

   a) Zusätzlich zur Bibliothek `$.SYSLNK.CRTE` muss die Bibliothek `$.SYSLIB.POSIX-HEADER` angegeben werden, damit die richtigen Include-Dateien gefunden werden (Option `STD-INCLUDE-LIBRARY`).

   b) Das Define `_OSD_POSIX` muss gesetzt werden. Dazu wählen Sie eine der folgenden Möglichkeiten:

      – Im Quellcode vor der ersten `#include`-Anweisung Folgendes angeben:

        ```
        #define _OSD_POSIX
        ```

      – Beim Übersetzungslauf die Option `SOURCE-PROPERTIES` setzen:

        ```
        SOURCE-PROPERTIES=PAR(DEFINE=_OSD_POSIX)
        ```

2. Beim Binden muss der Bindeschalter `$.SYSLNK.CRTE.POSIX` vorrangig vor der Bibliothek `$.SYSLNK.CRTE` bzw. `$.SYSLNK.CRTE.PARTIAL-BIND` eingebunden werden.

Dieses Programm, das auf der BS2000-Kommandoebene übersetzt, gebunden und gestartet wurde, läuft in einer Task und hat damit keinen Vaterprozess.

BS2000-Funktionalität

Wenn Sie in Ihrem Programm nur die BS2000-Funktionalität verwenden wollen, übersetzen Sie das Programm und binden nur die Bibliothek `$.SYSLNK.CRTE` dazu. Die Umgebungsvariable `PROGRAM-ENVIRONMENT='SHELL'` darf nicht gesetzt sein.
Wenn Sie nur die BS2000-Funktionalität verwenden ist es sinnvoll, mit dem Handbuch „C-Bibliotheksfunktionen V2.3A (BS2000)“ zu arbeiten.

Bei Wahl der BS2000-Funktionalität wird nur ein Teil der Bibliothek unterstützt:

– alle XPG4 Version 2-konformen Funktionen, die auch von der bisherigen C-Bibliothek (BS2000) unterstützt wurden (in der Tabelle auf Seite 19 ff. in der Spalte „XPG4“ mit xx gekennzeichnet)

– alle Funktionen, die als Erweiterung mit *BS2000* gekennzeichnet sind (in der Tabelle auf Seite 19 ff. in der Spalte „BS2000“ mit x gekennzeichnet)

– bei den Funktionen, die mit xa gekennzeichnet sind, wird nur die BS2000-Funktionalität zur Verfügung gestellt

– bei den Funktionen, die in der Tabelle mit xd gekennzeichnet sind, kann nur auf BS2000-Dateien zugegriffen werden.

Wahl des Dateisystems und der Systemumgebung

Für alle Ein-/Ausgabe- und Dateizugriffsfunktionen, die sowohl POSIX- als auch BS2000-Dateien bearbeiten können und bei denen ein Pfadname als Argument angegeben werden muss, können Sie im Quellcode einzeln festlegen, welcher Dateityp bearbeitet werden soll. Mit der Wahl des Dateityps legen Sie automatisch fest, mit welcher Funktionalität die jeweilige Funktion aufgerufen wird. Dies geschieht zum einen über die Umgebungsvariable `PROGRAM_ENVIRONMENT`, zum anderen durch Einhalten einer bestimmten Syntax auf Quellprogrammebene.

Verknüpfung der Ein-/Ausgabeströme

Wenn Sie beim Binden des Programms den POSIX-Bindeschalter angegeben haben und POSIX aktiv ist, werden die Standard-Ein-/Ausgabeströme `stdin`, `stdout` und `stderr` über POSIX geöffnet.
In Batchjobs, Prozeduren oder falls die Umgebungsvariable `PROGRAM_ENVIRONMENT` nicht auf SHELL gesetzt ist, werden die Standard-Ein-/Ausgabeströme über POSIX mit den BS2000-Systemdateien (`SYSDTA, SYSOUT`) verknüpft, sonst mit dem Terminal.

Ohne POSIX werden die Standard-Ein-/Ausgabeströme `stdin`, `stdout` und `stderr` direkt mit den BS2000-Systemdateien (`SYSDTA`, `SYSOUT`) verknüpft.

Umgebungsvariable PROGRAM_ENVIRONMENT

Mit der Umgebungsvariablen `PROGRAM_ENVIRONMENT` kann im BS2000/OSD eingestellt werden, ob Dateinamen oder im Funktionsaufruf `system()` angegebene Kommandos, die kein BS2000- oder POSIX-Präfix haben, als BS2000- oder POSIX-Datei bzw. Kommando interpretiert werden.

Auf der BS2000-Kommandoebene ist `PROGRAM_ENVIRONMENT` nicht gesetzt. Zum Setzen der Umgebungsvariablen siehe Abschnitt „Umgebungsvariablen“ auf Seite 71.

Beim Start der POSIX-Shell wird `PROGRAM_ENVIRONMENT` automatisch auf den Wert `SHELL` gesetzt, d.h. Dateinamen und Kommandos, die nicht mit `"/BS2/"` beginnen, werden als POSIX-Dateinamen bzw. -Kommandos interpretiert.

Dateinamen bzw. Kommandos, die den Syntaxregeln der entsprechenden Umgebung widersprechen, werden mit einer Fehlermeldung quittiert.
Existiert die angegebene Datei oder das angegebene Kommando in der gewählten Umgebung nicht, wird dies ebenfalls gemeldet.

Explizite Kennzeichnung von Dateinamen als POSIX oder BS2000

Beginnt der Dateiname mit einem Schrägstrich (/), wird der Dateiname als absoluter Pfadname einer POSIX-Datei interpretiert.

Ist der Dateiname in der Form `*POSIX(`*name*`)` angegeben, wird er ebenfalls als POSIX-Dateiname interpretiert.

Beginnt der Dateiname mit `/BS2/`, wird der auf `/BS2/` folgende Dateiname als BS2000-Dateiname interpretiert.

Explizite Kennzeichnung von Kommandos

Beginnt das im Funktionsaufruf `system()` angegebene Kommando mit `/BS2/`, wird das auf `/BS2/` folgende Kommando als BS2000-Kommando interpretiert.

Ist das Kommando in der Form `*POSIX(` *kommando*`)` angegeben, wird es als POSIX-Kommando interpretiert.

Syntax im Quellprogramm

Wenn eine POSIX-Datei bearbeitet werden soll, geben Sie entweder den absoluten Pfadnamen der Datei an (siehe auch Handbuch „POSIX Grundlagen (BS2000/OSD)“) oder Sie kennzeichnen den Namen mit `*POSIX(`*dateiname*`)`.

Wenn eine BS2000-Datei bearbeitet werden soll, kennzeichnen Sie den Dateinamen mit `/BS2/`. Sobald auf eine BS2000-Datei zugegriffen wird, gilt die BS2000-Funktionalität der entsprechenden Funktion. Wenn sie von der XPG4 Version 2-Funktionalität abweicht, ist dies am linken Rand der Beschreibung mit *BS2000* gekennzeichnet.

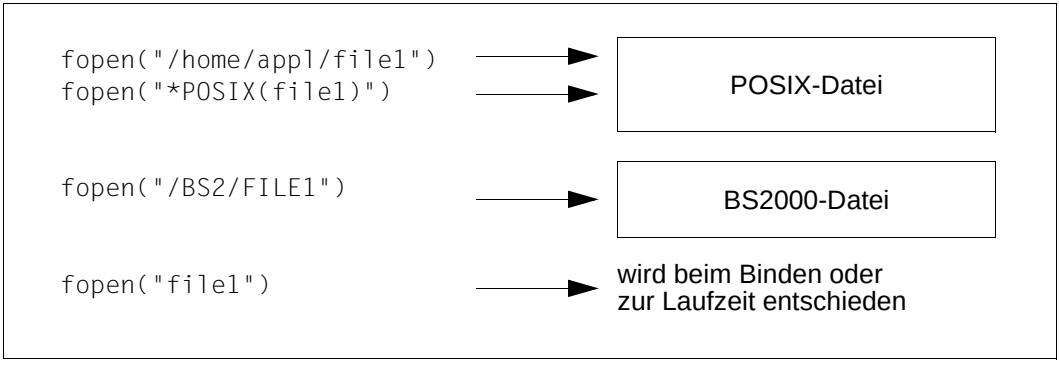


Bild 2: Steuermöglichkeiten auf Quellcodeebene

Die Funktion `system()` kann auf dieselbe Weise gesteuert werden, nur wird statt eines Dateinamens ein Kommando für die gewünschte Systemumgebung angegeben.

Portabilität

Anwender, die Programme schreiben wollen, die gemäß XPG4-Standard Version 2 portabel sind, müssen das Makro _XOPEN_SOURCE setzen und ebenfalls das Makro _XOPEN_SOURCE_EXTENDED auf den Wert 1. Auf diese Weise werden die vom XPG-Standard Version 2 geforderten und ausdrücklich zugelassenen Bezeichner sichtbar gemacht. Diese Makros müssen gesetzt werden, bevor die erste Include-Datei eingebunden wird. Dies kann entwerden beim Übersetzungslauf durch Angabe der entsprechenden Compiler-Option erfolgen oder im Quellcode mit Hilfe von #define-Anweisungen.

XPG4 Version 2-definierte Bezeichner sind nur dann nicht definiert, wenn die `#undef`-Anweisung angegeben wird (siehe auch Abschnitt „Funktionen und Makros“ auf Seite 13). Diese `#undef`-Anweisungen müssen nach den `#include`-Anweisungen aufgerufen werden.

Wenn das Makro `_XOPEN_SOURCE` auf 500 gesetzt ist, werden nur die vom XPG4 Version 2-Standard ausdrücklich zugelassenen Bezeichner sichtbar gemacht. Das Makro `_XOPEN_SOURCE_EXTENDED` wird in dieser Konstellation ignoriert. Wird das Makro `_XOPEN_SOURCE` nicht auf den Wert 500 gesetzt, aber das Makro `_XOPEN_SOURCE_EXTENDED` auf 1, so sind nur die im erweiterten XPG4 Version 2-Standard enthaltenen Bezeichner sichtbar.

`_XOPEN_SOURCE_EXTENDED` kann beim Übersetzungslauf definiert werden. Um also maximale Portabilität zu unterstützen, sollte in Anwendungen sichergestellt werden, dass `_XOPEN_SOURCE _EXTENDED` auf 1 gesetzt ist, indem entweder eine Compileroption benutzt oder eine `#define`-Anweisung vor der ersten `#include`-Anweisung in den Quellcode eingetragen wird.

Anwendungen, die Funktionalität verwenden, die in diesem Handbuch als Erweiterung gekennzeichnet ist (markiert mit *BS2000* oder *Erweiterung*), sind nicht streng XPG4 Version 2- oder ISO C-konform.

Um Programme zu schreiben, die gemäß XPG5-Standard portabel sind, muss das Makro _XOPEN_SOURCE auf den Wert 500 gesetzt werden. Das Makro _XOPEN_SOURCE_EXTENDED wird in dieser Konstellation ignoriert. In dieser Implementierung sind nicht alle im XPG5-Standard enthaltenen Funktionsgruppen und Include-Dateien realisiert (z.B. gibt es keine asynchrone Ein-/Ausgabe und keine Echtzeitfunktionen). Die entsprechenden Funktionstest-Makros sind in der Include-Datei <unistd.h> auf den Wert -1 gesetzt.

Namensraum

Alle Bezeichner, die in diesem Handbuch erwähnt werden, außer `environ`, werden in mindestens einer Include-Datei definiert (siehe auch Kapitel „Funktionen und Variablen alphabetisch (n - y)“ auf Seite 625). Wenn `_XOPEN_SOURCE` definiert ist, definiert oder deklariert jede Include-Datei Bezeichner, die möglicherweise mit Bezeichnern der Anwendung in Konflikt geraten. Die Bezeichnermenge, die für die Anwendung sichtbar ist, besteht aus den Bezeichnern, die über die `#include`-Anweisung eingebunden werden, und den zusätzlichen Bezeichnern, die von der Implementierung reserviert sind (siehe auch C- und C++-Benutzerhandbücher).

Zeichensätze

Die C-Laufzeitbibiliothek unterstützt den portablen Zeichensatz von XPG4 Version 2 und in dieser Version als kodierten Zeichensatz nur EBCDIC.

Portabler Zeichensatz

Jede unterstützte Lokalität bezieht sich auf den portablen Zeichensatz. Er besteht aus 128 Zeichen (7-Bit-Code). Die folgende Tabelle gibt für jedes Zeichen des portablen Zeichensatzes den symbolischen Namen, die zugehörige Glyphe, den Klassennamen der POSIX-Lokalität, die Kodierung im ASCII- und im EBCDIC-Format an:

symbolischer Name	**Glyphe**	**Klasse der POSIX-Lokalität**	**ASCII**		**EBCDIC**
			dez	**hex**	**hex**
<NUL>		control	0	00	00
<SOH>		control	1	01	01
<STX>		control	2	02	02
<ETX>		control	3	03	03
<EOT>		control	4	04	37
<ENQ>		control	5	05	2D
<ACK>		control	6	06	2E
<alert>		control	7	07	2F
<backspace>		control	8	08	16
<tab>		control space blank	9	09	05
<newline>		control space	10	0A	15
<vertical--tab>		control space	11	0B	0B
<form-feed>		control space	12	0C	0C
<carriage-return>		control space	13	0D	0D
<SO>		control	14	0E	0E
<SI>		control	15	0F	0F
<DLE>		control	16	10	10
<DC1>		control	17	11	11
<DC2>		control	18	12	12
<DC3>		control	19	13	13
<DC4>		control	20	14	3C
<NAK>		control	21	15	3D

symbolischer Name	Glyphe	Klasse der POSIX-Lokalität	ASCII		EBCDIC
			dez	hex	hex
<SYN>		control	22	16	32
<ETB>		control	23	17	26
<CAN>		control	24	18	18
<EM>		control	25	19	19
<SUB>		control	26	1A	3F
<ESC>		control	27	1B	27
<IS4>		control	28	1C	1C
<IS3>		control	29	1D	1D
<IS2>		control	30	1E	1E
<IS1>		control	31	1F	1F
<space>		space blank	32	20	40
<exclamation-mark>	!	punct	33	21	5A
<quotation-mark>	"	punct	34	22	7F
<number-sign>	#	punct	35	23	7B
<dollar-sign>	$	punct	36	24	5B
<percent-sign>	%	punct	37	25	6C
<ampersand>	&	punct	38	26	50
<apostrophe>	'	punct	39	27	7D
<left-parenthesis>	(	punct	40	28	4D
<right-parenthesis>	)	punct	41	29	5D
<asterisk>	*	punct	42	2A	5C
<plus-sign>	+	punct	43	2B	4E
<comma>	,	punct	44	2C	6B
<hyphen>	-	punct	45	2D	60
<period>	.	punct	46	2E	4B
<slash>	/	punct	47	2F	61
<zero>	0	digit xdigit	48	30	F0
<one>	1	digit xdigit	49	31	F1
<two>	2	digit xdigit	50	32	F2
<three>	3	digit xdigit	51	33	F3
<four>	4	digit xdigit	52	34	F4
<five>	5	digit xdigit	53	35	F5

symbolischer Name	Glyphe	Klasse der POSIX-Lokalität	ASCII		EBCDIC
			dez	hex	hex
<six>	6	digit xdigit	54	36	F6
<seven>	7	digit xdigit	55	37	F7
<eigth>	8	digit xdigit	56	38	F8
<nine>	9	digit xdigit	57	39	F9
<colon>	:	punct	58	3A	7A
<semicolon>	;	punct	59	3B	5E
<less-than-sign>	<	punct	60	3C	4C
<equals-sign>	=	punct	61	3D	7E
<greater-than-sign>	>	punct	62	3E	6E
<question-mark>	?	punct	63	3F	6F
<commercial-at>	@	punct	64	40	7C
<A>	A	upper xdigit	65	41	C1
<B>	B	upper xdigit	66	42	C2
<C>	C	upper xdigit	67	43	C3
<D>	D	upper xdigit	68	44	C4
<E>	E	upper xdigit	69	45	C5
<F>	F	upper xdigit	70	46	C6
<G>	G	upper	71	47	C7
<H>	H	upper	72	48	C8
<I>	I	upper	73	49	C9
<J>	J	upper	74	4A	D1
<K>	K	upper	75	4B	D2
<L>	L	upper	76	4C	D3
<M>	M	upper	77	4D	D4
<N>	N	upper	78	4E	D5
<O>	O	upper	79	4F	D6
<P>	P	upper	80	50	D7
<Q>	Q	upper	81	51	D8
<R>	R	upper	82	52	D9
<S>	S	upper	83	53	E2
<T>	T	upper	84	54	E3
<U>	U	upper	85	55	E4

symbolischer Name	Glyphe	Klasse der POSIX-Lokalität	ASCII		EBCDIC
			dez	hex	hex
<V>	V	upper	86	56	E5
<W>	W	upper	87	57	E6
<X>	X	upper	88	58	E7
<Y>	Y	upper	89	59	E8
<Z>	Z	upper	90	5A	E9
<left-square-bracket>	[	punct	91	5B	BB
<backslash>	\	punct	92	5C	BC
<right-square-bracket>	]	punct	93	5D	BD
<circumflex>	^	punct	94	5E	6A
<underscore>	]	punct	95	5F	6D
<grave-accent>	_	punct	96	60	4A
<a>	a	lower xdigit	97	61	81
<b>	b	lower xdigit	98	62	82
<c>	c	lower xdigit	99	63	83
<d>	d	lower xdigit	100	64	84
<e>	e	lower xdigit	101	65	85
<f>	f	lower xdigit	102	66	86
<g>	g	lower	103	67	87
<h>	h	lower	104	68	88
<i>	i	lower	105	69	89
<j>	j	lower	106	6A	91
<k>	k	lower	107	6B	92
<l>	l	lower	108	6C	93
<m>	m	lower	109	6D	94
<n>	n	lower	110	6E	95
<o>	o	lower	111	6F	96
<p>	p	lower	112	70	97
<q>	q	lower	113	71	98
<r>	r	lower	114	72	99
<s>	s	lower	115	73	A2
<t>	t	lower	116	74	A3
<u>	u	lower	117	75	A4

symbolischer Name	Glyphe	Klasse der POSIX-Lokalität	ASCII		EBCDIC
			dez	hex	hex
<v>	v	lower	118	76	A5
<w>	w	lower	119	77	A6
<x>	x	lower	120	78	A7
<y>	y	lower	121	79	A8
<z>	z	lower	122	7A	A9
<left-curly-bracket>	{	punct	123	7B	FB
<vertical-line>	|	punct	124	7C	4F
<right-curly-bracket>	}	punct	125	7D	FD
<tilde>	~	punct	126	7E	FF
<DEL>	DEL	control	127	7F	07

Der EBCDIC-Zeichensatz ist ein 8-Bit-Code und umfasst insgesamt 256 Zeichen. Die verschiedenen Varianten des EBCDIC-Zeichensatzes finden Sie im Handbuch „BS2000/OSD-BC Systemanwendung“.

Die symbolischen Namen des portablen Zeichensatzes werden für die Zuordnung der Zeichensatzkodierung in einer Zeichensatztabelle verwendet.

Langzeichensatz

Alle Langzeichensätze in einem Prozess bestehen aus Zeichen mit der gleichen Bitanzahl. **Langzeichen** sind nicht zu verwechseln mit **Multibyte-Zeichen**, die aus einer variablen Anzahl von Bytes bestehen können.

In der C-Laufzeitbibiliothek werden zwar Funktionen unterstützt, die Multibyte-Zeichen behandeln, die tatsächliche Länge eines Multibyte-Zeichen beträgt in dieser Version jedoch immer ein Byte (= 8 Bit), da für den Langzeichensatz nur EBCDIC zur Verfügung steht.

Zeichenklassen

In der vorherigen Tabelle sind den Zeichen des portablen Zeichensatzes Zeichenklassen zugeordnet, wie sie in der POSIX-Lokalität für die Kategorie LC_CTYPE definiert sind. Darüber hinaus sind weitere Zeichenklassen definiert, die Ober- und Untermengen dieser Zeichenklassen darstellen:

Zeichenklasse	Umfang
alpha	upper + lower
blank	Untermenge von space: <blank> und <tab>
cntrl	Steuerzeichen
digit	Dezimalzeichen
graph	alpha + digit + punct + space
lower	Kleinbuchstaben
print	alpha + digit + punct
punct	Interpunktionszeichen
space	Zwischenraumzeichen
tolower	Abbildung von Großbuchstaben auf Kleinbuchstaben
toupper	Abbildung von Kleinbuchstaben auf Großbuchstaben
upper	Großbuchstaben
xdigit	Zeichenumfang für Hexadezimalzeichen: digit + A–F + a–f

Lokalität

Die Lokalität ist eine Untermenge der Festlegungen für die Laufzeitumgebung. Durch die Lokalität wird das Verhalten von C-Programmen in Bezug auf landesspezifische Konventionen, Normen und Sprachen beeinflusst. Die Lokalität besteht aus einer oder mehreren Kategorien. In XPG4 Version 2-konformen Umgebungen werden folgende Kategorien unterstützt:

LC_ALL Beeinflusst alle Werte der aktuellen Lokalität.

LC_COLLATE Beeinflusst die Sortierreihenfolge von Zeichen. Jedes Zeichen wird im Verhältnis zu einem anderen Zeichen durch ein Gewicht definiert. Dies hat Auswirkung auf das Verhalten von strcoll() und strxfrm().

Im POSIX-Subsystem heißt die entsprechende Beschreibungsdatei /usr/lib/locale/*locale*/LC_COLLATE.

Im BS2000 heißt die entsprechende Tabelle `COLL/uscol`.

`LC_CTYPE` Beeinflusst die Zuordnung von Zeichen zu Zeichenklassen, die Zuordnung zwischen Groß- und Kleinbuchstaben und andere Zeicheneigenschaften.

Im POSIX-Subsystem heißt die entsprechende Beschreibungsdatei `/usr/lib/locale/`*locale*`/LC_CTYPE`.

Im BS2000 gibt es 3 Tabellen für alle EBCDIC-Zeichen:

Die Klassifizierungstabelle `TYPE/ustyp` ordnet jedes EBCDIC-Zeichen einer bestimmten Zeichenklasse zu. Diese Klasse wird durch folgende Werte dargestellt:

Zeichenklasse	**Assembler-Kodierung**	**C-Kodierung**
Großbuchstabe (`upper`)	`X'01'`	`_U`
Kleinbuchstabe (`lower`)	`X'02'`	`_L`
Dezimalziffer (`digit`)	`X'04'`	`_N`
Zwischenraum (`space`)	`X'08'`	`_S`
Sonderzeichen (`punct`)	`X'10'`	`_P`
Steuerzeichen (`cntrl`)	`X'20'`	`_C`
Hexadezimalzeichen (`xdigit`)	`X'40'`	`_X`

Die C-Werte sind in der Include-Datei `ctype.h` definiert.

Die Tabellen für die Umwandlung von Groß- in Kleinbuchstaben `LOWER/uslow` bzw. von Klein- in Großbuchstaben `UPPER/usupp` geben für jedes Zeichen von `X'00'` bis `X'FF'` das Ergebniszeichen der Umwandlung an. Diese Tabellen werden von den Makros `toupper()` und `tolower()` zur Umwandlung in Groß- bzw. Kleinbuchstaben verwendet. Die Tabelle braucht nur für die Zeichen belegt sein, die in der Klassifizierungstabelle als Groß- bzw. Kleinbuchstaben klassifiziert sind.

`LC_MESSAGES` Beeinflusst die Formate von Meldungen.

Im POSIX-Subsystem heißt die entsprechende Beschreibungsdatei `/usr/lib/locale/`*locale*`/LC_MESSAGES`.

Diese Kategorie wird durch die BS2000-Funktionalität nicht unterstützt.

`LC_MONETARY` Beeinflusst die Formate von monetären Werten.

Im POSIX-Subsystem heißt die entsprechende Beschreibungsdatei `/usr/lib/locale/`*locale*`/LC_MONETARY`.

`LC_NUMERIC` Beeinflusst die Darstellung von nichtmonetären numerischen Werten bei formatierter Ein-/Ausgabe (`fprintf()`, `fscanf()`) und bei der Umwandlung von Zeichenketten (`atof()`, `strtod()`), sowie die von `localeconv()` gelieferten Werte.

Im POSIX-Subsystem heißt die entsprechende Beschreibungsdatei `/usr/lib/locale/`*locale*`/LC_NUMERIC`.

`LC_TIME` Beeinflusst die Darstellung von Datum und Uhrzeit beim Aufruf von `strftime()`.

Im POSIX-Subsystem heißt die entsprechende Beschreibungsdatei `/usr/lib/locale/`*locale*`/LC_TIME`.

Diese Lokalitätskategorien sind auch als Umgebungsvariablen definiert.

Die XPG4 Version 2-konformen Kommandos (z.B. POSIX-Kommandos) werden in ihrem Verhalten durch die aktuelle Lokalität beeinflusst (siehe Abschnitt „Umgebungsvariablen“ auf Seite 71 und Handbuch „POSIX Kommandos (BS2000/OSD)“). Mit den C-Bibliotheksfunktionen `setlocale()` und `localeconv()` kann die aktuelle Lokalität zur Laufzeit eines C-Programms geändert werden, andere C-Bibliotheksfunktionen werden von der aktuellen Lokalität beeinflusst:

`atof()`	`isgraph()`	`isxdigit()`	`strxfrm()`
`isalnum()`	`islower()`	`localeconv()`	`tolower()`
`isalpha()`	`isprint()`	`setlocale()`	`toupper()`
`isascii()`	`ispunct()`	`strcoll()`	`wctomb()`
`iscntrl()`	`isspace()`	`strftime()`	`wcstombs()`
`isdigit()`	`isupper()`	`strtod()`	

Die C-Laufzeitbibiliothek stellt einige vordefinierte Lokalitäten zur Verfügung (siehe Abschnitt „Vordefinierte Lokalitäten“ auf Seite 55). Der Benutzer kann aber auch eigene Lokalitäten definieren (siehe Abschnitt „Benutzerspezifische Lokalitäten“ auf Seite 70).

Für die Unterstützung des Euros stellt Ihnen CRTE die vordefinierten Lokalitäten `De.EDF04F` und `De.EDF04F@euro` zur Verfügung. Diese beiden Lokalitäten unterscheiden sich nur in der Kategorie `LC_MONETARY`, die für die Lokalität `De.EDF04F` die deutsche Mark (DM) darstellt, für die Lokalität `De.EDF04F@euro` den Euro.

Wenn der Wert einer Lokalitätsumgebungsvariablen mit einem Schrägstrich (/) beginnt, wird er als Pfadname für die Lokalitätsdefinition interpretiert.

Anwendungen können die aktuelle Lokalität durch einen `setlocale`-Aufruf ändern, d.h. mit einer anderen vordefinierten Lokalität besetzen. Wird die Funktion mit einer leeren Zeichenkette für *locale* aufgerufen, wird der Wert der Umgebungsvariablen ausgewertet, die durch das *category*-Argument angegeben ist:

```
setlocale(LC_ALL, "");
```

In diesem Fall werden alle Kategorien durch die entsprechenden Umgebungsvariablen bestimmt. Wenn die Umgebungsvariable nicht gesetzt ist oder eine leere Zeichenkette enthält, wird die Umgebung ausgewertet (siehe auch Abschnitt „Umgebungsvariablen“ auf Seite 71).

Vordefinierte Lokalitäten

In der C-Laufzeitbibiliothek sind folgende Lokalitäten vordefiniert:

Lokalität	für BS2000-Funktionalität	für XPG4 Version 2-Funktionalität
POSIX	x	x
C	x	x
GERMANY	x	-
V1CTYPE	x	-
De.EDF04F	x	
De.EDF04F@euro	x	
V2CTYPE	x	-

Die vordefinierten Lokalitäten werden einem Programmodul beim Binden hinzugefügt. Bei einem `setlocale`-Aufruf wird ein Zugriffszeiger auf die angegebene Lokalität gesetzt. Sie ist damit die aktuelle Lokalität für den Prozess.

Lokalitätsdateien

Im POSIX-Dateisystem sind die Lokalitäten, die für die XPG4 Version 2-Funktionalität vordefiniert sind, im Dateiverzeichnis `/usr/lib/locale` abgelegt und zwar gemäß folgender Konvention: `/usr/lib/locale/`*locale*`/`*category*.

POSIX- oder C-Lokalität

Alle XPG4 Version 2-konformen Systeme unterstützen die POSIX-Lokalität, die auch als C-Lokalität bekannt ist. Für C-Programme ist die POSIX-Lokalität bei Programmstart die voreingestellte Lokalität, wenn `setlocale()` nicht aufgerufen wird.

Es gibt die POSIX-Lokalitäten `C`, `De`, `De.EDF04F`, `De_DE.EDF04`, `De.EDF04@euro`, `De_DE.EDF04@EU`, `En_US.EDF04` oder `POSIX`. Für die POSIX-Lokalität sind die einzelnen Kategorien wie folgt definiert:

`LC_COLLATE` Die Sortierreihenfolge entspricht für die in der Tabelle auf Seite 47 angegebenen Zeichen der dort angegebenen Reihenfolge. Dies betrifft nur die Funktionen `strcoll()` und `strxfrm()`.

LC_CTYPE Die Klassifizierung entspricht der EBCDIC-Definition der einzelnen Zeichen (EBCDIC.DF.03-IRV, internationale Version).

LC_NUMERIC Die in `localeconv()` definierten Komponenten haben folgende Werte:

localeconv-Komponente	**Wert der POSIX-Lokalität**
`decimal_point`	"<period>"
`thousands_sep`	""
`grouping`	""

LC_MESSAGES Die in `langinfo.h` definierten Konstanten haben folgende Werte:

langinfo-Konstante	**Wert**
`YESEXPR`	`"^[yY]"`
`NOEXPR`	`"^[nN]"`
`YESSTR` Wird zukünftig vom X/Open-Standard nicht mehr unterstützt.	`"yes"`
`NOSTR` Wird zukünftig vom X/Open-Standard nicht mehr unterstützt.	`"no"`

LC_MONETARY Die in `localeconv()` definierten Komponenten haben folgende Werte:

localeconv-Komponente	**Wert**
`int_curr_symbol`	""
`currency_symbol`	""
`mon_decimal_point`	""
`mon_thousands_sep`	""
`mon_grouping`	""
`positive_sign`	""
`negative_sign`	""
`int_frac_digits`	`{CHAR_MAX}`
`frac_digits`	`{CHAR_MAX}`
`p_cs_precedes`	`{CHAR_MAX}`
`n_cs_precedes`	`{CHAR_MAX}`
`p_sep_by_space`	`{CHAR_MAX}`
`n_sep_by_space`	`{CHAR_MAX}`
`p_sign_pos`	`{CHAR_MAX}`
`n_sign_pos`	`{CHAR_MAX}`

LC_TIME

Die in `langinfo.h` definierten Konstanten haben folgende Werte:

langinfo-Konstante	**Wert**
D_T_FMT	"%a %b %e %H:%M:%S %Y"
D_FMT	"%m/%d/%y"
T_FMT	"%H:%M:%S"
AM_STR	"AM"
PM_STR	"PM"
T_FMT_AMPM	"%I:%M:%S %p"
DAY_1	"Sunday"
DAY_2	"Monday"
DAY_3	"Tuesday"
DAY_4	"Wednesday"
DAY_5	"Thursday"
DAY_6	"Friday"
DAY_7	"Saturday"
ABDAY_1	"Sun"
ABDAY_2	"Mon"
ABDAY_3	"Tue"
ABDAY_4	"Wed"
ABDAY_5	"Thu"
ABDAY_6	"Fri"
ABDAY_7	"Sat"
MON_1	"January"
MON_2	"February"
MON_3	"March"
MON_4	"April"
MON_5	"May"
MON_6	"June"
MON_7	"July"
MON_8	"August"
MON_9	"September"
MON_10	"October"
MON_11	"November"
MON_12	"December"

langinfo-Konstante	Wert
ABMON_1	"Jan"
ABMON_2	"Feb"
ABMON_3	"Mar"
ABMON_4	"Apr"
ABMON_5	"May"
ABMON_6	"Jun"
ABMON_7	"Jul"
ABMON_8	"Aug"
ABMON_9	"Sep"
ABMON_10	"Oct"
ABMON_11	"Nov"
ABMON_12	"Dec"

V1CTYPE

Diese Lokalität wird mit "V1CTYPE" oder LC_C_V1CTYPE bezeichnet. Sie entspricht im Wesentlichen der Lokalität "C". Es gibt nur bezüglich der Klassifizierung von Zeichen (Kategorie LC_CTYPE) folgende Unterschiede:

In der Lokalität "V1CTYPE" gehören die Zeichen X'8B', X'8C', X'8D' zur Zeichenklasse lower, die Zeichen X'AB', X'AC', X'AD' zur Zeichenklasse upper und die Zeichen X'C0' und X'D0' zur Zeichenklasse punct. In der Lokalität "C" gehören diese Zeichen zur Zeichenklasse cntrl.

V2CTYPE

Diese Lokalität wird mit "V2CTYPE" oder LC_C_V2CTYPE bezeichnet. Sie entspricht im Wesentlichen der Lokalität "C". Es gibt nur bezüglich der Sortierreihenfolge von Zeichen (Kategorie LC_COLLATE) den folgenden Unterschied: Die Sortierreihenfolge entspricht der des EBCDIC-Zeichensatzes.

GERMANY

Für den deutschen Sprachraum steht eine länderspezifische Lokalität zur Verfügung. Diese Lokalität wird mit "GERMANY" oder LC_C_GERMANY bezeichnet. In Abweichung zur POSIX-Lokalität gelten folgende Werte:

`LC_CTYPE` Die Zeichen `ä (X'FB')`, `ö (X'4F')`, `ü (X'FD')`, `ß (X'FF')` gehören zur Zeichenklasse `lower`.
Die Zeichen `Ä (X'BB')`, `Ö (X'BC')` und `Ü (X'BD')` gehören zur Zeichenklasse `upper`.

Beim Umwandeln von Kleinbuchstaben in Großbuchstaben (`toupper()`, `strupper()`) bleibt das Zeichen `ß` (`X'FF'`) unverändert.

`LC_MONETARY` Internationales Währungssymbol (`int_curr_symbol`): `"EUR"`

Lokales Währungssymbol (`currency_symbol`): `"€"`

Dezimalpunkt (`mon_decimal_point`): `","`

`LC_TIME` Für Wochentags- und Monatsnamen wird die deutsche Sprache verwendet.

Die Datumsdarstellung entspricht den im deutschen Sprachraum üblichen Konventionen:

wochentagsname, *tag*. *monatsname jahr*

Beispiel:

```
Donnerstag, 25. Juli 1991
```

De.EDF04F und De.EDF04F@euro

Diese beiden Lokalitäten unterstützen die Bearbeitung von Dateien und Texten, die das Euro-Zeichen enthalten.

Die zugrundeliegenden Konvertierungstabellen sind in beiden Lokalitäten kompatibel auf einen 8-Bit Code erweitert, der auch das Euro-Zeichen enthält. Die Konvertierungstabellen basieren dabei auf dem ASCII-Code ISO 8859-15 bzw. dem EBCDIC-Code EDF04F.

Die beiden Lokalitäten unterscheiden sich nur in der Kategorie LC_MONETARY.

LC_CTYPE

Zu welcher Basisklasse jedes Zeichen gehört, ergibt sich aus der folgenden Tabelle:

Symbolischer Name	Glyphe	Klasse (n)	ASCII	EBCDIC
<NUL>		control	00	00
<SOH>		control	01	01
<STX>		control	02	02
<ETX>		control	03	03
<EOT>		control	04	37
<ENQ>		control	05	2D
<ACK>		control	06	2E
<alert>		control	07	2F
<backspace>		control	08	16
<tab>		control space blank	09	05
<newline>		control space	0A	15
<vertical-tab>		control space	0B	0B
<form-feed>		control space	0C	0C
<carriage-return>		control space	0D	0D
<SO>		control	0E	0E
<SI>		control	0F	0F
<DLE>		control	10	10
<DC1>		control	11	11
<DC2>		control	12	12
<DC3>		control	13	13
<DC4>		control	14	3C
<NAK>		control	15	3D
<SYN>		control	16	32
<ETB>		control	17	26

Symbolischer Name	Glyphe	Klasse (n)	ASCII	EBCDIC
<CAN>		control	18	18
<EM>		control	19	19
<SUB>		control	1A	3F
<ESC>		control	1B	27
<IS4>		control	1C	1C
<IS3>		control	1D	1D
<IS2>		control	1E	1E
<IS1>		control	1F	1F
<space>		space blank	20	40
<exclamation-mark>	!	punct	21	5A
<quotation-mark>	“	punct	22	7F
<number-sign>	#	punct	23	7B
<dollar-sign>	$	punct	24	5B
<percent-sign>	%	punct	25	6C
<ampersand>	&	punct	26	50
<apostrophe>	’	punct	27	7D
<left-parenthesis>	(	punct	28	4D
<right-parenthesis>	)	punct	29	5D
<asterisk>	*	punct	2A	5C
<plus-sign>	+	punct	2B	4E
<comma>	,	punct	2C	6B
<hyphen>	-	punct	2D	60
<period>	.	punct	2E	4B
<slash>	/	punct	2F	61
<zero>	0	digit xdigit	30	F0
<one>	1	digit xdigit	31	F1
<two>	2	digit xdigit	32	F2
<three>	3	digit xdigit	33	F3
<four>	4	digit xdigit	34	F4
<five>	5	digit xdigit	35	F5
<six>	6	digit xdigit	36	F6
<seven>	7	digit xdigit	37	F7
<eight>	8	digit xdigit	38	F8

Symbolischer Name	Glyphe	Klasse (n)	ASCII	EBCDIC
<nine>	9	digit xdigit	39	F9
colon	:	punct	3A	7A
<semicolon>	;	punct	3B	5E
<less-than-sign>	<	punct	3C	4C
<equals-sign>	=	punct	3D	7E
<greater-than-sign>	>	punct	3E	6E
<question-mark>	?	punct	3F	6F
<commercial-at>	@	punct	40	7C
<A>	A	upper xdigit	41	C1
<B>	B	upper xdigit	42	C2
<C>	C	upper xdigit	43	C3
<D>	D	upper xdigit	44	C4
<E>	E	upper xdigit	45	C5
<F>	F	upper xdigit	46	C6
<G>	G	upper	47	C7
<H>	H	upper	48	C8
<I>	I	upper	49	C9
<J>	J	upper	4A	D1
<K>	K	upper	4B	D2
<L>	L	upper	4C	D3
<M>	M	upper	4D	D4
<N>	N	upper	4E	D5
<O>	O	upper	4F	D6
<P>	P	upper	50	D7
<Q>	Q	upper	51	D8
<R>	R	upper	52	D9
<S>	S	upper	53	E2
<T>	T	upper	54	E3
<U>	U	upper	55	E4
<V>	V	upper	56	E5
<W>	W	upper	57	E6
<X>	X	upper	58	E7
<Y>	Y	upper	59	E8

Symbolischer Name	Glyphe	Klasse (n)	ASCII	EBCDIC
<Z>	Z	upper	5A	E9
<left-sqare-bracket>	[	punct	5B	BB
<backslash>	\	punct	5C	BC
<right-sqare-bracket>	]	punct	5D	BD
<circumflex>	^	punct	5E	6A
<underscore>	_	punct	5F	6D
<grave-accent>	`	punct	60	4A
<a>	a	lower xdigit	61	81
<b>	b	lower xdigit	62	82
<c>	c	lower xdigit	63	83
<d>	d	lower xdigit	64	84
<e>	e	lower xdigit	65	85
<f>	f	lower xdigit	66	86
<g>	g	lower	67	87
<h>	h	lower	68	88
<i>	i	lower	69	89
<j>	j	lower	6A	91
<k>	k	lower	6B	92
<l>	l	lower	6C	93
<m>	m	lower	6D	94
<n>	n	lower	6E	95
<o>	o	lower	6F	96
<p>	p	lower	70	97
<q>	q	lower	71	98
<r>	r	lower	72	99
<s>	s	lower	73	A2
<t>	t	lower	74	A3
<u>	u	lower	75	A4
<v>	v	lower	76	A5
<w>	w	lower	77	A6
<x>	x	lower	78	A7
<y>	y	lower	79	A8
<z>	z	lower	7A	A9

Symbolischer Name	Glyphe	Klasse (n)	ASCII	EBCDIC
<left-curly-bracket>	{	punct	7B	FB
<vertical-line>	|	punct	7C	4F
<right-curly-bracket>	}	punct	7D	FD
<tilde>	~	punct	7E	FF
<DEL>	DEL	control	7F	07
<sc00>			80	20
<sc01>			81	21
<sc02>			82	22
<sc03>			83	23
<sc04>			84	24
<sc05>		control	85	25
<sc06>			86	06
<sc07>			87	17
<sc08>			88	28
<sc09>			89	29
<sc0a>			8A	2A
<sc0b>			8B	2B
<sc0c>			8C	2C
<sc0d>			8D	09
<sc0e>			8E	0A
<sc0f>			8F	1B
<sc10>			90	30
<sc11>			91	31
<sc12>			92	1A
<sc13>			93	33
<sc14>			94	34
<sc15>			95	35
<sc16>			96	36
<sc17>			97	08
<sc18>			98	38
<sc19>			99	39
<sc1a>			9A	3A
<sc1b>			9B	3B

Symbolischer Name	Glyphe	Klasse (n)	ASCII	EBCDIC
<sc1c>			9C	04
<sc1d>			9D	14
<sc1e>			9E	3E
<sc1f>			9F	5F
<nbsp>	NBSP		A0	41
<revexcl>	¡	punct	A1	AA
<cent>	¢	punct	A2	B0
<pound>	£	punct	A3	B1
<euro>	€	punct	A4	9F
<yen>	¥	punct	A5	B2
<CARON-S>	Š	upper	A6	D0
<section>	§	punct	A7	B5
<caron-s>	š	lower	A8	79
<copyright>	©	punct	A9	B4
<fem-ord>	ª	punct	AA	9A
<ang_q_l>	«	punct	AB	8A
<not>	¬	punct	AC	BA
<shy>	SHY	punct	AD	CA
<register>	®	punct	AE	AF
<macron>	¯	punct	AF	A1
<degree>	°	punct	B0	90
<plu-min>	±	punct	B1	8F
<sup-two>	²	punct	B2	EA
<sup-three>	³	punct	B3	FA
<CARON-Z>	Ž	upper	B4	BE
<micro>	µ	punct	B5	A0
<pilcrow>	¶	punct	B6	B6
<mid-dot>	·	punct	B7	B3
<caron-z>	ž	lower	B8	9D
<sup-one>	¹	punct	B9	DA
<mas-ord>	º	punct	BA	9B
<ang-q-r>	»	punct	BB	8B
<OE>	Œ	upper	BC	B7

Symbolischer Name	Glyphe	Klasse (n)	ASCII	EBCDIC
<oe>	œ	lower	BD	B8
<DIA-Y>	Ÿ	upper	BE	B9
<revquest>	¿	punct	BF	AB
<GRAVE-A>	À	upper	C0	64
<ACUTE-A>	Á	upper	C1	65
<CIRC-A>	Â	upper	C2	62
<TILDE-A>	Ã	upper	C3	66
<DIA-A>	Ä	upper	C4	63
<RING-A>	Å	upper	C5	67
<AE>	Æ	upper	C6	9E
<CEDIL-C>	Ç	upper	C7	68
<GRAVE-E>	È	upper	C8	74
<ACUTE-E>	É	upper	C9	71
<CIRC-E>	Ê	upper	CA	72
<DIA-E>	Ë	upper	CB	73
<GRAVE-I>	Ì	upper	CC	78
<ACUTE-I>	Í	upper	CD	75
<CIRC-I>	Î	upper	CE	76
<DIA-I>	Ï	upper	CF	77
<ETH>	Ð	upper	D0	AC
<TILDE_N>	Ñ	upper	D1	69
<GRAVE-O>	Ò	upper	D2	ED
<ACUTE-O>	Ó	upper	D3	EE
<CIRC-O>	Ô	upper	D4	EB
<TILDE_O>	Õ	upper	D5	EF
<DIA-O>	Ö	upper	D6	EC
<multiply>	×	punct	D7	BF
<SLASH-O>	Ø	upper	D8	80
<GRAVE-U>	Ù	upper	D9	E0
<ACUTE-U>	Ú	upper	DA	FE
<CIRC-U>	Û	upper	DB	DD
<DIA-U>	Ü	upper	DC	FC
<ACUTE-Y>	Ý	upper	DD	AD

Symbolischer Name	Glyphe	Klasse (n)	ASCII	EBCDIC
<THORN>	Þ	upper	DE	8E
<sharp-s>	ß	lower	DF	59
<grave-a>	à	lower	E0	44
<acute-a>	á	lower	E1	45
<circ-a>	â	lower	E2	42
<tilde-a>	ã	lower	E3	46
<dia-a>	ä	lower	E4	43
<ring-a>	å	lower	E5	47
<ae>	æ	lower	E6	9C
<cedil-c>	ç	lower	E7	48
<grave-e>	è	lower	E8	54
<acute-e>	é	lower	E9	51
<circ-e>	ê	lower	EA	52
<dia-e>	ë	lower	EB	53
<grave-i>	ì	lower	EC	58
<acute-i>	í	lower	ED	55
<circ-i>	î	lower	EE	56
<dia-i>	ï	lower	EF	57
<eth>	ð	lower	F0	8C
<tilde-n>	ñ	lower	F1	49
<grave-o>	ò	lower	F2	CD
<acute-o>	ó	lower	F3	CE
<circ-o>	ô	lower	F4	CB
<tilde-o>	õ	lower	F5	CF
<dia-o>	ö	lower	F6	CC
<divide>	÷	punct	F7	E1
<slash-o>	ø	lower	F8	70
<grave-u>	ù	lower	F9	C0
<acute-u>	ú	lower	FA	DE
<circ-u>	û	lower	FB	DB
<dia-u>	ü	lower	FC	DC
<acute-y>	ý	lower	FD	8D
<thorn>	þ	lower	FE	AE

Symbolischer Name	Glyphe	Klasse (n)	ASCII	EBCDIC
<dia-y>	ÿ	lower	FF	DF

Die weiteren Klassen sind definiert wie folgt:

alpha — Das Zeichen gehört zur Klasse `upper` oder `lower`.

alnum — Das Zeichen gehört zur Klasse `alpha` oder `digit`.

print — Das Zeichen gehört zur Klasse `alnum` oder `punct` oder es handelt sich um das Zeichen <space>.

graph — Das Zeichen gehört zur Klasse `alnum` oder `punct`.

Die Abbildungen `toupper` und `tolower` zeigen das gewohnte Verhalten:
<XYZ> wird zu <xyz> und <xyz> wird zu <XYZ>.

`LC_COLLATE`

Für die Sortierreihenfolge werden analog zu der Implementierung unter UNIX nur die Zeichen des 7-bit-Codes sowie die im Deutschen verwendeten Umlaute berücksichtigt. Die Umlaute werden ihrem Basisvokal gleichgestellt; in ihrem Sekundärgewicht folgen die Umlaute auf den jeweiligen Basisvokal.
Das Zeichen ’ß’ hat den ASCII-Wert X’DF’ (EBCDIC: X’59’).
Ansonsten entspricht die Reihenfolge der des ASCII-Zeichensatzes.

`LC_NUMERIC`

decimal_point: ","
thousend_sep: "."
grouping: 0;0

`LC_TIME`

Für Wochentags- und Monatsnamen wird die deutsche Sprache verwendet.
Die abgekürzten Wochentagsnamen sind: So, Mo, Di, Mi, Do, Fr, Sa.
Die abgekürzten Monatsnamen sind: Jan, Feb, Mär, Apr, Mai, Jun, Jul, Aug, Sep, Okt, Nov, Dez.

am_pm: "AM", "PM"

Datums- und Zeitdarstellung (%c) `d_t_fmt`: `"%a %d.%h.%Y, %T, %Z"`

Datumsdarstellung (%x) `d_fmt`: `"%d.%m.%y"`

Zeitdarstellung (%X) `t_fmt`: `"%T %Z"`

12-Stunden-Zeitdarstellung (%r) `t_fmt_ampm`: `"%T:%M:%S:%p"`

`time_fmt`: `"%H.%M:%S"`

`day_fmt`: `"&d.%m"`

```
    full_day:"%a %e.%b"
    ar_date:"%b %d %H:%M %Y"
    last_date:"%a %e.%b %H:%M"
    ls_date: "%h %e %H:%M"
    ls_date2:"%h %e %Y"
    ps_date:"%d.%b"
    su_date: "%d.%m %H:%M"
    tar_date:"%e.%b %H:%M %Y"
    diff_date:"%a %e.%b.%Y, %T"
LC_MESSAGES
    yesstring       "ja"
    nostr           "nein"
    quitstr         "abbrechen"
    noexpr          "^[nN]"
    yesexpr         "^[jJ]"
    quitexpr        "^[aA]"
LC_MONETARY
```

Element	De.EDF04F	De.EDF04F@euro
int_curr_symbol	"DEM"	"EUR"
currency_symbol	"DM"	"€"
mon_decimal_point	","	","
mon_thousands_sep	"."	"."
mon_grouping	3;3	3;3
positive_sign	""	""
negativ_sign	"-"	"-"
int_frac_digits	2	2
frac_digits	2	2
p_cs_precedes	0	0
p_sep_by_space	1	1
n_cs_precedes	0	0
n_sep_by_space	1	1
p_sign_posn	1	1
n_sign_posn	1	1

Benutzerspezifische Lokalitäten

Der Benutzer kann eigene Lokalitäten definieren.

Dazu stellt die Bibliothek $.SYSLNK.CRTE zwei Quellprogrammelemente (Typ S) mit den Namen USLOCC und USLOCA bereit. USLOCC ist ein C-Quellprogramm, USLOCA ist ein Assembler-Quellprogramm. Die beiden Quellprogramme sind für die Erzeugung von benutzerspezifischen Lokalitäten gleichwertig.

Die Quellprogramme legen die Daten für die einzelnen Lokalitätskategorien fest und sind mit den Daten der POSIX-Lokalität vorbelegt (siehe Abschnitt „POSIX- oder C-Lokalität“ auf Seite 55). Diese Daten können auf die gewünschten Werte geändert werden.

Außerdem sind in den Quellprogrammen folgende Änderungen vorzunehmen:

– In den Quellprogrammen ist eine Adresstabelle mit dem Namen USERLOC definiert. Dieser Name muss auf einen vom Benutzer festzulegenden Namen geändert werden. Dieser Name muss ein gültiger Entryname sein.

– Im C-Quellprogramm braucht dazu nur der Name USERLOC mit einer #define-Anweisung modifiziert zu werden. Im Assembler-Quellprogramm muss der Name USERLOC in der Definitionszeile der Tabelle und in der ENTRY-Anweisung modifiziert werden.

– Der vom Benutzer modifizierte Name wird beim Aufruf von setlocale() als *locale*-Argument zur Kennzeichung der benutzerspezifischen Lokalität verwendet.

Die modifizierten Quellprogramme können mit dem C/C++-Compiler bzw. mit dem Assembler (auch ASSGEN) übersetzt werden. Wird das Modul nicht in der Bibliothek $.SYSLNK.CRTE, sondern in einer anderen PLAM-Bibliothek abgelegt, muss diese Bibliothek vor Start des C-Programms mit folgendem ADD-FILE-LINK-Kommando zugewiesen werden:

```
/ADD-FILE-LINK LINK-NAME=IC@LOCAL,FILE-NAME=bibliothek
```

Umgebungsvariablen

Die Umgebungsvariablen - auch Shell-Variablen genannt -, die in diesem Abschnitt beschrieben werden, beeinflussen das Verhalten von Kommandos, Funktionen und Anwendungen. Es gibt Umgebungsvariablen, die nur für die Kommandos von Interesse sind (siehe Handbuch „POSIX Kommandos (BS2000/OSD)“ unter dem Stichwort „Shell-Variable“). Wenn ein Prozess mit der Ausführung beginnt, stellen die `exec`-Funktionen einen Vektor von Zeichenketten zur Verfügung, der **Umgebung** genannt wird (siehe `exec`). Auf diesen Vektor wird durch die folgende externe Variable gezeigt:

```
extern char **environ;
```

Entsprechend dem XPG4 Version 2-Standard haben diese Zeichenketten die Form "*name*=*value*", z.B. "`PATH=/sbin:/usr/sbin`".

Anwendungen können eigene Umgebungsvariablen definieren, wobei die Namenskonventionen eingehalten werden müssen (siehe Handbuch „POSIX Kommandos (BS2000/OSD).

Vorbesetzung der Umgebungsvariablen vom BS2000/OSD aus

Sie haben die Möglichkeit, Umgebungsvariablen vom BS2000 aus vorzubesetzen, indem Sie eine SDF-P-Variable mit dem Namen `SYSPOSIX` als Struktur definieren (siehe Handbuch „SDF-P. Programmieren in der Kommandosprache“). Wenn die Variable `SYSPOSIX.`*name* den Wert *value* hat, wird die Zeichenkette "*name*=*value*" in den globalen Datenbereich des Programms geschrieben, wobei jedoch nur Variablen vom Typ String berücksichtigt werden.

Die SDF-P-Variablenstruktur kann über den Parameter `Scope` als Prozedur oder Task deklariert werden. Taskvariablen werden immer gefunden, Prozedurvariablen überschreiben die Taskvariablen eventuell.

Auf der BS2000-Kommandoebene können nur Großbuchstaben für Variablennamen verwendet werden. Bindestriche in Namen der SDF-P-Variablen werden in Unterstriche umgewandelt. Aus `SYSPOSIX.LC-`*NAME* wird also z.B. die Zeichenkette "`LC_`*NAME*`=...`".

Umgebungsvariablen für die Internationalisierung

Ein internationalisiertes Programm macht keine festen Annahmen über die Umgebung, in der es abläuft. Es ermittelt die konkrete Umgebung, in der es abläuft, aus Umgebungsvariablen.

So wird z.B. die Umgebung für die Darstellung von Ausgaben aus den Umgebungsvariablen `LANG` und `LC_`*xxx* ermittelt, während die Funktionen zur Bearbeitung von Meldungskatalogen die Umgebungsvariable `NLSPATH` auswerten.

Folgende Umgebungsvariablen für die Internationalisierung werden unterstützt:

`LANG` Bestimmt die Lokalität für sprach- und kulturspezifische Eigenheiten und den kodierten Zeichensatz, wenn `LC_ALL` oder andere Umgebungsvariablen für die Lokalität nicht definiert sind. `LANG` kann von Anwendungen z.B. für das Format von Fehlermeldungen, Sortierreihenfolgen oder Datumsdarstellungen genutzt werden. Der Wert der Umgebungsvariablen hat folgendes Format:

`LANG=`*sprache*`[_`*gebiet*`[.`*zeichensatz*`]]`

Ein Benutzer aus Österreich, der Deutsch spricht und ein Terminal mit dem ISO 8859/1-Zeichensatz verwendet, stellt die Variable `LANG` auf den folgenden Wert:

```
LANG=De_A.88591
```

Auf diese Weise sollte es einem Benutzer möglich sein, entsprechende Meldungskataloge zu finden, vorausgesetzt, sie existieren.

Besondere Sprach-Operationen werden zur Laufzeit durch den Aufruf von `setlocale()` initialisiert. Normalerweise werden die Sprach-Anforderungen des Benutzers, wie durch die Umgebungsvariable `LANG` angegeben, durch den nachfolgenden Aufruf von `setlocale()` zur internationalen Umgebung des Programms gebunden:

```
setlocale (LC_ALL, "");
```

`LC_ALL` Auf X/Open-Systemen ist diese Form eines Aufrufs von `setlocale()` definiert, um die internationale Umgebung des Programms aus den zugehörigen Umgebungsvariablen zu initialisieren. `LC_ALL` spricht die gesamte internationale Umgebung des Programms an und `LANG` stellt die notwendigen Voreinstellungen zur Verfügung, wenn eine oder mehrere der kategorie-spezifischen Variablen nicht gesetzt oder gleich der leeren Zeichenkette sind.

`LC_COLLATE` Diese Kategorie gibt die verwendete Sortierfolge an. Die diesbezüglichen Informationen werden in einer Datenbank gespeichert, die von dem Kommando `colltbl()` erzeugt wird. Diese Umgebungsvariable beeinflusst `strcoll()` und `strxfrm()`.

`LC_CTYPE` Diese Kategorie legt die Klassifikation von Zeichen, die Umwandlung von Zeichen und die Größe von Multibyte-Zeichen fest. Die diesbezüglichen Informationen werden in einer Datenbank gespeichert, die von dem Kommando `chrtbl()` erzeugt wird. Die Standardvereinbarung für `C` entspricht der 7-Bit-Zeichenmenge. Diese Umgebungsvariable wird von `ctype()`, `mbchar()` und vielen Kommandos verwendet, z.B. die Kommandos `cat`, `ed`, `ls` und `vi`.

`LC_MESSAGES` Diese Kategorie gibt die Sprache an, die vom Meldungskatalog verwendet wird. Zum Beispiel kann eine Anwendung je einen Meldungskatalog mit französischen Meldungen und mit deutschen Meldungen haben.

`LC_MONETARY` Diese Kategorie spezifiziert die Währungssymbole und Trennzeichen für eine bestimmte Umgebung. Diese Umgebungsvariable wird von `localeconv()` verwendet.

`LC_NUMERIC` Diese Kategorie gibt die Trennzeichen für Dezimalstellen und Tausenderstellen an. Diese Umgebungsvariable wird von `localeconv()`, `printf()` und `strtod()` verwendet.

`LC_TIME` Diese Kategorie spezifiziert Datums- und Zeitformate.

`NLSPATH` Die Umgebungsvariable `NLSPATH` liefert sowohl die Position von Meldungskatalogen in der Form eines Suchpfads, als auch die Namenskonventionen, die mit den Meldungskatalogen verknüpft sind. Zum Beispiel:

```
NLSPATH=/nlslib/%L/%N.cat:/nlslib/%N/%L
```

Das Metazeichen `%` zeigt ein Substitutionsfeld an, wobei `%L` die aktuelle Einstellung der Umgebungsvariablen `LANG` (siehe unten) ersetzt und `%N` den Wert des Parameters *name* ersetzt, der an `catopen()` übergeben wird. Im obigen Beispiel sucht `catopen()` daher zuerst in `/nlslib/$LANG/`*name*`.cat` und dann in `/nlslib/`*name*`/$LANG` nach dem angegebenen Meldungskatalog.

`NLSPATH` wird üblicherweise systemweit eingestellt (z.B. in `/etc/profile`) und macht daher die Positionierungs- und Namenskonventionen für die Meldungskataloge sowohl für die Programme als auch für die Benutzer transparent.

Der komplette Satz der Metazeichen umfasst folgende Symbole:

Metazeichen	Bedeutung
`%N`	Wert des Namensparameters, der an `catopen()` übergeben wird
`%L`	Wert von `LANG`
`%l`	Wert des Sprachenelements aus `LANG`
`%t`	Wert des Landelements aus `LANG`
`%c`	Wert des Zeichensatzelements aus `LANG`
`%%`	das Zeichen %

Das Verhalten der Sprachinformations-Funktion `nl_langinfo()` wird ebenfalls durch die Belegungen dieser Umgebungsvariablen beeinflusst (siehe auch `langinfo.h`).

LC_COLLATE, LC_CTYPE, LC_MONETARY, LC_NUMERIC und LC_TIME sind so definiert, dass sie ein zusätzliches Feld *@modifikator* akzeptieren, welches es dem Benutzer erlaubt, einen besonderen Fall eines Umgebungsdatums innerhalb einer speziellen Kategorie auszuwählen (zum Beispiel, um das Wörterbuch entgegengesetzt zur Sortierreihenfolge der Zeichen zu definieren). Die Syntax dieser Umgebungsvariablen lautet daher:

[*sprache*[*_gebiet*[*.zeichensatz*]][*@modifikator*]]

Wenn zum Beispiel ein Benutzer mit dem System in französisch kommunizieren will, aber deutsche Textdateien sortieren muss, so könnten LANG und LC_COLLATE möglicherweise folgendermaßen definiert sein:

```
LANG=Fr_FR
LC_COLLATE=De_DE
```

Dies könnte noch erweitert werden, um zum Beispiel die Wörterbuch-Sortierung durch die Verwendung des Felds *@modifikator* auszuwählen:

```
LC_COLLATE=De_DE@dict
```

Zur Laufzeit werden diese Werte zur internationalen Umgebung des Programms gebunden, indem setlocale() aufgerufen wird.

Dateibearbeitung

Mit den C-Bibliotheksfunktionen kann bei Verwendung der POSIX-Funktionalität prinzipiell sowohl auf POSIX- bzw. UFS-Dateien als auch auf BS2000-Dateien zugegriffen werden. Die Übersetzungsumgebung stellt ferner explizite 64-Bit-Funktionen und Typen zusätzlich zu den 32-Bit-Funktionen und Typen zur Verfügung. Die Verwendung der 64-Bit-Schnittstelle ist erforderlich, um Dateien > 2 GB via NFS bearbeiten zu können. Mit NFS V3.0 können Dateisysteme bearbeitet werden, die Dateien enthalten, die größer sind als 2 GB. Siehe hierzu auch Abschnitt „Umfang der unterstützten C-Bibliothek“ auf Seite 19.

Im Folgenden werden die Dateibearbeitungs-Funktionen in unterschiedliche Gruppen eingeteilt, je nachdem, ob sie sowohl POSIX- als auch BS2000-Dateien oder nur POSIX-Dateien bearbeiten können. Funktionen, die nur POSIX-Dateien bearbeiten, setzen explizit errno, wenn statt einer POSIX-Datei eine BS2000-Datei angegeben wurde.

Im Anschluss an diese Funktionsgruppierung folgt eine Beschreibung der POSIX-Dateibearbeitung. Die Besonderheiten der BS2000-Dateibearbeitung werden weiter unten beschrieben.

Funktionen für POSIX- und BS2000-Dateien

Mit den C-Bibliotheksfunktionen der folgenden Tabelle können sowohl POSIX- als auch BS2000-Dateien bearbeitet werden.

btowc	fstatvfs()	open()	vprintf()
creat()	ftell()	perror()	vswprintf
clearerr()	ftruncate()	printf()	vwfprintf
close()	fwide	putc()	wcrtomb
creat()	fwprinft	putchar()	wcscat
fclose()	fwscanf	puts()	wcschr
fcntl()	fwrite()	putw()	wcscmp
fdopen()	getc()	putwc()	wcscoll
feof()	getchar()	putwchar()	wcscpy
ferror()	getdents()	read()	wcscspn
fflush()	getrlimit()	readdir()	wcsftime
fgetc()	gets()	remove()	wcscat
fgetwc	getw()	rename()	wcslen
fgetws	getwc()	rewind()	wcsncat
fgetpos()	getwchar()	scanf()	wcsncmp
fgets()	iswalnum	setbuf()	wcsncpy
fgetwc()	iswalpha	setrlimit()	wcspbrk
fgetws()	iswcntrl	setvbuf()	wcsrchr
fileno()	iswctype	swscanf	wcsrtombs
fopen()	iswdigit	swprintf	wcsspn
freopen()	iswgraph	stat()	wcsstr
fprintf()	iswlower	statvfs()	wctob
fputc()	iswprint	tmpfile()	wcstod
fputs()	iswpunct	tmpnam()	wcstok
fputwc()	iswspace	towctrans	wcstol
fputws()	iswupper	towlower	wcstoul
fread()	iswxdigit	towupper	wcsxfrm
freopen()	lockf()	truncate()	wctrans
fscanf()	lseek()	ungetc()	wctype
fseek()	lstat()	ungetwc()	write()
fsetpos()	mktemp()	unlink()	
fstat()	mmap()	vfprintf()	

Funktionen, die BS2000-Dateien abweisen

Die folgenden Funktionen bearbeiten nur POSIX-Dateien (siehe auch Handbuch „POSIX Grundlagen (BS2000/OSD)“). Alle diese Funktionen - außer `sync()` - setzen `errno` gleich `EINVAL`, wenn versucht wird, auf BS2000-Dateien zuzugreifen.

`access()`	`fpathconf()`	`symlink()`	`tcgetpgrp()`
`chmod()`	`fsync()`	`sync()`	`tcsendbreak()`
`chown()`	`isatty()`	`sysfs() *)`	`tcsetattr()`
`dup()`	`link()`	`tempnam() *)`	`tcsetpgrp()`
`dup2()`	`mknod()`	`tcdrain()`	`utime()`
`fchmod()`	`mkfifo()`	`tcflow()`	
`fchown()`	`pathconf()`	`tcflush()`	
`fcntl()`	`readlink()`	`tcgetattr()`	

*) `sync()` hat auf BS2000-Dateien keine Wirkung.
`tempnam()` setzt `errno` auf `EINVAL`, wenn `PROGRAM-ENVIRONMENT` nicht gesetzt ist.

Bei Standardströmen funktionieren diese Funktionen mit Einschränkungen, wenn sie vom POSIX-Subsystem mit den BS2000-SYSFILE-Managementdateien verknüpft wurden (siehe nächter Abschnitt „Datenströme“).

Funktionen, die nur auf POSIX-Dateien zugreifen

Die Funktionen der folgenden Liste greifen immer auf POSIX-Dateien zu, unabhängig davon, welche Funktionalität (POSIX oder BS2000) gewählt wurde, denn sie sind nicht als BS2000-Funktionen vorhanden.

`chdir()`	`getpass()`	`popen()`	`telldir()`
`chroot()`	`mkdir()`	`readdir()`	`ttyname()`
`closedir()`	`opendir()`	`rewinddir()`	`umount()`
`ftw()`	`pclose()`	`rmdir()`	`umask()`
`getcwd()`	`pipe()`	`seekdir()`	

Datenströme

Ein Datenstrom wird mit einer externen Datei - das kann auch ein physikalisches Gerät sein - verbunden, wenn eine Datei **geöffnet** wird. Dies ist auch der Fall, wenn eine neue Datei **erzeugt** wird. Wird eine existierende Datei erzeugt, wird der ursprüngliche Inhalt verworfen, wenn es notwendig ist. Wenn eine Datei Positionierungsanforderungen unterstützt (z.B. eine Plattendatei, aber kein Terminal), wird ein **Lese-/Schreibzeiger**, der mit dem Datenstrom verbunden ist, an den Anfang der Datei (Bytenummer Null) positioniert, solange die Datei nicht im Anfügemodus geöffnet wurde. In letzterem Fall wird der Lese-/Schreib-

zeiger entweder an den Anfang oder an das Ende der Datei positioniert. Der Lese-/Schreibzeiger erleichtert nachfolgenden Lese-, Schreib- und Positionierungsanforderungen ein Positionieren in der Datei. Alle Eingaben werden behandelt, als ob durch aufeinander folgende `fgetc`-Aufrufe Bytes gelesen werden würden. Alle Ausgaben werden behandelt, als ob durch aufeinander folgende `fputc`-Aufrufe Bytes geschrieben werden würden.

Pufferung von Datenströmen

Wenn ein Datenstrom **nicht gepuffert** ist, werden die Bytes direkt ans System durchgereicht. Andernfalls können die Bytes als Block akkumuliert und übertragen werden. Wenn ein Datenstrom **voll gepuffert** ist, werden die Bytes als Block übertragen, sobald der Puffer voll ist. Wenn ein Datenstrom **zeilenweise gepuffert** ist, werden die Bytes als Block übertragen, sobald ein Neue-Zeile-Zeichen auftritt. Des Weiteren werden Bytes als Block übertragen, wenn ein Puffer voll ist und Eingaben von einem nicht gepufferten Datenstrom oder einem zeilenweise gepufferten Datenstrom, der die Byte-Übertragung erfordert, angefordert werden. Die Unterstützung dieser Charakteristika kann durch `setbuf()` und `setvbuf()` veranlasst werden.

Datei und Datenstrom trennen

Eine Datei kann von einem steuernden Datenstrom getrennt werden, indem sie **geschlossen** wird. Ausgabeströme werden **geleert**, d.h.alle noch nicht geschriebenen Pufferinhalte werden übertragen, bevor der Datenstrom von der Datei getrennt wird. Der Wert eines Zeigers auf ein `FILE`-Objekt ist nach dem Schließen der Datei, einschließlich der Standard-Datenströme, nicht definiert.

Eine Datei kann anschließend vom selben oder einem anderen Programm wieder geöffnet und ihre Inhalte verändert werden, wenn der Lese-/Schreibzeiger an den Anfang positioniert werden kann. Wenn die `main`-Funktion zu ihrem ursprünglichen Aufruf zurückkehrt oder die `exit`-Funktion aufgerufen wird, werden alle Ausgabeströme geleert und alle offenen Dateien geschlossen, bevor das Programm beendet wird. Andere Methoden der Programmbeendigung, wie z.B. ein `abort`-Aufruf, schließen offene Dateien nicht zuverlässig.

Die Adresse des `FILE`-Objekts, die verwendet wird, um einen Datenstrom zu steuern, kann signifikant sein; die Kopie eines `FILE`-Objekts muss nicht unbedingt auf den Speicherplatz des Originals zugreifen.

Standard-Ein-/Ausgabeströme

Beim Programmstart sind drei Datenströme vordefiniert, die also nicht explizit geöffnet werden müssen:

- **Standard-Eingabe**, um konventionelle Eingaben zu lesen
- **Standard-Ausgabe**, um konventionelle Ausgaben zu schreiben
- **Standard-Fehlerausgabe**, um Diagnoseausgaben zu schreiben

Der geöffnete Standard-Fehlerausgabestrom ist nicht voll gepuffert. Die Standard-Ein/-Ausgabeströme sind nur dann voll gepuffert, wenn der Strom nicht mit einem interaktiven Gerät in Verbindung steht. Ansonsten wird zeilenweise gepuffert.

Die Standard-Ein-/Ausgabeströme werden je nach Funktionalität (siehe Abschnitt „Wahl des Dateisystems und der Systemumgebung“ auf Seite 42) mit POSIX- oder BS2000-Dateien verbunden.

Wenn auf das DVS zugegriffen wird, wird folgende Beziehung hergestellt:

`stdin`	SYSDTA
`stdout`, `stderr`	SYSOUT

In diesem Fall ist das Verhalten kompatibel zu vorhergehenden Versionen der C-Laufzeitbibiliothek (siehe auch Abschnitt „BS2000-Systemdateien“ auf Seite 82).

Funktionen, die nur POSIX-Funktionalität verwenden, können in diesem Fall nicht auf `stdin`, `stdout` oder `stderr` angewendet werden.

Wenn auf das POSIX-Dateisystem zugegriffen wird, werden die Standard-Ein-/Ausgabe-Ströme in der Regel mit `/dev/tty` verbunden (siehe auch Abschnitt „Verknüpfung der Ein-/Ausgabeströme“ auf Seite 42).

Im Batch-Modus wird in jedem Fall mit SYSFILE verknüpft, da kein Terminal vorhanden ist. In Sohnprozessen kann auf Ein-/Ausgabe-Ströme, die mit SYSFILE verknüpft wurden, nicht mehr zugegriffen werden, auch wenn die Verknüpfung über POSIX erfolgt ist.

Wird die Verknüpfung der Standard-Ein-/Ausgabeströme durch die Wahl der POSIX-Funktionalität mit der Umgebungsvariablen gesteuert, kann man die Verknüpfung durch Veränderung der Variablen mit `putenv()` beeinflussen: Wird mit einer `exec`-Funktion ein Programm neu gestartet, werden bei der C-Laufzeitinitialisierung die Umgebungsvariablen neu ausgewertet und mit der `exec`-Funktion gestarteten Programm entsprechend neu verknüpft.

Interaktion von Dateideskriptoren und Datenströmen

Auf eine Dateibeschreibung kann über einen Dateideskriptor zugegriffen werden, der durch `open()` oder `pipe()` erzeugt worden ist oder über einen Datenstrom, der durch `fopen()` oder `popen()` erzeugt worden ist. Sowohl ein Dateideskriptor als auch ein Datenstrom wird ein **Verweis** auf die Dateibeschreibung genannt; auf eine Dateibeschreibung können mehrere Verweise zeigen.

Verweise können durch konkrete Benutzeraktionen erzeugt oder gelöscht werden, ohne die zu Grunde liegende Dateibeschreibung zu beeinflussen. Funktionen, die solche Verweise erzeugen, sind z.B. `fcntl()`, `dup()`, `fdopen()`, `fileno()` und `fork()`. Diese Verweise können zumindest mit den Funktionen `fclose()`, `close()` und den `exec`-Funktionen wieder gelöscht werden.

Wird ein Dateideskriptor niemals in einer Operation verwendet, welche die Dateiposition beeinflusst (d.h. `read()`, `write()` oder `lseek()`), so gilt dieser Dateideskriptor nicht als Verweis. Er kann aber zu einem solchen werden, z.B. als Ergebnis von `fdopen()`, `dup()` oder `fork()`. Ein Dateideskriptor, der einem Datenstrom zu Grunde liegt, ist niemals eine solche Ausnahme, gleichgültig ob er durch `fopen()` oder `fdopen()` erzeugt wurde, solange er nicht direkt von der Anwendung benutzt wird, um die Dateiposition zu beeinflussen. `read()` und `write()` beeinflussen die Dateiposition implizit; `lseek()` beeinflusst sie explizit.

Das Ergebnis von Funktionsaufrufen, die nur mit einem Verweis arbeiten (dem **aktiven Verweis**), kann im Nachschlageteil nachgelesen werden. Wenn jedoch zwei oder mehr Verweise benutzt werden und einer davon ein Datenstrom ist, werden deren Aktionen so koordiniert, wie dies im folgenden Abschnitt „Aktionen“ beschrieben wird.

Ein Verweis, der ein Datenstrom ist, gilt dann als geschlossen, wenn entweder die Funktion `fclose()` oder die Funktion `freopen()` für diesen ausgeführt wird (das Ergebnis von `freopen()` ist dann ein neuer Datenstrom, der kein Verweis auf dieselbe Dateibeschreibung sein kann, auf die sein vorheriger Wert verwiesen hat), oder wenn der Prozess, zu dem dieser Datenstrom gehört, mit `exit()` oder `abort()` beendet wird. Ein Dateideskriptor wird durch `close()`, `_exit()` oder eine der `exec`-Funktionen mit für diesen Dateideskriptor gesetztem `FD_CLOEXEC`-Bit geschlossen.

Damit ein Verweis zum aktiven Verweis wird, müssen zwischen der letzten Verwendung des ersten, zurzeit aktiven Verweises, und der ersten Verwendung des zweiten, zukünftig aktiven Verweises die unten beschriebenen Aktionen erfolgen. Dadurch wird der zweite Verweis zum aktiven Verweis. Alle die Dateiposition für den ersten Verweis beeinflussenden Aktivitäten der Anwendung müssen solange unterbunden werden, bis dieser wieder der aktive Verweis ist. Für eine Funktion zur Bearbeitung eines Datenstroms, die eine zu Grunde deliegende Funktion aufruft, die ihrerseits die Position in der Datei verändert, wird angenommen, dass die aufrufende Funktion zur Bearbeitung des Datenstroms selbst die Dateiposition verändert. Die jeweils zu Grunde liegenden Funktionen werden unten beschrieben.

Die Verweise müssen nicht im selben Prozess vorhanden sein, damit diese Regeln Anwendung finden.

Aktionen

Wenn nach der Ausführung der jeweiligen Aktionen der Verweis noch offen ist, kann ihn die Anwendung schließen.

- Wenn eine der folgenden Bedingungen erfüllt ist, ist für den **ersten Verweis** keine Aktion notwendig:
  - Der Verweis ist ein Dateideskriptor oder ein ungepufferter Datenstrom.
  - Die einzige weitere auszuführende Aktion für einen Verweis ist das Schließen.
  - Der Verweis ist ein zeilengepufferter Datenstrom und die letzte Aktion hat denselben Effekt auf die zugehörige Datei wie `fputs()`.
  - Der Verweis ist ein zum Lesen geöffneter Datenstrom und `feof()` liefert TRUE.
- Wenn keine der oben aufgelisteten Bedingungen zutrifft, muss in folgenden Fällen `fflush()` ausgeführt oder der Datenstrom geschlossen werden:
  - Wenn es sich um einen zum Schreiben oder Anfügen geöffneten Datenstrom handelt, der nicht gleichzeitig zum Lesen geöffnet ist.
  - Wenn der Datenstrom auf eine Art geöffnet ist, die das Lesen gestattet, und wenn die zu Grunde liegende Dateibeschreibung auf ein Gerät verweist, das positionieren kann.
- In allen anderen Fällen ist das Ergebnis undefiniert.

Für den **zweiten** Verweis gilt Folgendes:

Wenn ein vorher aktiver Verweis von einer Funktion verwendet wurde, welche die Dateiposition ausdrücklich veränderte, außer wie oben für den ersten Verweis benötigt, dann muss die Anwendung, je nach Art des Verweises, eine der Funktionen `lseek()` oder `fseek()` ausführen, um an die entsprechende Position zu positionieren.

Wenn der aktive Verweis aufhört, zugreifbar zu sein, bevor die Anforderungen für den ersten Verweis erfüllt sind, dann geht die Dateibeschreibung in einen undefinierten Zustand über. Dies kann dann der Fall sein, wenn eine Funktion `fork()` oder `exit()` ausgeführt wird.

Die `exec`-Funktionen sorgen dafür, dass auf alle Datenströme, die zum Zeitpunkt ihres Aufrufs offen sind, nicht mehr zugegriffen werden kann, gleichgültig, welche Datenströme oder Dateideskriptoren für das Speicherabbild des neuen Prozesses zur Verfügung stehen.

Die C-Laufzeitbibiliothek stellt sicher, dass eine Anwendung, auch wenn sie aus mehreren Prozessen besteht, stets korrekte Ergebnisse liefert, d.h. dass beim Schreiben keine Daten verlorengehen oder doppelt geschrieben werden, dass alle Daten in der richtigen Reihenfolge geschrieben werden (außer bei einer entsprechenden Änderung durch das Positionieren) und dass beim sequenziellen Lesen alle Daten gefunden werden, sofern nach den oben angeführten Regeln vorgegangen wird. Dabei spielt es keine Rolle, in welcher Reihenfolge die Verweise verwendet werden. Werden die oben aufgeführten Regeln nicht befolgt, dann ist das Ergebnis undefiniert.

Siehe auch Handbuch „POSIX Grundlagen (BS2000/OSD)“.

Unterstützung von Dateisystemen in ASCII

Es können Dateisysteme in das POSIX-Dateisystem gemountet werden, die auf Rechnern liegen, die üblicherweise nicht mit dem EBCDIC, sondern mit dem ASCII-Zeichensatz arbeiten. Um diese Interaktion zu vereinfachen, wird bei Textdateien in der C-Bibliothek automatisch konvertiert.

Damit automatisch konvertiert wird, müssen folgende Voraussetzungen erfüllt sein:

– Die Datei muss mit `fopen()`, `fdopen()` oder `freopen()` geöffnet worden und damit mit einem Datenstrom verbunden sein.

– Der Modus „b“ für binär darf nicht angegeben sein.

– `fstat()` liefert nicht das Bit BS2000-Dateisystem (.

– Die Umgebungsvariable `IO_CONVERSION` hat den Wert "`YES`".

Für die Fälle, bei denen die automatische Konvertierung nicht greift, werden die Funktionen `ascii_to_ebcdic()` und `ebcdic_to_ascii()` zur Verfügung gestellt.

BS2000-Dateibearbeitung

Mit den Ein-/Ausgabefunktionen von CRTE können außer POSIX-Dateien auch folgende Dateiarten verarbeitet werden:

– die BS2000-Systemdateien SYSDTA, SYSOUT und SYSLST,

– katalogisierte Plattendateien mit den Zugriffsmethoden SAM, ISAM und PAM,

– temporäre PAM-Dateien (INCORE).

Im C-BS2000 wird einerseits zwischen Binär- und Textdateien unterschieden, andererseits zwischen strom- und satzorientierter Ein-/Ausgabe.

Folgende Tabelle zeigt die möglichen Kombinationen, in denen die verschiedenen Dateiarten verarbeitet werden können:

	Textdatei Strom-Ein-/Ausgabe	**Binärdatei Strom-Ein-/Ausgabe**	**Binärdatei Satz-Ein-/Ausgabe**
Systemdateien	X		
INCORE		X	
SAM	X	X	X
ISAM	X		X
PAM		X	X

Es können (inkl. `stdin`, `stdout` und `stderr`) maximal 256 Dateien gleichzeitig geöffnet sein.

BS2000-Systemdateien

Den Datenströmen entsprechen im BS2000 die Systemdateien. Deren Funktionalität ist relevant, wenn eine Funktion mit BS2000-Funktionalität aufgerufen wurde.

SYSDTA

Ein C-Programm kann SYSDTA folgendermaßen verwenden:

– Mit einer Öffnungsfunktion (`fopen()`, `freopen()`, `open()`) wird eine Datei mit dem Namen `"(SYSDTA)"` oder `"(SYSTERM)"` zum Lesen geöffnet. Der von der Öffnungsfunktion gelieferte Dateizeiger dient dann als Argument einer anschließenden Eingabefunktion.

  Beispiel

  ```
  FILE *fp;
  fp = fopen("(SYSDTA)", "r");
  fgetc(fp);
  ```

– Bei Eingabefunktionen wird als Dateiargument der Dateizeiger `stdin` bzw. der Dateideskriptor 0 angegeben.

  Beispiele

  ```
  fgetc(stdin);
  read(0, buf, n);
  ```

– Es werden Eingabefunktionen benutzt, die standardmäßig von `stdin` lesen (z.B. `scanf()`, `getchar()`, `gets()`).

Soll die Eingabe nicht vom Terminal aus erfolgen, sondern aus einer katalogisierten Datei, kann dies auf zweierlei Weise geschehen:

1. Wurde mit PARAMETER-PROMPTING=YES (in der Compiler-Option RUNTIME-OPTIONS) eine Parameterzeile angefordert, kann in dieser Parameterzeile die Standardeingabe (Dateizeiger stdin bzw. Dateideskriptor 0) auf eine katalogisierte Datei umgewiesen werden (siehe auch C- und C++-Benutzerhandbücher).

   Diese Umlenkung wirkt sich **nicht** auf Dateien aus, die mit dem Namen "(SYSDTA)" bzw. "(SYSTERM)" geöffnet werden. Die Eingabe aus Dateien dieses Namens wird nach wie vor vom Terminal erwartet.

2. Vor Programmstart mit dem Kommando ASSIGN-SYSDTA *dateiname*.

   Bei allen Eingabefunktionen werden die Eingabedaten dann aus der zugewiesenen Datei erwartet.

   Bei der Zuweisung mit dem ASSIGN-SYSDTA-Kommando ist Folgendes zu beachten:

   – Nach Programmablauf steht der interne Satzzeiger hinter dem zuletzt gelesenen Satz bzw. auf Dateiende. Soll die Datei in einem weiteren Programmlauf wieder ab Dateianfang eingelesen werden, muss vor dem Programmstart ein neues ASSIGN-SYSDTA-Kommando abgesetzt werden.

   – Wurde PARAMETER-PROMPTING=YES (in der RUNTIME-OPTIONS-Option) gewählt, so wird der erste Satz der zugewiesenen Datei als Parameterzeile für die main-Funktion interpretiert.

Hinweis

Ist im C-Programm kein anderes Endekriterium vereinbart, lässt sich die EOF-Bedingung bei Eingaben am Terminal folgendermaßen erreichen: K2-Taste drücken und die Kommandos EOF und RESUME-PROGRAM eingeben.

SYSOUT

Ein C-Programm kann SYSOUT folgendermaßen verwenden:

– Mit einer Öffnungsfunktion (fopen(), freopen(), open()) wird eine Datei mit dem Namen "(SYSOUT)" oder "(SYSTERM)" zum Schreiben geöffnet. Der von der Öffnungsfunktion gelieferte Dateizeiger dient dann als Argument einer anschließenden Ausgabefunktion.

  Beispiel

```
FILE *fp;
fp = fopen("(SYSTERM)", "w");
fputc(fp);
```

– Bei Ausgabefunktionen wird als Dateiargument der Dateizeiger `stdout` bzw. der Dateideskriptor 1 angegeben.

  Beispiele

  ```
  fputc(stdout);
  write(1, buf, n);
  ```

– Bei Ausgabefunktionen wird als Dateiargument der Dateizeiger `stderr` bzw. der Dateideskriptor 2 angegeben.

– Es werden Ausgabefunktionen benutzt, die standardmäßig auf `stdout/stderr` schreiben, z.B. `printf()`, `puts()`, `putchar()` bzw. `perror()`.

Wurde mit `PARAMETER-PROMPTING=YES` (in der Compiler-Option `RUNTIME-OPTIONS`) eine Parameterzeile angefordert, kann in dieser Parameterzeile die Standardausgabe (Dateizeiger `stdout` bzw. Dateideskriptor 1) und die Standard-Fehlerausgabe (Dateizeiger `stderr` bzw. Dateideskriptor 2) auf eine katalogisierte Datei umgewiesen werden (siehe auch C- und C++-Benutzerhandbücher).

Diese Umlenkung wirkt sich **nicht** auf Dateien aus, die mit dem Namen "`(SYSOUT)`" bzw. "`(SYSTERM)`" geöffnet wurden.

SYSLST

Ein C-Programm kann `SYSLST` folgendermaßen verwenden:

– Mit einer Öffnungsfunktion (`fopen()`, `freopen()`, `open()`) wird eine Datei mit dem Namen `"(SYSLST)"` zum Schreiben geöffnet. Der von der Öffnungsfunktion gelieferte Dateizeiger dient als Argument einer anschließenden Ausgabefunktion.

  Beispiel

  ```
  FILE *fp;
  fp = fopen("(SYSLST)", "w");
  fprintf(fp, "\t TEXT \n");
  ```

– Wurde mit `PARAMETER-PROMPTING=YES` (in der Compiler-Option `RUNTIME-OPTIONS`) eine Parameterzeile angefordert, kann in dieser Parameterzeile die Standardausgabe bzw. die Standard-Fehlerausgabe auf `SYSLST` umgelenkt werden (siehe auch C- und C++-Benutzerhandbücher).

  Diese Umlenkung wirkt sich **nicht** auf Dateien aus, die mit dem Namen `"(SYSOUT)"` geöffnet wurden.

Standardmäßig werden `SYSLST`-Dateien automatisch bei Taskende (`LOGOFF`) ausgedruckt.

Sollen die Daten nicht automatisch auf den Drucker, sondern in eine katalogisierte Datei ausgegeben werden, muss vor Programmablauf `SYSLST` umgelenkt werden. Dies geschieht mit dem Kommando `ASSIGN-SYSLST` *dateiname*.

Zwischenraumzeichen

Die Steuerzeichen für Zwischenraum sowie das Steuerzeichen '\b' (vgl. Tabelle unten) werden von allen Ausgabefunktionen ausgewertet, die in Textdateien schreiben und als Argument das Steuerzeichen entweder als Zeichenkonstante (beginnend mit \) oder als numerischen EBCDIC-Wert erhalten. Die dezimalen bzw. hexadezimalen Werte der Steuerzeichen finden Sie in den C- und C++-Benutzerhandbüchern (EBCDIC-Tabelle).

In der folgenden Tabelle bedeutet:

X Steuerzeichen wird in die entsprechende Wirkung umgesetzt.

leer Steuerzeichen wird als Textzeichen (EBCDIC-Wert) in die Datei geschrieben.

Ausgabemedium	\ n	\ t	\ f	\ v	\ r	\ b
SAM/ISAM	X	X				
SYSOUT/SYSTERM	X	X	X			
SYSLST	X	X	X	X	X	X

Tabulator (\t)
: Das Tabulatorzeichen wird in die entsprechende Anzahl Leerzeichen umgesetzt. Die Tabulatorpositionen haben einen Acht-Spalten-Abstand (1, 9, 17, ...). Statt des Tabulatorzeichens werden die entsprechenden Leerzeichen eingelesen.

  Bei SAM- und ISAM-Dateien wird das Tabulatorzeichen nur im Übersetzungsmodus KERNIGHAN-RITCHIE standardmäßig in Leerzeichen umgesetzt, im ANSI-Modus dagegen nicht (siehe fopen(), freopen()).

Zeilenvorschub (\n)
: Das Neue-Zeile-Zeichen wird in einen Zeilenwechsel (Satzwechsel) umgesetzt. Anschließende Lesefunktionen liefern dann für einen Satzwechsel ein Neue-Zeile-Zeichen.

Seitenvorschub (\f)
: SYSLST: Es wird ein Seitenvorschub durchgeführt, die folgenden Daten werden auf einer neuen Seite ausgegeben.
  SYSOUT, SYSTERM zum Schreiben: Am Terminal wird die Meldung please acknowledge ausgegeben.

Vertikaler Tabulator (\v)
: Es wird eine entsprechende Anzahl von Leerzeilen ausgegeben, um die nächste Zeilen-Tabulatorposition zu erreichen. Diese Tabulatorpositionen haben einen Acht-Zeilen-Abstand (1, 9, 17, ...).

Wagenrücklauf (`\r`)
: Es wird ohne Zeilenvorschub an den Beginn der aktuellen Zeile positioniert, d.h. die folgenden Daten werden in die gleiche Zeile geschrieben. Damit lässt sich z.B. eine Unterstreichung erzielen.

Zeichen rücksetzen (`\b`)
: Das nachfolgende Zeichen wird auf die Position des vorhergehenden Zeichens geschrieben. Damit kann z.B. ein Buchstabe mit einem Akzent versehen werden. `\b` zählt nicht im engeren Sinne zu den Zwischenraumzeichen (vgl. `isspace()`) sondern zu den Steuerzeichen (vgl. `iscntrl()`).

Die Verwendung von `\r` und `\b` ist nur sinnvoll bei Druckern mit Überdruckfunktion.

Katalogisierte Plattendateien (SAM, ISAM, PAM)

C-Programme verarbeiten katalogisierte Plattendateien mit den Zugriffsmethoden SAM, ISAM und PAM.

Beim Öffnen einer bereits existierenden Datei werden die Zugriffsmethode und andere Dateiattribute dem Katalogeintrag entnommen.

Beim Neuanlegen einer Datei gelten je nach C-Dateiart (Binärdatei, Textdatei, stromorientierte oder satzorientierte Ein-/Ausgabe) Standardwerte der C-Laufzeitbibiliothek. Diese Werte können mit einem `ADD-FILE-LINK`-Kommando vor Aufruf des Programms geändert werden. Dazu muss bei den Öffnungsfunktionen (`open()`, `creat()`, `fopen()`, `freopen()`) ein Linkname angegeben (`"link=`*linkname*`"`) und dieser Linkname im `ADD-FILE-LINK`-Kommando mit dem Namen der katalogisierten Datei verknüpft werden.

Es sind nicht alle möglichen Dateiattribute kombinierbar. Kombinationen, die weder funktionell noch aus Performancegründen nötig sind, werden von den Ein-/Ausgabefunktionen der C-Laufzeitbibiliothek nicht unterstützt.

Im folgenden erhalten Sie Informationen

– zu den Standardwerten sowie den möglichen Modifikationen der Dateiattribute,

– zum K- und NK-Blockformat,

– zur strom- und satzorientierten Verarbeitung von Plattendateien.

Standardwerte und zulässige Modifikationen der Dateiattribute

Mit den Ein-/Ausgabe-Funktionen der C-Laufzeitbibiliothek können Plattendateien mit den in den folgenden Tabellen aufgeführten Dateiattributen verarbeitet werden. Die Standardattribute, die das Laufzeitsystem einsetzt, wenn der Benutzer keine Angaben im ADD-FILE-LINK-Kommando bzw. bei den Öffnungsfunktionen macht, sind jeweils unterstrichen.

Erläuterungen zu den folgenden Tabellen

- Die maximale Anzahl Datenbytes in den Tabellen gibt die Anzahl der Zeichen an, die vom C-Programm in einen Satz bzw. Block abgelegt werden (feste Satzlänge) oder maximal abgelegt werden können (variable Satzlänge).
- Die Größe des logischen Blocks (BLKSIZE) ist abhängig von Art und Format des Datenträgers:
  K- und NK2-Platten: Standardblock (2048 Bytes) oder ein ganzzahliges Vielfaches eines Standardblocks (maximal 16 Standardblöcke).
  NK4-Platten: Mindestens zwei Standardblöcke (4096 Bytes) oder ein ganzzahliges Vielfaches davon (2, 4, 6, 8 Standardblöcke).
- Zum Blockformat (BLKCTRL) und zur maximalen Anzahl Datenbyte beachten Sie auch Abschnitt „K- und NK-Blockformat“ auf Seite 91. Insbesondere finden Sie dort Hinweise, wie bei NK-ISAM-Dateien Überlaufblöcke vermieden werden können, die dann entstehen, wenn beim Schreiben der Sätze die volle Länge einer Übertragungseinheit (RECSIZE=BLKSIZE) ausgenutzt wird.
- Bei Dateien mit variabler Satzlänge (RECFORM=V) zählt in C generell das 4 Byte lange Satzlängenfeld nicht zu den Satzdaten. Die maximale Anzahl Datenbytes reduziert sich deshalb um 4 Bytes.

– Bei Dateien mit RECFORM=U legt RECSIZE (RECORD-SIZE-Parameter im ADD-FILE-LINK-Kommando) das Register fest, in dem die Länge eines Satzes übergeben wird. Dieses Register ist fest vorgegeben (R4) und darf nicht geändert werden.

FCB-TYPE	REC-FORM	BLKCTRL	BLKSIZE (STD,*n*)	RECSIZE (*r* byte)	Max. Anzahl Datenbytes
SAM [1]	V	PAMKEY	1≤ *n* ≤16	4≤ *r* ≤*n**2048-4	RECSIZE - 4
		DATA(2K)	1≤ *n* ≤16	4≤ *r* ≤*n**2048-16	RECSIZE - 4
		DATA(4K)	2≤ *n* ≤16		
	U	PAMKEY	1≤ *n* ≤16		BLKSIZE
		DATA(2K)	1≤ *n* ≤16		BLKSIZE - 16
		DATA(4K)	2≤ *n* ≤16		
ISAM [2]	V	PAMKEY	1≤ *n* ≤16	12≤ *r* ≤*n**2048	RECSIZE - 12
		DATA(2K)	1≤ *n* ≤16	12≤ *r* ≤*n**2048	RECSIZE - 12
		DATA(4K)	2≤ *n* ≤16		

[1] SAM-Dateien werden nur im KR-Modus (vgl. SOURCE-PROPERTIES-Option in den C- und C++-Benutzerhandbüchern) standardmäßig erstellt. Im ANSI-Modus werden standardmäßig ISAM-Dateien erstellt.

[2] Der Standardwert für die Schlüsselposition ist 5, für die Schlüssellänge 8. Diese Werte können nicht modifiziert werden. Auf die Schlüssel kann der Benutzer nicht zugreifen; sie werden von der C-Laufzeitbibiliothek erzeugt und verwaltet: Beim Neuerstellen einer ISAM-Datei erhält der erste Satz den Schlüssel "00010000", bei jedem weiteren Satz wird der Schlüssel um die Schrittweite 100 erhöht.

FCB-TYPE	REC-FORM	BLKCTRL	BLKSIZE (STD,*n*)	RECSIZE (*r* byte)	Max. Anzahl Datenbytes
SAM	F	PAMKEY	1≤ *n* ≤16	1≤ *r* ≤*n**2048	RECSIZE
		DATA(2K)	1≤ *n* ≤16	1≤ *r* ≤*n**2048-16	RECSIZE
		DATA(4K)	2≤ *n* ≤16		
	V	PAMKEY	1≤ *n* ≤16	4≤ *r* ≤*n**2048-4	RECSIZE - 4
		DATA(2K)	1≤ *n* ≤16	4≤ *r* ≤*n**2048-16	RECSIZE - 4
		DATA(4K)	2≤ *n* ≤16		
	U	PAMKEY	1≤ *n* ≤16		BLKSIZE
		DATA(2K)	1≤ *n* ≤16		BLKSIZE - 16
		DATA(4K)	2≤ *n* ≤16		
PAM		PAMKEY	1≤ *n* ≤16		BLKSIZE
		DATA(2K)	1≤ *n* ≤16		BLKSIZE - 12
		DATA(4K)	2≤ *n* ≤16		
		NO(2K)	1≤ *n* ≤16		BLKSIZE
		NO(4K)	2≤ *n* ≤16		
SAM	V	PAMKEY	1≤ *n* ≤16	4≤ *r* ≤*n**2048-4	RECSIZE - 4
		DATA(2K)	1≤ *n* ≤16	4≤ *r* ≤*n**2048-16	RECSIZE - 4
		DATA(4K)	2≤ *n* ≤16		
	F	PAMKEY	1≤ *n* ≤16	1≤ *r* ≤*n**2048	RECSIZE
		DATA(2K)	1≤ *n* ≤16	1≤ *r* ≤*n**2048-16	RECSIZE
		DATA(4K)	2≤ *n* ≤16		
	U	PAMKEY	1≤ *n* ≤16		BLKSIZE
		DATA(2K)	1≤ *n* ≤16		BLKSIZE - 16
		DATA(4K)	2≤ *n* ≤16		
PAM		PAMKEY	1≤ *n* ≤16		BLKSIZE
		DATA(2K)	1≤ *n* ≤16		BLKSIZE - 12
		DATA(4K)	2≤ *n* ≤16		
		NO(2K)	1≤ *n* ≤16		BLKSIZE
		NO(4K)	2≤ *n* ≤16		

FCB-TYPE	REC-FORM	BLKCTRL	BLKSIZE (STD,*n*)	RECSIZE (*r* byte)	Max. Anzahl Datenbytes
ISAM [1]	V	PAMKEY	1≤ *n* ≤16	5≤ *r* ≤*n**2048	RECSIZE - 4
		DATA(2K)	1≤ *n* ≤16	5≤ *r* ≤*n**2048	RECSIZE - 4
		DATA(4K)	2≤ *n* ≤16		
	F	PAMKEY	1≤ *n* ≤16	1≤ *r* ≤*n**2048-4	RECSIZE
		DATA(2K)	1≤ *n* ≤16	1≤ *r* ≤*n**2048-4	RECSIZE
		DATA(4K)	2≤ *n* ≤16		

[1] Die Standardattribute für Schlüsselposition (bei Satzformat V = 5, bei F = 1) und Schlüssellänge (8) können modifiziert werden, und zwar die Schlüsselposition bis auf maximal 32767 und die Schlüssellänge bis auf maximal 255.

Außerdem können Mehrfachschlüssel vereinbart werden (`DUP-KEY=Y`). Standardmäßig gilt `DUP-KEY=N`.

Im Gegensatz zur stromorientierten Ein-/Ausgabe gehören die ISAM-Schlüssel zu den Satzdaten, die vom C-Programm geschrieben bzw. beim Lesen an das C-Programm geliefert werden.

K- und NK-Blockformat

Das BS2000 unterstützt Datenträger, die unterschiedlich formatiert sind:

– **K**ey-Datenträger für das Abspeichern von Dateien, in denen die Blockkontrollinformation in einem separaten Feld ("Pamkey") pro 2 Kbyte-Datenblock steht. Diese Dateien besitzen das Blockformat PAMKEY.

– **N**on-**K**ey-Datenträger für Dateien, in denen keine separaten Pamkey-Felder existieren, sondern die Blockkontrollinformation entweder fehlt (Blockformat NO) oder im jeweiligen Datenblock untergebracht ist (Blockformat DATA).

Ab BS2000/OSD V1.0 werden NK-Datenträger nach der Mindestgröße der Übertragungseinheit (Transfer Unit) unterschieden. NK2-Datenträger haben die bisherige Transfer Unit von 2 Kbyte. NK4-Datenträger haben eine Transfer Unit von 4 Kbyte.

Das Blockformat wird durch den Operanden `BLOCK-CONTROL-INFO` des `ADD-FILE-LINK`-Kommandos gesteuert:

```
BLOCK-CONTROL-INFO = BY-PROGRAM / BY-CATALOG / WITHIN-DATA-BLOCK / NO /
PAMKEY
```

Für NK-ISAM-Dateien gibt es ab BS2000 V11.0 zwei weitere Operandenwerte, nämlich:

```
WITHIN-DATA-2K-BLOCK / WITHIN-DATA-4K-BLOCK
```

Die ausführliche Beschreibung des `BLOCK-CONTROL-INFO`-Operanden der verschiedenen Datei- und Datenträgerstrukturen sowie der Umstellung von K-Dateiformat auf NK-Dateiformat finden sich im Handbuch „Einführung in das DVS“.

Wird beim Neuerstellen einer Datei kein `ADD-FILE-LINK`-Kommando verwendet oder `BLOCK-CONTROL-INFO=BY-PROGRAM` angegeben, gelten Standardwerte der C-Laufzeitbibliothek. Diese Werte hängen ab vom Plattentyp, der vom Systemverwalter angebbaren CLASS2-OPTION und von der Zugriffsmethode:

Dateiorganisation	`CLASS2-OPTION BLKCTRL=NONKEY`			
	nicht angegeben		**angegeben**	
	K-Platte	**NK-Platte**	**K-Platte**	**NK-Platte**
SAM	PAMKEY	DATA	DATA	DATA
ISAM	PAMKEY	DATA	DATA	DATA
PAM	PAMKEY	NO	NO	NO

K- und NK-ISAM-Dateien

ISAM-Dateien im K-Format, die die maximale Satzlänge ausnützen, werden im NK-Format länger als der nutzbare Bereich des Datenblocks. Sie können im NK-Format behandelt werden, da das DVS Verlängerungen von Datenblöcken, sog. Überlaufblöcke, bildet.

Die Bildung von Überlaufblöcken bringt folgende Probleme mit sich:

– die Überlaufblöcke erhöhen den Platzbedarf auf der Platte und damit die Zahl der Ein-/Ausgaben während der Dateibearbeitung,

– der ISAM-Schlüssel darf in keinem Fall in einem Überlaufblock liegen.

Überlaufblöcke können vermieden werden, wenn man dafür sorgt, dass der längste Satz der Datei nicht länger ist, als der bei NK-ISAM-Dateien nutzbare Bereich eines logischen Blockes.

Für Datensätze nutzbarer Bereich bei NK-ISAM-Dateien

Die folgende Tabelle stellt dar, wie man bei ISAM-Dateien errechnen kann, wieviel Platz pro logischem Block für Datensätze zur Verfügung steht.

Dateiformat	RECORD–FORMAT	**maximaler nutzbarer Bereich**
K-ISAM	VARIABLE	BUF–LEN
	FIXED	BUF–LEN – (*s**4) wobei *s* = Anzahl der Sätze pro logischem Block
NK-ISAM	VARIABLE	BUF–LEN – (*n**16) – 12 – (*s**2) (auf nächste durch 4 teilbare Zahl abgerundet) wobei *n* = Blockungsfaktor *s* = Anzahl der Sätze pro logischem Block
	FIXED	BUF–LEN – (*n**16) – 12 – (*s**2) – (*s**4) (auf nächste durch 4 teilbare Zahl abgerundet) wobei *n* = Blockungsfaktor *s* = Anzahl der Sätze pro logischem Block

Zur Erläuterung der Formeln

Bei NK-ISAM-Dateien enthält jede PAM-Seite eines logischen Blocks jeweils 16 Byte Verwaltungsinformation. Der logische Block enthält zusätzlich weitere 12 Byte Verwaltungsinformation und pro Satz einen 2 Byte langen Satzpointer.
Bei RECORD–FORMAT=FIXED ist pro Satz ein 4 Byte langes Satzlängenfeld zwar vorhanden, wird aber nicht zur Satzlänge gerechnet. Deshalb müssen in diesen Fällen pro Satz jeweils 4 Byte abgezogen werden.

Beispiel: maximale Satzlänge einer NK-ISAM-Datei (feste Satzlänge)

Dateivereinbarung:

```
/ADD-FILE-LINK ...,RECORD-FORMAT=FIXED,BUFFER-LENGTH=STD(SIZE=2),
BLOCK-CONTROL-INFO=WITHIN-DATA-BLOCK
```

maximale Satzlänge nach der Formel:

```
4096 - (2*16) - 12 - 1*2 - 1*4 = 4046, abgerundet auf die nächste durch
vier teilbare Zahl: 4044 (byte).
```

Unterstützung der Zugriffsmethode DIV

Ab BS2000/OSD V1.0 bietet das DVS die neue Zugriffsart DIV (DATA IN VIRTUAL) an. Diese Zugriffsart eignet sich insbesondere für die Bearbeitung von unstrukturierten Datenströmen, wie sie in C-Programmen (u.a. aus UNIX portierten) häufig vorkommen.

Mit DIV können NK-PAM-Dateien verarbeitet werden, die keine Datenverwaltungsinformationen enthalten (BLOCK-CONTROL-INFO=NO) und die auf gemeinschaftlicher Platte (Public) liegen.

Wenn wiederholt auf Daten zugegriffen wird, die bereits durch einen vorangegangenen Zugriff in ein „Fenster“ eingelesen wurden, kann sich ein beachtlicher Performance-Gewinn ergeben.

Weitere Hintergrundinformationen zur Zugriffsart DIV finden Sie im Handbuch „DVS Assembler-Schnittstelle“.

In BS2000/OSD-Versionen ab V1.0 führt die C-Laufzeitbibiliothek die stromorientierte Ein-/Ausgabe auf NK-PAM-Dateien ohne Datenverwaltungsinformationen generell mit der Zugriffsart DIV durch. Bei NK-PAM-Dateien, die für satzorientierte Ein-/Ausgabe geöffnet werden, ist die Verwendung von DIV nicht möglich.

Hinweise zur stromorientierten Ein-/Ausgabe

Binärdateien (SAM)

Standardmäßig wird mit fester Satzlänge (F) gearbeitet. Beim Schließen der Datei wird der letzte Satz mit binären Nullen (falls nötig) aufgefüllt. Wird diese Datei neuerlich geöffnet, und es werden Daten an das Ende der Datei geschrieben, wird stets ein neuer Satz begonnen. Das Weiterschreiben erfolgt also hinter den binären Nullen.

Wird mit variabler Satzlänge gearbeitet (V oder U), kann das Weiterschreiben bytebezogen erfolgen. Allerdings sind durch die variablen Satzlängen beim Positionieren (z.B. mit `fseek()`, `ftell()`) Performanceverluste in Kauf zu nehmen.

Binärdateien (PAM)

Um bei PAM-Dateien ein bytebezogenes Fortschreiben (nach einem Schließen und neuerlichem Öffnen) zu ermöglichen, schreibt das C-Laufzeitsystem Verwaltungsdaten an das Ende der Datei. Diese Daten werden zum Zeitpunkt des Öffnens oder Schließens konsistent verwaltet. Aus diesem Grund ist ein simultanes Bearbeiten einer PAM-Datei durch verschiedene Tasks nicht möglich, sofern eine der beteiligten Tasks die Datei verlängert. Außerdem setzt das C-Laufzeitsystem keine Sperren. Werden Daten von mehreren Anwendern verändert, kann dies zu inkonsistenten Zuständen führen.

Textdateien (SAM, ISAM)

Werden SAM- oder ISAM-Dateien im Update-Modus verarbeitet, darf beim Ändern bereits existierender Sätze die ursprüngliche Satzlänge nicht geändert werden. Das heißt, ein Neue-Zeile-Zeichen (`\n`) darf nicht in ein anderes Zeichen geändert werden oder umgekehrt.

Hinweise zur satzorientierten Ein-/Ausgabe

Satzorientierte Ein-/Ausgabe ist für SAM-, ISAM- und PAM-Dateien möglich.

Mit den Funktionen `fopen()` bzw. `freopen()` muss die Datei stets im Binärmodus und mit dem Zusatz `type=record` geöffnet werden.

Ein-/Ausgabefunktionen, die Zeichen oder Zeichenketten (bis `\n`) einlesen oder ausgeben, sind bei satzorientierter Ein-/Ausgabe nicht anwendbar.

Verfügbare Ein-/Ausgabefunktionen

Folgende Funktionen stehen zur Verarbeitung von Dateien mit STREAM-Ein-/Ausgabe zur Verfügung:

`fopen()`, `freopen()`	Öffnen
`fclose()`	Schließen
`fread()`	Lesen
`fwrite()`	Schreiben
`fsetpos()`, `fgetpos()`	Positionieren auf ermittelte Position
`fseek()`	Positionieren auf Dateianfang/Dateiende
`rewind()`	Positionieren auf Dateianfang
`flocate()`	explizit Positionieren in einer ISAM-Datei
`fdelrec()`	Löschen eines Satzes in einer ISAM-Datei

Außerdem sind folgende Funktionen zur Dateiverwaltung bzw. Fehlerbehandlung unverändert anwendbar:

`feof()`, `ferror()`, `clearerr()`, `unlink()`, `remove()`, `rename()`

Alle hier nicht aufgeführten Ein-/Ausgabefunktionen stehen für die satzorientierte Ein-/Ausgabe nicht zur Verfügung und werden mit einem Fehler-Returnwert abgewiesen.
Die beiden Makros `getc()` und `putc()` haben jedoch aus Performancegründen keine Prüfung. Das Verhalten ist undefiniert, wenn diese Makros auf Dateien mit satzorientierter Ein-/Ausgabe angewendet werden.

Verarbeitung einer Datei in satz- und stromorientierter Ein-/Ausgabe

Es ist möglich, eine Datei, die mit satzorientierter Ein-/Ausgabe erstellt wurde, für stromorientierte Ein-/Ausgabe zu öffnen und umgekehrt. Es ist jedoch zu beachten, dass bei stromorientierter Ein-/Ausgabe nicht alle Dateiattribute unterstützt werden, die bei satzorientierter Ein-/Ausgabe möglich sind.

FCBTYPE einer neu zu erstellenden Datei

Der `FCBTYPE` einer neu zu erstellenden Datei kann folgendermaßen festgelegt werden:

– Angabe in einem `ADD-FILE-LINK`-Kommando und Verwendung des LINK-Namens bei den Funktionen `fopen()` bzw. `freopen()`.

– Angabe des `forg`-Parameters bei den Funktionen `fopen()` bzw. `freopen()`, und zwar:
`forg=seq`: Es wird eine SAM-Datei erstellt
`forg=key`: Es wird eine ISAM-Datei erstellt.

Es gibt folgende Einschränkungen für den `FCBTYPE` einer Datei und die Angaben bei `fopen()` bzw. `freopen()`:

– Bei Angabe von `type=record` muss die Datei den `FCBTYPE` SAM, PAM oder ISAM haben.

– Bei Angabe von `forg=seq` muss die Datei den `FCBTYPE` SAM oder PAM haben.

– Bei Angabe von `forg=key` muss die Datei den `FCBTYPE` ISAM haben.

– Die Angabe des Anfügemodus "a" ist für ISAM-Dateien unzulässig. Die Position bestimmt sich aus dem Schlüssel im Satz.

Mehrfachschlüssel bei ISAM-Dateien

Standardmäßig sind für ISAM-Dateien keine Mehrfachschlüssel zugelassen. Durch die Angabe von DUP-KEY=Y in einem ADD-FILE-LINK-Kommando können jedoch Mehrfachschlüssel verwendet werden.

Temporäre PAM-Dateien im virtuellen Speicher (INCORE-Dateien)

Wird mit den Funktionen fopen(), freopen() oder open() der Dateiname "(INCORE)" angegeben, wird eine temporäre PAM-Datei im virtuellen Speicher angelegt. Diese Datei „lebt“ nur für die Dauer eines Programmablaufs.

INCORE-Dateien müssen zuerst zum Schreiben geöffnet werden, bevor auf sie lesend zugegriffen werden kann (vgl. fopen(), freopen(), open()).

INCORE-Dateien werden als Binärdateien verarbeitet.

Allgemeine Terminalschnittstelle

Dieser Abschnitt beschreibt eine allgemeine Terminalschnittstelle, die zur Steuerung der seriellen Kommunikationsschnittstellen angeboten wird. Dies sind lokal angeschlossene, asynchrone Leitungen.

Diese Schnittstelle wird auf BS2000-Blockterminals nur eingeschränkt unterstützt.

Terminal-Gerätedatei öffnen

Wenn eine Gerätedatei für ein Terminal geöffnet wird, dann wartet der Prozess normalerweise solange, bis eine Verbindung hergestellt wurde. In der Praxis öffnen Anwendungen solche Dateien nur sehr selten; diese Dateien werden von speziellen Programmen geöffnet und werden dann zur Standardeingabe, Standardausgabe und Standardfehlerausgabe von Anwendungen.

Wie unter open() beschrieben, bewirkt das Öffnen einer Gerätedatei für ein Terminal ohne gesetztes O_NONBLOCK-Bit, dass der Prozess blockiert, bis das Terminal bereit ist. Wenn der CLOCAL-Modus nicht eingeschaltet ist, dann bedeutet dies, dass gewartet wird, bis eine Verbindung aufgebaut ist. Wenn der CLOCAL-Modus für das Terminal eingeschaltet oder das Bit O_NONBLOCK beim Aufruf von open() angegeben ist, dann liefert open() einen Dateideskriptor, ohne auf den Aufbau einer Verbindung zu warten.

Prozessgruppen

Ein Terminal kann einer Vordergrund-Prozessgruppe zugeordnet sein. Diese Vordergrund-Prozessgruppe spielt eine besondere Rolle bei der Behandlung von Eingabezeichen, die Signale erzeugen, wie dies im Abschnitt „Sonderzeichen“ auf Seite 102 behandelt wird.

Die Vordergrund-Prozessgruppe eines Terminals kann von einem Prozess gesetzt oder abgefragt werden, wenn die in diesem Abschnitt angegebenen Anforderungen hinsichtlich der Zugriffsrechte erfüllt sind; siehe auch `tcgetpgrp()` und `tcsetpgrp()`. Die Terminalschnittstelle hilft bei dieser Zuteilung, indem sie den Zugriff auf das Terminal für solche Prozesse einschränkt, die nicht in der aktuellen Prozessgruppe sind (siehe auch Abschnitt „Zugriffssteuerung für Terminals“ auf Seite 97).

Das steuernde Terminal

Ein Terminal kann zu einem Prozess als sein steuerndes Terminal gehören. Jeder Prozess einer Sitzung, der ein steuerndes Terminal besitzt, besitzt dasselbe steuernde Terminal. Ein Terminal kann für höchstens eine Sitzung das steuernde Terminal sein. Als steuerndes Terminal reserviert der Sitzungsleiter die erste offene Terminal-Gerätedatei. Wenn ein Sitzungsleiter, der kein steuerndes Terminal besitzt, die Terminal-Gerätedatei ohne gesetztes `O_NOCTTY`-Bit öffnet, die noch nicht einer Sitzung zugeordnet ist (siehe auch `open()`), kann dieses Terminal das steuernde Terminal des Sitzungsleiters werden. Wenn ein Prozess, der kein Sitzungsleiter ist, die Terminal-Gerätedatei öffnet, oder wenn die Option `O_NOCTTY` beim Aufruf von `open()` verwendet wird, wird das Terminal nicht zum steuernden Terminal für den Prozess. Wenn ein steuerndes Terminal einer Sitzung zugeordnet wird, dann wird dessen Vordergrund-Prozessgruppe gleich der Prozessgruppe des Sitzungsleiters gesetzt.

Das steuernde Terminal wird von einem Sohnprozess durch einen `fork`-Aufruf geerbt. Ein Prozess gibt sein steuerndes Terminal auf, wenn er eine neue Sitzung durch die Funktion `setsid()` erzeugt oder wenn alle Dateideskriptoren, die dem steuernden Terminal zugeordnet waren, geschlossen wurden.

Wenn ein steuernder Prozess beendet wird, dann wird das steuernde Terminal von der aktuellen Sitzung gelöst, was einem neuen Sitzungsleiter erlaubt, dieses für sich zu reservieren. Nachfolgende Zugriffe auf dieses Terminal durch andere Prozesse aus der früheren Sitzung können verweigert werden, wobei Versuche, auf das Terminal zuzugreifen, behandelt werden, als sei ein Verbindungsabbruch bei einem Modem festgestellt worden.

Zugriffssteuerung für Terminals

Wenn ein Prozess in der Vordergrund-Prozessgruppe seines steuernden Terminals ist, dann ist ihm das Lesen von diesem Terminal erlaubt, so wie dies im Abschnitt „Eingaben verarbeiten und Daten lesen“ auf Seite 98 beschrieben ist. Für die Implementierungen, die Auftragssteuerung (job control) unterstützen, verursacht jeder Versuch eines Prozesses

aus einer Hintergrund-Prozessgruppe, von seinem steuernden Terminal zu lesen, dass seiner Prozessgruppe das Signal SIGTTIN gesendet wird, sofern nicht einer der folgenden Fälle zutrifft:

– Der lesende Prozess ignoriert oder blockiert das Signal SIGTTIN.

– Die Prozessgruppe des lesenden Prozesses ist verwaist.

In diesen Fällen liefert die Funktion read() das Ergebnis –1, wobei errno gleich EIO gesetzt ist und kein Signal gesendet wird. Die voreingestellte Signalaktion für SIGTTIN ist, den Prozess anzuhalten, dem dieses Signal gesendet wird (siehe auch signal.h).

Wenn ein Prozess in der Vordergrund-Prozessgruppe seines steuernden Terminals ist, dann sind Schreiboperationen erlaubt, wie dies im Abschnitt „Daten schreiben und Ausgaben verarbeiten“ auf Seite 102 beschrieben ist. Versuche eines Prozesses aus einer Hintergrund-Prozessgruppe, auf sein steuerndes Terminal zu schreiben, verursachen, dass der Prozessgruppe das Signal SIGTTOU gesendet wird, sofern nicht einer der folgenden Spezialfälle gegeben ist:

– Wenn TOSTOP nicht gesetzt oder TOSTOP gesetzt ist und der Prozess das Signal SIGTTOU ignoriert oder blockiert, darf der Prozess auf das Terminal schreiben und das Signal SIGTTOU wird nicht gesendet.

– Wenn TOSTOP gesetzt ist, die Prozessgruppe des schreibenden Prozesses verwaist ist und der schreibende Prozess das Signal SIGTTOU nicht blockiert, dann liefert die Funktion write() das Ergebnis –1, wobei errno gleich EIO gesetzt ist und kein Signal gesendet wird.

Bestimmte Aufrufe von Funktionen, die Parameter des Terminals setzen, werden auf dieselbe Art behandelt wie Aufrufe von write(), ausser dass TOSTOP ignoriert wird; d.h. deren Wirkung ist dieselbe wie die eines Schreibversuchs auf das Terminal, wenn TOSTOP gesetzt ist (siehe auch Abschnitt „Lokalmodi“ auf Seite 111, tcdrain(), tcflow(), tcflush(), tcsendbreak() und tcsetattr()).

Eingaben verarbeiten und Daten lesen

Ein Terminal, das einer Gerätedatei zugeordnet ist, kann im Vollduplexbetrieb arbeiten, so dass es jederzeit möglich ist, Zeichen einzugeben, auch bei laufender Ausgabe. Im POSIX-Subsystem wird der Vollduplexbetrieb für Terminals von TIAM simuliert.

Jeder Gerätedatei eines Terminals ist ein **Eingabepuffer** zugeordnet, in den die eingehenden Daten durch das System gespeichert werden, bevor sie vom Prozess gelesen werden können. Die Eingabe geht verloren, wenn die Eingabepuffer des Systems voll sind oder wenn eine Eingabezeile die zulässige Höchstzahl {MAX_INPUT} für die Eingabe von Zeichen überschreitet (siehe limits.h). {MAX_INPUT} muss größer oder gleich {_POSIX_MAX_CANON} sein. Dieser Wert ist mit pathconf() abfragbar.

Es sind zwei generelle Arten von Eingabeverarbeitung verfügbar, je nachdem, ob die Gerätedatei für das Terminal im **Standard-Eingabemodus** oder im **besonderen Eingabemodus** arbeitet. Diese Modi sind in den nächsten beiden Abschnitten „Standard-Eingabeverarbeitung“ und „Besondere Eingabeverarbeitung“ beschrieben. Zusätzlich werden Eingabezeichen entsprechend der Einstellung der Komponenten `c_iflag` (siehe auch Abschnitt „Eingabemodi“ auf Seite 105) und `c_lflag` (siehe auch Abschnitt „Lokalmodi“ auf Seite 111) verarbeitet. Diese Verarbeitung kann das lokale Echo einschließen. Dies bedeutet, dass Eingabezeichen sofort nach ihrem Empfang an das entsprechende Terminal zurückgesendet werden. Dies ist besonders nützlich für Terminals, die im Vollduplexbetrieb arbeiten.

Wenn das Bit `O_NONBLOCK` nicht gesetzt ist, werden Leseanforderungen solange blockiert, bis Daten verfügbar sind oder ein Signal eintrifft. Wenn das Bit `O_NONBLOCK` gesetzt ist, dann wird die Leseanforderung auf eine der folgenden Arten ohne Warten beendet:

– Sind genügend Daten verfügbar, um die konkrete Anforderung zu erfüllen, dann kehrt die Funktion `read()` erfolgreich zurück und liefert als Ergebnis die Anzahl der gelesenen Bytes.

– Wenn nicht genügend Daten verfügbar sind, um die konkrete Anforderung zu erfüllen, dann kehrt die Funktion `read()` erfolgreich zurück. Dabei hat sie so viele Daten wie möglich gelesen. Sie liefert als Ergebnis die Anzahl der tatsächlich gelesenen Bytes zurück.

– Sind keine Daten verfügbar, dann liefert die Funktion `read()` den Wert -1, wobei `errno` gleich `EAGAIN` gesetzt ist.

Wann Daten verfügbar sind, hängt davon ab, ob die Standard- oder die besondere Eingabeverarbeitung aktiv ist. Die folgenden Abschnitte „Standard-Eingabeverarbeitung“ und „Besondere Eingabeverarbeitung“ beschreiben jeden dieser Eingabeverarbeitungs-Modi.

Standard-Eingabeverarbeitung

Bei der Standard-Eingabeverarbeitung werden Eingaben von einem Terminal zeilenweise bearbeitet. Eine Zeile wird begrenzt durch ein Neue-Zeile-Zeichen (`LF`), ein Dateiende- oder Zeilenende-Zeichen. Für mehr Informationen zu `EOF` und `EOL` siehe auch Abschnitt „Sonderzeichen“ auf Seite 102. Dies bedeutet, dass ein lesendes Programm so lange angehalten wird, bis eine vollständige Zeile eingegeben wurde. Ebenso besteht die Eingabe aus maximal einer Zeile, gleichgültig, wie viele Zeichen in dem Leseaufruf angefordert wurden. Es muss jedoch nicht eine ganze Zeile auf einmal gelesen werden; es kann eine beliebige Anzahl Zeichen in einem Leseaufruf angefordert werden (auch nur 1 Zeichen), ohne dass Daten verloren gehen.

{MAX_CANON}, die maximale Anzahl von Bytes in einer Zeile (siehe limits.h), muss größer oder gleich {_POSIX_MAX_CANON} sein. Wenn diese Grenze überschritten wird, dann ist das Verhalten des Systems undefiniert. Wenn {MAX_CANON} nicht definiert ist, dann gibt es keine solche Grenze (siehe auch pathconf()). Beide Konstanten haben für BS2000-Blockterminals keine Wirkung, weil die Ein-/Ausgabe dort von TIAM gesteuert wird.

Die Verarbeitung von ERASE- und KILL-Zeichen geschieht dann, wenn eines der Sonderzeichen ERASE und KILL gelesen wird (siehe Abschnitt „Sonderzeichen“ auf Seite 102). Die Verarbeitung dieser Zeichen betrifft den Eingabepuffer, der noch nicht durch ein Neue-Zeile-Zeichen (LF), ein Dateiende- oder ein Zeilenende-Zeichen begrenzt wurde. Diese noch nicht begrenzten Daten bilden die aktuelle Zeile. Das Löschzeichen ERASE löscht das zuletzt eingegebene Zeichen der aktuellen Zeile, sofern ein solches nach dem Zeilenanfang vorhanden ist. Das Löschzeichen KILL entfernt die gesamte aktuelle Eingabezeile, sofern eine solche vorhanden ist. Dabei kann wahlweise die Ausgabe eines neuen Neue-Zeile-Zeichens erfolgen. Die Zeichen ERASE und KILL haben keine Wirkung, wenn sich keine Daten in der aktuellen Zeile befinden. Die Löschzeichen selbst werden nicht im Eingabepuffer abgelegt. Beide Zeichen wirken unmittelbar bei Betätigen der entsprechenden Taste, unabhängig von eventuell eingegebenen Backspace- oder Tabulatorzeichen. Sie können auch direkt als Konstante eingegeben werden, indem man ihnen das Escape-Zeichen \ voranstellt. Das Escape-Zeichen selbst wird nicht gelesen. Die Löschzeichen können geändert werden.

Besondere Eingabeverarbeitung

Diese Art der Eingabeverarbeitung wird nur von zeichenorientierten Terminals, nicht aber von Blockterminals unterstützt.

Bei der besonderen Eingabeverarbeitung werden die Eingabezeichen nicht zu Zeilen zusammengefasst und eine Verarbeitung von ERASE- und KILL-Zeichen findet nicht statt. Die Werte der Elemente MIN und TIME des Vektors c_cc werden verwendet, um zu entscheiden, wie der Prozess die Zeichen erhalten soll. Das O_NONBLOCK-Bit (siehe auch open() oder fcntl()) hat Vorrang vor den Festlegungen im Vektor c_cc . Wenn daher O_NONBLOCK gesetzt ist, kehrt read() sofort zurück, unabhängig von den MIN- und TIME-Werten. Außerdem kann read(), wenn keine Daten vorhanden sind, entweder 0 oder -1 zurückgeben und in letzterem Fall errno gleich EAGAIN setzen.

MIN gibt die Mindestanzahl an Zeichen (maximal 255) an, die bei einer erfolgreich ausgeführten Funktion read() empfangen werden sollten (d.h. die dann dem Benutzer zurückgeliefert werden). TIME ist ein Timer (eine Zeitüberwachung) auf Zehntelsekunden-Basis für schubweise und geringe Datenübertragungen. Wenn MIN größer als {MAX_INPUT} ist, dann ist nicht festgelegt, wie die Anforderung behandelt wird. Die folgenden Absätze beschreiben die vier möglichen Kombinationen von MIN und TIME sowie ihre Wechselwirkung:

1. Fall: MIN > 0, TIME > 0

In diesem Fall dient TIME als zeichenorientierter Timer und wird nach dem ersten empfangenen Zeichen aktiviert. Bei jedem neuen Zeichen wird TIME zurückgesetzt; sobald ein Zeichen empfangen wird, wird TIME gestartet. Werden MIN Zeichen empfangen, bevor der Timer TIME abgelaufen ist, so wird der Leseauftrag erfüllt. Läuft der Timer TIME ab, bevor MIN Zeichen empfangen wurden, so werden die bis zu diesem Zeitpunkt empfangenen Zeichen an den Benutzer übergeben. Es wird immer mindestens ein Zeichen zurückgeliefert, wenn TIME abläuft, da der Timer erst nach dem Empfang des ersten Zeichens aktiviert wird. In diesem Fall blockiert die Leseoperation solange, bis entweder der MIN- und TIME-Mechanismus durch den Empfang des ersten Bytes aktiviert wird oder ein Signal eintrifft.

2. Fall: MIN > 0, TIME = 0

Da TIME den Wert 0 hat, ist die Zeitüberwachung wirkungslos und nur MIN ist signifikant. In diesem Fall blockiert die Leseoperation solange, bis MIN Zeichen empfangen wurden oder bis ein Signal eintrifft. Ein Programm, das diesen Fall nutzt, um Datensätze von einem Terminal zu lesen, kann bei einer Leseoperation beliebig lange blockieren (d.h. auch unendlich lange).

3. Fall: MIN = 0, TIME > 0

Da MIN gleich 0 ist, dient TIME nicht mehr als zeichenorientierter Timer, sondern als Zeitüberwachung für die gesamte Leseoperation, die bei der Bearbeitung des read()-Aufrufs aktiviert wird (Standardbehandlung). In diesem Fall wird eine Leseoperation ausgeführt, sobald entweder ein Zeichen empfangen wird oder der Timer TIME abläuft. Wenn innerhalb des Zeitraums von TIME * 0,1 Sekunden nach dem Aufruf von read() kein Byte empfangen wird, dann liefert die Funktion read() das Ergebnis 0 und hat keine Daten gelesen.

4. Fall: MIN = 0, TIME = 0

In diesem Fall wird sofort die geforderte Anzahl von zu lesenden Zeichen zurückgeliefert oder, wenn nicht so viele verfügbar sind, die Anzahl der aktuell verfügbaren Zeichen. Es wird nicht auf eine weitere Eingabe gewartet. Sind keine Eingabezeichen verfügbar, dann liefert die Funktion read() den Wert 0 als Ergebnis und hat keine Daten gelesen.

Daten schreiben und Ausgaben verarbeiten

Wenn ein Prozess Bytes in eine Gerätedatei für ein Terminal schreibt, dann werden diese Bytes gemäß den Einstellungen in `c_oflag` verarbeitet (siehe Abschnitt „Ausgabemodi“ auf Seite 107). Das System kann einen Puffer-Mechanismus bieten, der so arbeitet, dass alle Bytes, die ein Aufruf von `write()` geschrieben hat, nach dessen Beendigung zur Übertragung zum jeweiligen Gerät anstehen, aber noch nicht notwendigerweise auch schon vollständig übertragen wurden (siehe dazu `write()`, Auswirkungen von `write()` mit gesetztem `O_NONBLOCK`).

Sonderzeichen

Die unten beschriebenen Sonderzeichen werden bei der Initialisierung einer Task durch eine Vortask den Programmtasten zugeordnet. Ihnen sind bei der Ein-/Ausgabe bestimmte Sonderfunktionen zugeordnet. In den Fällen, in denen die Zuordnung von Zeichen und Funktion nicht verändert werden darf, ist das entsprechende Zeichen in Klammern angegeben:

`INTR` Sonderzeichen für die Eingabe, das erkannt wird, wenn das Bit `ISIG` gesetzt ist. Es erzeugt ein Unterbrechungssignal (interrupt) `SIGINT`, das an alle Prozesse in der Vordergrund-Prozessgruppe des Terminals abgesetzt wird. Wenn das Bit `ISIG` gesetzt ist, dann wird das Zeichen nach der Verarbeitung verworfen. Damit werden im Normalfall alle diese Prozesse abgebrochen. Man kann jedoch Vorkehrungen treffen, dass das Signal ignoriert wird oder ein Sprung an eine vorher vereinbarte Adresse erfolgt (siehe `sigaction()` bzw. `signal()`).

`QUIT` Sonderzeichen für die Eingabe, das erkannt wird, wenn das Bit `ISIG` gesetzt ist. Es erzeugt das Signal `SIGQUIT` für alle Prozesse in der Vordergrund-Prozessgruppe, die dem Terminal zugeordnet ist. Wenn `ISIG` gesetzt ist, dann wird das Zeichen `QUIT` nach der Verarbeitung verworfen. Es wird fast genauso behandelt wie das Unterbrechungssignal `SIGINT`, mit einer Ausnahme: Hat der empfangende Prozess keine anderen Vorkehrungen getroffen, so wird er nicht nur abgebrochen, sondern es wird auch ein Speicherabzug (core) erzeugt (siehe auch `sigaction()`).

`ERASE` Sonderzeichen für die Eingabe, das erkannt wird, wenn das Bit `ICANON` gesetzt ist. Es löscht das vorhergehende Zeichen, allerdings nicht über den Zeilenanfang – d.h. ein `NL`–, `EOF`- oder `EOL`-Zeichen – hinaus (vgl. Abschnitt „Standard-Eingabeverarbeitung“). Wenn `ICANON` gesetzt ist, dann wird das Zeichen `ERASE` nach der Verarbeitung verworfen.

`KILL` Sonderzeichen für die Eingabe, das erkannt wird, wenn das Bit `ICANON` gesetzt ist. Es löscht die gesamte Zeile vom letzten `NL`–, `EOF`- oder `EOL`-Zeichen ab. Wenn `ICANON` gesetzt ist, dann wird das Zeichen `KILL` nach der Verarbeitung verworfen.
Dieses Zeichen wird auf den BS2000-Blockterminals nicht unterstützt.

`EOF` Sonderzeichen für die Eingabe, das erkannt wird, wenn das Bit `ICANON` gesetzt ist. Beim Empfang von `EOF` werden alle noch nicht gelesenen Zeichen sofort an das Programm übergeben, ohne auf ein `NL`-Zeichen zu warten; das `EOF`-Zeichen wird verworfen. Sind keine Zeichen vorhanden, d.h. das `EOF`-Zeichen steht am Zeilenanfang, so liefert `read()` den Wert 0 zurück. Das Ergebnis 0 bei einer Leseoperation ist die Standardanzeige für das Dateiende. Wenn `ICANON` gesetzt ist, dann wird das Zeichen `EOF` nach der Verarbeitung verworfen.

`NL` Sonderzeichen für die Eingabe, das erkannt wird, wenn das Bit `ICANON` gesetzt ist. `NL` ist das normale Zeilen-Begrenzungszeichen `\n`. Es kann nicht geändert werden.

`EOL` Sonderzeichen für die Eingabe, das erkannt wird, wenn das Bit `ICANON` gesetzt ist. `EOL` ist ein zusätzliches Zeilen-Begrenzungszeichen und hat dieselbe Funktion wie `NL`. Es wird normalerweise nicht verwendet.

`SUSP` Wenn ein X/Open-kompatibles System Auftragssteuerung unterstützt (siehe auch Abschnitt „Steuerzeichen“ auf Seite 113), dann wird das Sonderzeichen `SUSP` bei der Eingabe erkannt. Wenn das Bit `ISIG` gesetzt ist, dann verursacht der Empfang des Zeichens `SUSP`, dass das Signal `SIGTSTP` an alle Prozesse in der Vordergrund-Prozessgruppe gesendet wird, die dem Terminal zugeordnet ist. Dann wird das Zeichen ebenfalls nach der Verarbeitung verworfen. Dieses Zeichen hat im POSIX-Subsystem keine Wirkung, da hier die Auftragssteuerung nicht unterstützt wird.

`STOP` Sonderzeichen für die Eingabe und für die Ausgabe, das erkannt wird, wenn eines der Bits `IXON` (für die Ausgabe) oder `IXOFF` (für die Eingabe) gesetzt ist. `STOP` kann dazu verwendet werden, eine Ausgabe vorübergehend anzuhalten. Damit kann man an Terminals verhindern, dass die Ausgabe vom Bildschirm verschwindet, bevor man sie lesen konnte. Wenn `IXON` gesetzt ist, dann wird das Zeichen `STOP` nach der Verarbeitung verworfen. Solange die Ausgabe angehalten wird, werden weitere `STOP`-Zeichen ignoriert und nicht gelesen. Das Zeichen `STOP` kann nicht geändert und nicht entwertet werden.
Dieses Zeichen wird auf den BS2000-Blockterminals nicht unterstützt.

`START` Sonderzeichen für die Eingabe und für die Ausgabe, das erkannt wird, wenn eines der Bits `IXON` (für die Eingabe) oder `IXOFF` (für die Ausgabe) gesetzt ist. Das Zeichen `START` dient dazu, eine mit dem `STOP`-Zeichen angehaltene Ausgabe fortzusetzen. Solange die Ausgabe läuft, werden nach-

folgende START-Zeichen ignoriert und nicht gelesen. Wenn IXON gesetzt ist, dann wird das Zeichen START nach der Verarbeitung verworfen. Das Zeichen START kann nicht geändert und nicht entwertet werden.
Dieses Zeichen wird auf den BS2000-Blockterminals nicht unterstützt.

CR Sonderzeichen für die Eingabe, das erkannt wird, wenn das Bit ICANON gesetzt ist; es entspricht dem Zeichen \r. Wenn ICANON und ICRNL gesetzt sind und IGNCR nicht, dann wird dieses Zeichen in das Zeichen NL umgesetzt und hat dieselbe Wirkung wie das Zeichen NL. Das Zeichen CR kann nicht geändert werden.

Die Werte für INTR, QUIT, ERASE, KILL, EOF, EOL und SUSP (nur für Auftragssteuerung) können vom Benutzer geändert werden.

Wenn zwei oder mehr Sonderzeichen denselben Wert haben, dann ist das Verhalten der Funktion undefiniert, die bei Empfang dieses Zeichens ausgeführt wird.

Die ERASE–, KILL- und EOF-Zeichen können durch ein vorangestelltes \ (Escape-Zeichen) entwertet werden; in diesem Fall wird die ihnen zugeordnete Funktion nicht ausgeführt.

Da der Anwender die Tastaturbelegung jederzeit überschreiben kann, kann die voreingestellte, XPG4 Version 2-konforme Tastenbelegung auf BS2000-Kommandoebene mit /RESTORE–CONTROL–KEYS wiederhergestellt werden.

Verbindung abbrechen

Beim Verschwinden des Carrier-Signals (modem disconnect) an der Schnittstelle für ein steuerndes Terminal, wird, wenn in c_cflag CLOCAL nicht gesetzt ist (siehe Abschnitt „Steuermodi“ auf Seite 109), das Signal für den Verbindungsabbruch SIGHUP an den steuernden Prozess gesendet, der diesem Terminal zugeordnet ist. Dadurch wird der steuernde Prozess abgebrochen, sofern keine anderen Vorkehrungen getroffen wurden (siehe exit()). Alle nachfolgenden Leseoperationen von diesem Terminal liefern dann die Anzeige für Dateiende. Damit können Prozesse, die Eingaben von einem Terminal lesen und auf Dateiende prüfen, nach einem Verbindungsabbruch entsprechend beendet werden. Jede nachfolgende Schreiboperation mit write() auf dieses Terminal liefert das Ergebnis -1, und errno ist dann gleich EIO gesetzt, bis die Datei geschlossen wird.

Terminal-Gerätedatei schließen

Wenn der letzte Prozess eine Gerätedatei für ein Terminal schließt, dann werden alle noch anstehenden Ausgaben an dieses Gerät gesendet und alle noch nicht gelesenen Eingaben verworfen. Wenn HUPCL in den Steuermodi gesetzt ist und die Kommunikations-Schnittstelle eine Verbindungsabbruch-Funktion unterstützt, dann führt die Terminalschnittstelle einen Verbindungsabbruch aus.

Einstellbare Parameter

Die Struktur termios

Programme, die Ein- und Ausgabe-Kennzeichen für Terminals steuern müssen, können dies über die Struktur `termios`, die in der Include-Datei `termios.h` definiert ist. Zu den Komponenten dieser Struktur gehören:

Komponententyp	Vektorgröße	Komponentenname	Beschreibung
`tcflag_t`		`c_iflag`	Eingabemodi
`tcflag_t`		`c_oflag`	Ausgabemodi
`tcflag_t`		`c_cflag`	Steuermodi
`tcflag_t`		`c_lflag`	Lokalmodi
`cc_t`	`NCCS`	`c_cc[]`	Sonderzeichen

Die Datentypen `tcflag_t` und `cc_t` sind in der Include-Datei `termios.h` definiert. Sie sind dort als `unsigned` definiert.

Eingabemodi

Die Komponente `c_iflag` beschreibt die grundlegende Eingabesteuerung des Terminals:

Maskenname	Beschreibung
`BRKINT`	Signal `SIGINT` bei `break` senden
`ICRNL`	`CR` bei Eingabe in `NL` umwandeln
`IGNBRK`	`break` ignorieren
`IGNCR`	`CR` ignorieren
`IGNPAR`	Zeichen mit Paritätsfehler ignorieren
`INLCR`	`NL` bei Eingabe in `CR` umwandeln
`INPCK`	Paritätsprüfung für Eingabe aktivieren
`ISTRIP`	8. Bit des Eingabezeichens löschen
`IXOFF`	`START/STOP`-Eingabesteuerung aktivieren
`IXON`	`START/STOP`-Ausgabesteuerung aktivieren
`PARMRK`	Paritätsfehler markieren
`IUCLC` Wird zukünftig vom X/Open-Standard nicht mehr unterstützt.	Bei Eingabe Groß- in Kleinbuchstaben umwandeln.
`IXANY`	Fortsetzung der Ausgabe durch beliebiges Eingabezeichen

Im Zusammenhang mit der asynchronen Datenübertragung über eine serielle Schnittstelle ist ein `break` als eine Folge von 0-Bits definiert, deren Übertragung länger dauert, als für die Übertragung eines Bytes notwendig ist. Die gesamte Folge von 0-Bits wird als ein einziges `break` interpretiert, auch wenn es sich dabei um eine Folge handelt, die mehrere Bytes lang ist. In anderen Zusammenhängen als der asynchronen seriellen Datenübertragung ist die Bedeutung eines `break` nicht festgelegt.

Bei gesetztem `IGNBRK` wird ein in der Eingabe auftretendes `break` ignoriert, d.h. nicht in den Eingabepuffer eingetragen und deshalb von keinem Prozess gelesen. Bei gesetztem `BRKINT` dagegen erzeugt ein `break` ein einzelnes Unterbrechungssignal `SIGINT` und sowohl die Ein- als auch die Ausgabepuffer werden gelöscht. Wenn weder `IGNBRK` noch `BRKINT` gesetzt ist, dann wird ein `break` als einzelnes Zeichen `\0` gelesen, wenn `PARMRK` gesetzt ist, dann als `\377`, `\0`, `\0`.

Ist `IGNPAR` gesetzt, dann wird jedes Byte mit einem Zeichen- oder Paritätsfehler ungleich einem `break` ignoriert.

Wenn `PARMRK` gesetzt und `IGNPAR` nicht gesetzt ist, dann wird jedes Byte mit einem Rahmen- oder Paritätsfehler, welches ungleich einem `break` ist, als eine Folge von drei Zeichen weitergegeben: `\377`, `\0` und `X`, wobei `\377` und `\0` ein 2 Byte langes Kennzeichen für jede dieser Sequenzen ist und `X` dem fehlerhaften Zeichen entspricht. Um Zweifelsfälle auszuschließen wird, wenn `ISTRIP` nicht gesetzt ist, ein gültiges Zeichen `\377` als `\377`, `\377` an die Anwendung ausgeliefert. Wenn weder `PARMRK` noch `IGNPAR` gesetzt ist, dann wird ein Rahmen- oder Paritätsfehler, der ungleich einem `break` ist, als ein einzelnes Zeichen `\0` an die Anwendung weitergegeben.

Bei gesetztem `INPCK` wird die Paritätsprüfung bei der Eingabe aktiviert. Bei nicht gesetztem `INPCK` wird die eingabeseitige Paritätsprüfung deaktiviert. Damit kann das Paritätsbit bei der Ausgabe ohne Berücksichtigung von eventuellen Paritätsfehlern bei der Eingabe erzeugt werden.

Hinweis

Ob die Paritätsprüfung bei der Eingabe aktiviert oder deaktiviert ist, hängt nicht davon ab, ob die Paritäts-Erkennung aktiviert oder deaktiviert ist (siehe auch Abschnitt „Steuermodi“ auf Seite 109). Wenn die Paritätserkennung aktiviert, die Paritätsprüfung bei der Eingabe jedoch deaktiviert ist, dann erkennt zwar die Hardware, mit der das Terminal verbunden ist, das Paritätsbit, aber die `Terminal`-Gerätedatei überprüft nicht, ob dieses Bit korrekt gesetzt ist.

Bei gesetztem `INLCR` wird ein empfangenes `NL`-Zeichen (Zeilenvorschub) in ein `CR`-Zeichen (Wagenrücklauf) umgewandelt. Bei gesetztem `IGNCR` wird ein empfangenes `CR`-Zeichen ignoriert (nicht gelesen). Ist dagegen `IGNCR` nicht gesetzt und `ICRNL` gesetzt, so wird ein empfangenes `CR`-Zeichen in ein `NL`-Zeichen umgewandelt.

Bei gesetztem `IUCLC` wird ein empfangener Großbuchstabe in den entsprechenden Kleinbuchstaben umgewandelt. (Wird zukünftig vom X/Open-Standard nicht mehr unterstützt.)

Bei gesetztem IXON wird die Ausgabesteuerung mit START/STOP aktiviert. Bei Empfang eines STOP-Zeichens wird die Ausgabe angehalten und bei Empfang eines START-Zeichens fortgesetzt. Die Steuerzeichen für START und STOP werden bei einer Leseoperation nicht gelesen, führen jedoch die Funktionen der Flusssteuerung aus, wenn IXON gesetzt ist. Ist IXON nicht gesetzt, dann werden START- und STOP-Zeichen gelesen. Bei gesetztem IXANY wird die angehaltene Ausgabe durch die Eingabe eines beliebigen Zeichens fortgesetzt.

Bei gesetztem IXOFF ist die Eingabe-Flusssteuerung aktiviert. Das System überträgt STOP-Zeichen, um das Terminal zu veranlassen, keine weiteren Daten mehr zu übertragen, wenn dies notwendig ist, um ein Überlaufen des Eingabepuffers zu verhindern (nicht mehr als {MAX_INPUT} Byte sind erlaubt). Es überträgt START-Zeichen, um das Terminal zu veranlassen, die Übertragung von Daten wieder aufzunehmen, sobald dies wieder ohne Gefahr eines Überlaufs des Eingabepuffers möglich ist.

Der Anfangswert für die Eingabemodi nach open() ist, dass kein Bit gesetzt ist.

Ausgabemodi

Die Komponente c_oflag gibt an, wie die Ausgaben der Terminalschnittstelle behandelt werden. Sie wird durch bitweises inklusives Oder der folgenden Masken erzeugt, die sich bitweise unterscheiden. Die Maskennamen in der folgenden Tabelle sind in termios.h definiert:

Maskenname	Beschreibung
OPOST	Ausgaben nachbearbeiten
OLCUC	Bei Ausgabe Klein- in Großbuchstaben umwandeln. Wird zukünftig vom X/Open-Standard nicht mehr unterstützt.
ONLCR	Bei Ausgabe NL in CR–NL umwandeln
OCRNL	Bei Ausgabe CR in NL umwandeln
ONOCR	CR in Spalte 0 nicht ausgeben
ONLRET	NL übernimmt CR-Funktion
OFILL	Füllzeichen für Verzögerung verwenden
OFDEL	Das Füllzeichen ist DEL (sonst NUL)
NLDLY NL0 NL1	Zeilenvorschub-(NL-)Verzögerungen auswählen: NL-Zeichen Typ 0 NL-Zeichen Typ 1
CRDLY CR0 CR1 CR2 CR3	Wagenrücklauf-(CR-)Verzögerungen auswählen: CR-Verzögerung Typ 0 CR-Verzögerung Typ 1 CR-Verzögerung Typ 2 CR-Verzögerung Typ 3

Maskenname	Beschreibung
TABDLY	Horizontaltabulator-Verzögerungen auswählen:
TAB0	Horizontaltabulator-Verzögerung Typ 0
TAB1	Horizontaltabulator-Verzögerung Typ 1
TAB2	Horizontaltabulator-Verzögerung Typ 2
TAB3	Tabulatorexpansion zu Leerzeichen
BSDLY	Backspace-Verzögerungen auswählen:
BS0	Backspace-Verzögerung Typ 0
BS1	Backspace-Verzögerung Typ 1
VTDLY	Vertikaltabulator-Verzögerungen auswählen:
VT0	Vertikaltabulator-Verzögerung Typ 0
VT1	Vertikaltabulator-Verzögerung Typ 1
FFDLY	Seitenvorschub-Verzögerungen auswählen:
FF0	Seitenvorschub-Verzögerung Typ 0
FF1	Seitenvorschub-Verzögerung Typ 1

Wenn OPOST gesetzt ist, dann werden Ausgabedaten gemäß den übrigen Bits von c_oflag nachbearbeitet, damit die Textzeilen so verändert werden, dass sie korrekt am Terminal erscheinen, andernfalls werden die Zeichen ohne Änderung übertragen.

Bei gesetztem OLCUC wird ein Kleinbuchstabe vor der Übertragung in den entsprechenden Großbuchstaben umgewandelt. Diese Funktion wird oft zusammen mit IUCLC bei den Eingabemodi verwendet. Wird zukünftig vom X/Open-Standard nicht mehr unterstützt.

Bei gesetztem ONLCR wird das NL-Zeichen (Zeilenvorschub) als das Zeichenpaar CR–NL (Wagenrücklauf-Zeilenvorschub) übertragen. Bei gesetztem OCRNL wird das CR-Zeichen als NL-Zeichen übertragen. Bei gesetztem ONOCR wird ein CR-Zeichen in Spalte 0 (erste Stelle in der Zeile) nicht übertragen. Bei gesetztem ONLRET wird angenommen, dass das NL-Zeichen die Wagenrücklauf-Funktion übernimmt; der Spaltenzeiger wird auf 0 gesetzt und die spezifischen Wagenrücklauf-Verzögerungen werden verwendet. Ist ONLRET nicht gesetzt, so wird angenommen, dass das NL-Zeichen nur die Zeilenvorschub-Funktion hat; der Spaltenzeiger bleibt dann unverändert. Der Spaltenzeiger wird ferner auf 0 gesetzt, wenn das CR-Zeichen übertragen wird.

Die Verzögerungs-Bits geben an, für wie lange die Übertragung angehalten wird, damit bestimmte mechanische oder sonstige Bewegungen bei der Übertragung bestimmter Zeichen am Terminal ausgeführt werden können. In allen Fällen bedeutet 0: keine Verzögerung. Bei gesetztem OFILL wird die zeitliche Verzögerung durch die Übertragung von Füllzeichen erreicht. Dies ist bei Terminals mit hoher Übertragungsgeschwindigkeit nützlich, die nur eine minimale Verzögerung benötigen. Bei gesetztem OFDEL wird DEL als Füllzeichen verwendet, sonst NUL.

Ist eine Seitenvorschub- oder Vertikaltabulator-Verzögerung angegeben, so dauert diese etwa 2 Sekunden.

Eine Zeilenvorschub-Verzögerung dauert etwa 0,10 Sekunden. Bei gesetztem `ONLRET` werden statt der Zeilenvorschub-Verzögerungen die Wagenrücklauf-Verzögerungen verwendet. Bei gesetztem `OFILL` werden zwei Füllzeichen übertragen.

Bei den Wagenrücklauf-Verzögerungen ist Typ 1 abhängig von der aktuellen Spaltenposition, Typ 2 dauert etwa 0,10 Sekunden, Typ 3 etwa 0,15 Sekunden. Bei gesetztem `OFILL` werden bei Typ 1 zwei Füllzeichen übertragen, bei Typ 2 vier.

Bei den Horizontaltabulatoren-Verzögerungen ist Typ 1 abhängig von der aktuellen Spaltenposition, Typ 2 dauert etwa 0,10 Sekunden, Typ 3 gibt an, dass Tabulatoren zu Leerzeichen expandiert werden sollen. Bei gesetztem `OFILL` werden für jede Verzögerung zwei Füllzeichen übertragen.

Die Backspace-Verzögerung dauert etwa 0,05 Sekunden. Bei gesetztem `OFILL` wird ein Füllzeichen übertragen.

Die tatsächlichen Verzögerungen hängen von der Leitungsgeschwindigkeit und der Systemauslastung ab.

Der Anfangswert für die Ausgabemodi (Wert von `c_oflag`) nach einem Aufruf von `open()` ist, dass kein Bit gesetzt ist.

Steuermodi

Die unten beschriebenen Steuermodi haben für BS2000-Rechner keine Bedeutung.

Die Komponente `c_cflag` beschreibt die hardwaremäßige Steuerung des Terminals, dabei werden für zeichenorientierte Terminals folgende Elemente unterstützt:

Maskenname	**Beschreibung**
`CLOCAL`	Zustand von Modem ignorieren
`CREAD`	Empfänger zulassen

`CSIZE`
`CS5`
`CS6`
`CS7`
`CS8`
	Anzahl der Bits je Byte:
	5 Bits
	6 Bits
	7 Bits
	8 Bits

`CSTOPB`	2 Stoppbits senden (sonst 1)
`HUPCL`	Bei letztem `close()` Verbindung abbauen
`PARENB`	Parität zulassen
`PARODD`	ungerade Parität zulassen

Zusätzlich werden die Ein- und Ausgabebaudraten in der Struktur `termios` abgespeichert. Die folgenden Werte werden unterstützt:

Name	Beschreibung
`B0`	Verbindung abbauen (Hang Up)
`B50`	50 Baud
`B75`	75 Baud
`B110`	110 Baud
`B134`	134.5 Baud
`B150`	150 Baud
`B200`	200 Baud
`B300`	300 Baud
`B600`	600 Baud
`B1200`	1200 Baud
`B1800`	1800 Baud
`B2400`	2400 Baud
`B4800`	4800 Baud
`B9600`	9600 Baud
`B19200`	19200 Baud
`B38400`	38400 Baud

Die folgenden Schnittstellen stehen für das Ermitteln und Setzen der Werte für Ein- und Ausgabebaudrate in der Struktur `termios` zur Verfügung: `cfgetispeed()`, `cfgetospeed()`, `cfsetispeed()` und `cfsetospeed()`.

Mit den `CSIZE`-Bits wird die Anzahl der Bits je Byte sowohl für die Übertragung als auch für den Empfang angegeben. Darin ist das Paritätsbit, sofern vorhanden, nicht enthalten. Bei gesetztem `CSTOPB` werden zwei Stoppbits verwendet (sonst eins). Bei einer Übertragungsgeschwindigkeit von 110 Baud werden z.B. normalerweise zwei Stoppbits verwendet.

Bei gesetztem `CREAD` wird der Empfänger aktiviert. Ist `CREAD` nicht gesetzt, werden keine Zeichen empfangen.

Bei gesetztem `PARENB` wird die Paritätserkennung und die Paritätserzeugung aktiviert, d.h. jedes Zeichen erhält ein Paritätsbit. In diesem Fall gibt das `PARODD`-Bit an, dass eine ungerade Parität verwendet wird (sonst wird eine gerade Parität verwendet).

Bei gesetztem `HUPCL` wird die Verbindung abgebaut, wenn der letzte Prozess, der diese Leitung benutzt, die Verbindung schließt oder sich beendet. Das heißt, das Data-Terminal-Ready-Signal (`DTR`) wird zurückgesetzt. Dadurch wird die Verbindung abgebrochen.

Bei gesetztem CLOCAL wird angenommen, dass es sich bei der bestehenden Leitung um eine lokale, direkte Verbindung ohne Modemsteuerung handelt. Die Verbindung hängt dann nicht von den Leitungssignalen ab. Ansonsten wird eine Modemsteuerung angenommen und die Melde-Signale werden überwacht.

Unter normalen Umständen wartet ein Aufruf der Funktion open() auf das Ende des Verbindungsaufbaus. Wenn jedoch das Bit O_NONBLOCK beim Aufruf von open() angegeben wird oder das Bit CLOCAL gesetzt ist, dann kehrt die Funktion open() sofort zurück, ohne auf die Verbindung zu warten.

Wenn das Objekt, für das die Steuermodi gesetzt sind, keine asynchrone serielle Verbindung ist, können einige der Modi ignoriert werden; wird z.B. der Versuch unternommen, die Baudrate für eine Netzverbindung zu einem Terminal an einem anderen Rechner zu setzen, kann die Baudrate für die Verbindung zwischen Terminal und Rechner, mit dem sie direkt verbunden ist, gesetzt werden oder nicht.

Der Anfangswert für die Steuermodi (Wert von c_cflag) nach einem Aufruf von open() ist, dass kein Bit gesetzt ist.

Lokalmodi

Die Komponente c_lflag der Struktur wird verwendet, um verschiedene Funktionen zu steuern:

Maskenname	Beschreibung
ECHO	Echo-Funktion aktivieren
ECHOE	ERASE-Zeichen als BS–SP–BS ausgeben ("Echo") (korrigierender Backspace)
ECHOK	NL-Zeichen nach KILL-Zeichen ausgeben ("Echo")
ECHONL	NL-Zeichen ausgeben ("Echo")
ICANON	Standard-Eingabeverarbeitung aktivieren (zeilenorientierte Eingabe mit Behandlung von ERASE- und KILL-Zeichen)
IEXTEN	Erweiterte Funktionen aktivieren
ISIG	Signalaktivierung
NOFLSH	Leeren der Ein- und Ausgabepuffer nach INTERRUPT oder QUIT von Tastatur deaktivieren
TOSTOP	Signal SIGTTOU bei Ausgabe für Hintergrund-Prozessgruppe senden
XCASE	Standardmäßige Darstellung von Groß-/Kleinbuchstaben. Wird zukünftig vom X/Open-Standard nicht mehr unterstützt.

Bei gesetztem ECHO werden eingegebene Zeichen so, wie sie empfangen wurden, wieder auf den Bildschirm ausgegeben. Ist ECHO nicht gesetzt, dann werden Eingabezeichen nicht angezeigt.

Sind ICANON und ECHOE gesetzt, so wird das ERASE-Zeichen als die Folge Backspace-Leerzeichen-Backspace zurückgeliefert, wodurch das letzte Zeichen, sofern vorhanden, auf dem Bildschirm gelöscht wird. Ist ECHOE gesetzt und ECHO nicht, so wird das ERASE-Zeichen als SP BS zurückgeliefert.

Bei gesetztem ECHOK und ICANON wird nach dem KILL-Zeichen ein NL-Zeichen auf den Bildschirm ausgegeben und damit angezeigt, dass die Zeile gelöscht wird, oder die Zeile wird vom Bildschirm gelöscht.

Wenn ECHONL und ICANON gesetzt sind, dann wird ein NL-Zeichen auch dann ausgegeben, wenn ECHO nicht gesetzt ist. Dies ist bei Terminals im lokalen Echo-Modus (sog. Halbduplexbetrieb) nützlich. Ein EOF-Zeichen wird nur dann auf den Bildschirm ausgegeben, wenn es entwertet ist. Da das EOT-Zeichen (Ende der Übertragung) standardmäßig als EOF-Zeichen verwendet wird, kann man auf diese Weise eine Verbindungsauflösung durch Terminals, die sich bei Empfang von EOT abmelden, verhindern.

Bei gesetztem ISIG wird bei jedem eingegebenen Zeichen geprüft, ob es sich um eins der Steuerzeichen INTR, QUIT oder SUSP (nur bei Auftragssteuerung) handelt. Ist dies der Fall, so wird die dazugehörige Funktion ausgeführt. Ist ISIG nicht gesetzt, so wird diese Prüfung nicht durchgeführt. D.h., diese Sonderfunktionen für die Eingabe können nur bei gesetztem ISIG durchgeführt werden. Sie können aber auch einzeln ausgeschaltet werden, indem man ihnen einen unwahrscheinlichen oder unmöglichen Wert als Steuerzeichen zuordnet (z.B. 0377).

Bei gesetztem ICANON wird die Standard-Eingabeverarbeitung aktiviert. Dies aktiviert die Funktionen zur Behandlung von ERASE- und KILL-Zeichen. Die Eingabezeichen werden zeilenweise zusammengefasst, das Ende einer Zeile wird mit NL, EOF oder EOL angegeben, so wie dies im Abschnitt „Standard-Eingabeverarbeitung“ auf Seite 99 beschrieben wurde.

Ist ICANON nicht gesetzt, werden Leseaufträge direkt aus dem Eingabepuffer bedient. Dies geschieht erst dann, wenn mindestens MIN Zeichen empfangen wurden oder wenn der Timer TIME abgelaufen ist (siehe Abschnitt „Besondere Eingabeverarbeitung“ auf Seite 100). Die Angabe des TIME-Wertes erfolgt in Zehntelsekunden. Bei gesetztem NOFLSH findet die normalerweise nach Empfang der Zeichen QUIT, INTR und SUSP (nur für Auftragssteuerung) durchgeführte Löschung der Ein- und Ausgabepuffer nicht statt.

Der Anfangswert für die Lokalmodi (Wert von c_local) nach einem Aufruf von open() ist, dass kein Bit gesetzt ist.

Steuerzeichen

Die Werte der Steuerzeichen werden durch den Vektor c_cc definiert. Die Namen für die jeweiligen Indizes in diesem Vektor sowie die Beschreibungen jedes Vektorelements sowohl für die Standard-Eingabeverarbeitung als auch für die besondere Eingabeverarbeitung werden in der folgenden Tabelle aufgeführt:

Indexname im		Beschreibung
Standardmodus	**besonderer Modus**	
VEOF		EOF-Zeichen
VEOL		EOL-Zeichen
VERASE		ERASE-Zeichen
VINTR	VINTR	INTR-Zeichen
VKILL		KILL-Zeichen
	VMIN	Wert für MIN
VQUIT	VQUIT	QUIT-Zeichen
VSUSP	VSUSP	SUSP-Zeichen
	VTIME	Wert für TIME
VSTART	VSTART	START-Zeichen
VSTOP	VSTOP	STOP-Zeichen

Die Indexnamen sind Konstanten, die den Index des jeweiligen Elements (Zeichens) im Vektor c_cc darstellen. So ist z.B. das Zeichen c_cc[VSTOP] sowohl im Standard-Eingabemodus als auch im besonderen Eingabemodus das STOP-Zeichen.

Die Indexnamen sind eindeutig, außer dass VMIN und VTIME jeweils dieselben Werte wie VEOF und VEOL haben können.

Implementierungen wie das POSIX-Subsystem, die die Auftragssteuerung nicht unterstützen, können den Wert für das SUSP-Zeichen ignorieren, das im Vektor c_cc durch VSUSP indiziert wird.

Wenn {_POSIX_VDISABLE} für die Terminal-Gerätedatei definiert ist und der Wert eines der änderbaren Sonderzeichen gleich {_POSIX_VDISABLE} ist (siehe Abschnitt „Sonderzeichen“ auf Seite 102), dann wird diese Funktion deaktiviert. Das heißt, kein Eingabezeichen wird als das deaktivierte Sonderzeichen erkannt. Wenn ICANON nicht gesetzt ist, dann hat der Wert von {_POSIX_VDISABLE} keine besondere Bedeutung für die Einträge mit den Indizes VMIN und VTIME im Vektor c_cc.

Blockterminalunterstützung

Das Terminal wird auf die Gerätedatei `/dev/tty` abgebildet. Terminal-Ein-/Ausgabe bedeutet somit Ein-/Ausgabe auf die Gerätedatei `/dev/tty`. Der Eingabepuffer und der Ausgabepuffer für `/dev/tty` ist jeweils 12 264 Byte groß. Es werden nur die Steuerzeichen `\n` (Zeilenende) und `\t` (Tabulator 8-Zeichen-Abstand) umgesetzt.

Die Eingabe von EM DÜ1 wird als Zeilenende `\n` interpretiert. Die Tabulatortaste erzeugt kein Tabulatorzeichen `\t`. Die Eingabe vom Terminal wird gepuffert. Wenn Restdaten im Puffer sind, liefert der Aufruf von `read()` nur maximal so viele Bytes zurück, wie im Puffer enthalten sind. Erst wenn der Puffer leer ist, wird der Anwender zur Eingabe vom Terminal aufgefordert.

Die Eingabe kann nicht mit K2 abgebrochen werden. Erst nach dem Einlesen vom Terminal wird in den Systemmodus gewechselt. Das heißt, Sie müssen einmal EM DÜ1 eingeben, bevor Sie in den Systemmodus gelangen.

Bei der Ausgabe bewirkt `\n` einen Zeilenvorschub und `\t` einen Tabulator. Alle anderen Steuerzeichen werden nicht umgesetzt, sie werden nur als Schmierzeichen abgebildet. Die Ausgabe erfolgt bei folgenden Ereignissen:

– ein Zeilenende-Zeichen (`\n`) wird erkannt
– der Puffer ist voll
– eine Eingabe vom Terminal erfolgt (abgeschlossen durch EM DÜ1)
– das Programm beendet sich

Unterstützung der BS2000-Console

Die Gerätedatei `/dev/console` kann nur zum Schreiben geöffnet werden. Hierzu muss sie mit `open("/dev/console")` geöffnet werden. Der Ausgabepuffer für `/dev/console` ist 230 Byte groß.

Prozesssteuerung

Im POSIX-Subsystem findet der Programmablauf in einem Prozess statt, im BS2000 in einer Task. Wird ein Programm in der POSIX-Shell aufgerufen, wird ein Sohnprozess erzeugt. Wird ein Programm von der BS2000-Kommandoschnittstelle aufgerufen, wird kein Prozess erzeugt.

Signale

Wenn ein Programm mit dem POSIX-Bindeschalter gebunden wurde, wickelt das C-Laufzeitsystem die Signalbehandlung über die XPG4 Version 2-konformen Möglichkeiten des POSIX-Subsystems ab.

Wenn ein Programm im BS2000 aufgerufen wird, wird die Signalbehandlung über die Mechanismen im BS2000 (STXIT) realisiert.

Mit der Funktion `cstxit()` können jedoch - an der POSIX-Signalbehandlung vorbei - STXIT-Routinen am System angemeldet werden. Trotzdem wird davor gewarnt, diese Möglichkeit zu nutzen.

Ansonsten dürfen POSIX- und STXIT-Signale nicht im selben Programm behandelt werden.

Die Signalbehandlung setzt auf den Funktionen `signal()`, `sigaction()`, `sigprocmask()` und `kill()` auf. Für jedes Signal sind drei Einstellungen möglich (siehe `sigaction()`).

Bei Abbruch des Prozesses wird die Nummer des auslösenden Signals, die Adresse, an der das Programm abgebrochen wird, und die Frage, ob ein Speicherabzug gewünscht wird, ausgegeben.

Alle im POSIX-Subsystem unterstützten Signale sind in der Include-Datei `signal.h` und im entsprechenden Abschnitt in diesem Handbuch beschrieben. Die Signale werden beim Eintreten des zugehörigen Ereignisses generiert.

Es gibt für den Anwender gewisse Einschränkungen:

– Eine angemeldete `STXIT`-Routine wird immer vor einer angemeldeten Signalbehandlung vom System aufgerufen. Wenn kein Signal angemeldet wurde, wird auch keine angemeldete `STXIT`-Routine aufgerufen.

– Auf keinen Fall sollen Contingency-Routinen oberhalb von Level 125 angemeldet werden, um damit die implizite `TU`-Contingency der Signalbehandlung nicht zu unterbrechen.

– Für folgende Signale ist eine Dialogtaste definiert:

Signal	Dialogtaste
SIGINT	INTR
SIGQUIT	QUIT
SIGSTOP	STOP
SIGTSTP	SUSP
SIGCONT	START

Die Tasten STOP und START werden auf Blockterminals nicht unterstützt (siehe auch Abschnitt „Blockterminalunterstützung“ auf Seite 114).

Interprozesskommunikation

Die Funktionen der Interprozesskommunikation beeinflussen andere Dienste. Die betroffenen Funktionen werden in der nachfolgenden Tabelle aufgeführt:

Beeinflusste Schnittstellen		
errno	execve()	execl()
execvp()	execle()	exit()
execlp()	fork()	execv()

Allgemeine Beschreibung

Das Paket Interprozesskommunikation umfasst drei Mechanismen:

– Nachrichten (messages) sind formatgebundene Datenströme, die von Prozessen an beliebige andere Prozesse gesendet werden können (dazu werden folgende Systemaufrufe verwendet: msgget(), msgsnd(), msgrcv(), msgctl()).

– Gemeinsam nutzbare Speicherbereiche (shared memory) erlauben, dass Prozesse Teile ihres virtuellen Adressraumes mit anderen Prozessen teilen (dazu werden folgende Systemaufrufe verwendet: shmget(), shmat(), shmdt(), shmctl()).

– Semaphoren ermöglichen die Synchronisation der Ausführung von Prozessen (dazu werden folgende Systemaufrufe verwendet: semget(), semop(), semctl()).

Die den drei Mechanismen gemeinsamen Aspekte werden nachfolgend beschrieben. Die Beschreibung gliedert sich in die Abschnitte:

– Einrichten eines Kommunikationselements (Nachrichten-Warteschlange, gemeinsam nutzbarer Speicherbereich, Semaphore)

– Datenstrukturen

– Statusinformationen abfragen oder ändern

Dabei steht *xxx* jeweils für `msg`, `sem` oder `shm`.

Jedes Kommunikationselement (Nachrichten-Warteschlange, Gemeinsamer Speicherbereich, Semaphor) wird durch eine positive ganze Zahl identifiziert. Die Nummer wird beim Einrichten des Kommunikationselements *xxx*`get()` vom System vergeben. Der Benutzer kann zusätzlich einen Zahlenschlüssel als Namen eines von ihm erzeugten Kommunikationselements festlegen.

Zu jedem Mechanismus existiert eine Tabelle, deren Einträge alle Kommunikationselemente des jeweiligen Mechanismus enthalten.
Dabei enthält jeder Eintrag einen vom Benutzer gewählten Zahlenschlüssel als Namen, durch den der Eintrag identifiziert wird.

Einrichten eines Kommunikationselements

Für jeden Mechanismus gibt es einen Systemaufruf *xxx*`get()`, mit dem ein neues Element erzeugt werden kann oder ein bereits existierendes Element für einen Prozess verfügbar gemacht werden kann. Die Parameter der Systemaufrufe *xxx*`get()` sind ein vom Benutzer gewählter Zahlenschlüssel *key* als Benutzername und ein Schalter *xxxflg*.

key Das Betriebssystem sucht in der zugehörigen Tabelle nach einem Eintrag, der durch den Schlüssel bezeichnet wird. Prozesse können den Systemaufruf *xxx*`get()` mit dem Schlüssel `IPC_PRIVATE` aufrufen; damit wird sichergestellt, dass ein unbenutzter Eintrag zurückgegeben wird.

xxxflg Der Schalter beeinflusst, ob und wie auf einen Eintrag zugegriffen werden kann, sowie gegebenenfalls die Zugriffsrechte. Wenn der Schalter `IPC_CREAT` gesetzt wird, wird ein neuer Eintrag erzeugt, falls noch keiner existiert. Die gewünschten Zugriffsrechte werden durch Bit-ODER mit `IPC_CREAT` kombiniert. Für den neuen Eintrag werden dann die neun rechten Bits des Schalters als Zugriffsrechte gesetzt. Die Anordnung der Bits entspricht der von *oflag* im Systemaufruf `open()`, wenngleich nur die Lese- und Schreibberechtigung von Bedeutung sind.

Falls bereits ein Eintrag mit dem angegebenen Schlüssel existiert, müssen die neun rechten Bits des Schalters eine Teilmenge der Zugriffsrechte des Eintrags sein, andernfalls scheitern die Systemaufrufe *xxx*`get()`. Es können also keine weitergehenden Rechte gefordert werden, als vorhanden sind. Zum Verändern der Zugriffsrechte muss der Systemaufruf *xxx*`ctl()` abgesetzt werden (s.u.). Wenn der Schalter `IPC_CREAT` zusätzlich durch Bit-ODER mit dem Schalter `IPC_EXCL` kombiniert wird, kehrt *xxx*`get()` mit

einem Fehler zurück, falls für den Schlüssel bereits ein Eintrag existiert. Wenn der Schalter `IPC_CREAT` nicht gesetzt ist, muss bereits ein Eintrag existieren, andernfalls scheitern die Systemaufrufe *xxx*`get()`.

Die Systemaufrufe *xxx*`get()` liefern eine vom Betriebssystem ausgewählte eindeutige positive ganze Kennzahl (Systemkennzahl *xxxid*), die in den anderen, dem jeweiligen Mechanismus zugehörigen Systemaufrufen verwendet wird. Die Kennzahlen funktionieren wie die Dateideskriptoren - wie sie z. B. `open()`, `dup()` und `pipe()` liefern -, mit der Ausnahme, dass jeder Prozess, der ihren Wert kennt, sie verwenden kann. D.h. sie müssen nicht vererbt werden, um gültig zu sein. Jeder gemeinsam nutzbare Speicherbereich (shared memory), jede Nachrichtenwarteschlange und jede Semaphorenmenge wird so durch die Kennzahl für gemeinsam nutzbaren Speicherbereich (*shmid*), die Semaphorenkennzahl (*semid*) bzw. die Warteschlangenkennzahl (*msqid*) identifiziert.

Datenstrukturen

Jeder Kennzahl ist eine Datenstruktur zugeordnet, die die operationsbezogenen Daten der durchzuführenden oder durchgeführten Operationen enthält. Diese Datenstrukturen (`msqid_ds`, `semid_ds`, `shmid_ds`) sind in `sys/shm.h`, `sys/sem.h` und `sys/msg.h` beschrieben. Unter anderem enthalten diese Datenstrukturen die Prozessnummer des letzten Prozesses, der eine Operation durchgeführt hat (Nachricht senden oder empfangen, auf gemeinsamen Speicher zugreifen usw.), und die Zeit des letzten Zugriffs.

Alle Datenstrukturen enthalten Eigentümerinformationen und eine `ipc_perm`-Struktur (siehe `sys/ipc.h`), auf Grund der Prozessen, die IPC-Funktionen verwenden, Schreib-/Leserechte (bei Semaphoren Änder-/Leserechte) erteilt oder verweigert werden. Die `ipc_perm`-Struktur enthält die effektive Benutzer- und Gruppennummer des Prozesses, der den Eintrag erstellt hat (*xxx_perm.cuid* und *xxx_perm.cgid*), sowie eine Benutzer- und eine Gruppennummer (*xxx_perm.uid* und *xxx_perm.gid*), die auch durch den Systemaufruf *xxx*`ctl()` gesetzt werden können. Dazu kommt ein Bitfeld von Zugriffsrechten in der Komponente `mode` der `ipc_perm`-Struktur. Die Bits sind wie folgt belegt:

Bit	Bedeutung
0400	lesen (Eigentümer)
0200	schreiben (Eigentümer)
0040	lesen (Gruppe)
0020	schreiben (Gruppe)
0004	lesen (Andere)
0002	schreiben (Andere)

Die Struktur vom Typ `ipc_perm` hat den Namen `shm_perm`, `sem_perm` oder `msg_perm`, je nach verwendetem Mechanismus. Lese- und Schreib-/ Änderungsberechtigungen werden einem Prozess erteilt, wenn eine oder mehrere der folgenden Bedingungen zutreffen.

– Die effektive Benutzernummer eines Prozesses mit Sonderrechten.
– Die effektive Benutzernummer des Prozesses ist identisch mit *xxx_perm.cuid* oder *xxx_perm.uid* in der der IPC-Kennzahl zugeordneten Datenstruktur, und das entsprechende Bit für Eigentümer in *xxx_perm.mode* ist gesetzt.
– Die effektive Benutzernummer des Prozesses ist nicht identisch mit *xxx_perm.cuid* oder *xxx_perm.uid*, aber die effektive Gruppennummer des Prozesses ist identisch mit *xxx_perm.cgid* oder *xxx_perm.gid* in der der IPC-Kennzahl zugeordneten Datenstruktur, und das entsprechende Bit für Gruppe in *xxx_perm.mode* ist gesetzt.
– Die effektive Benutzernummer des Prozesses ist nicht identisch mit *xxx_perm.cuid* oder *xxx_perm.uid*, und die effektive Gruppennummer des Prozesses ist nicht identisch mit *xxx_perm.cgid* oder *xxx_perm.gid* in der der IPC-Kennzahl zugeordneten Datenstruktur, aber das entsprechende Bit für Andere in *xxx_perm.mode* ist gesetzt.

In allen anderen Fällen wird keine Berechtigung erteilt.

Statusinformationen abfragen oder ändern

xxx`ctl()`

Für jeden Mechanismus gibt es einen Systemaufruf *xxx*`ctl()`, mit dem der Status eines Eintrags abgefragt werden kann, Statusinformation gesetzt oder ein Eintrag aus dem System entfernt werden kann.

– Fragt ein Prozess den Status eines Eintrags ab, so prüft das Betriebssystem, ob der Prozess die Leseberechtigung hat, und kopiert danach Daten aus dem Tabelleneintrag in die vom Benutzer angegebene Struktur.

– Will ein Prozess die Parameter des Eintrags neu setzen, dann prüft das Betriebssystem, ob die effektive Benutzernummer des Prozesses mit der Benutzernummer des Eintrags oder mit der Benutzernummer des Erstellers des Eintrags übereinstimmt bzw. ob die effektive Benutzernummer die eines Prozesses mit Sonderrechten ist. Um Parameter neu zu setzen, ist Schreibberechtigung allein nicht ausreichend. Das Betriebssystem kopiert die vom Benutzer angegebenen Daten in den Tabelleneintrag, setzt dabei die Benutzernummer, die Gruppennummer, die Zugriffsrechte und andere von der Art des Mechanismus abhängige Felder. Nicht verändert werden die Felder mit der Benutzer- und Gruppennummer des Erstellers des Eintrags; dadurch verbleiben dem Ersteller des Eintrags immer Kontrollrechte.

– Will ein Prozess einen Eintrag entfernen, stellt das Betriebssystem sicher, dass die effektive Benutzernummer des Prozesses mit einer der Benutzernummern in der `ipc_perm`-Struktur übereinstimmt. Nachdem ein Eintrag entfernt wurde, ist es nicht mehr möglich, mit der alten Kennzahl auf den Eintrag zuzugreifen.

Hinweis

Die Verwendung der IPC-Mechanismen verlangt hohe Sorgfalt, da nicht nutzbare oder nicht benötigte IPC-Elemente vom Betriebssystem nicht in allen Fällen erkannt werden. Im Betriebssystem gibt es keine Aufzeichnungen darüber, welche Prozesse auf ein IPC-Element zugreifen - in der Tat kann jeder Prozess auf ein IPC-Element zugreifen, dem die richtige Kennzahl bekannt ist und der zugriffsberechtigt ist, auch wenn er nie einen Systemaufruf *xxx*`get()` abgesetzt hat. Deshalb kann das Betriebssystem die IPC-Strukturen nicht implizit bereinigen (z.B. bei Prozessende).

Die IPC-Mechanismen sollten nur bei extremen Performance-Anforderungen verwendet werden.

Gemeinsam nutzbarer Speicher

Für die C-Bibliotheksfunktionen wird gemeinsam nutzbarer Speicherbereich zur Verfügung gestellt (siehe Handbuch „BS2000/OSD-BC Makroaufrufe an den Ablaufteil“).
Mit `shmget()` wird gemeinsam nutzbarer Speicherbereich angelegt. Auf Nicht-XS-Anlagen in Einheiten zu 64 Kbyte, die auf 64 Kbyte-Grenzen ausgerichtet sind, und auf XS-Anlagen in Einheiten zu 1 Mbyte im oberen Adressraum auf 1 Mbyte-Grenze ausgerichtet. Das Argument *size* der Funktion `shmget()` wird entsprechend aufgerundet.

`shmget()` liefert eine Kennzahl für gemeinsam nutzbaren Speicherbereich zurück. `shmat()` legt den gemeinsam nutzbaren Speicherbereich an.

Die Verbindung zum gemeinsam nutzbaren Speicherbereich wird nach einem `shmdt`-Aufruf oder bei Programmende abgebrochen. Die zugehörige Kennzahl für den gemeinsam nutzbaren Speicherbereich wird freigegeben:

– mit `shmctl()`

– nachdem sich der letzte an diesem gemeinsam nutzbaren Speicher beteiligte Prozess mit `shmdt()` abgemeldet hat

– bei Programmende

Erst danach kann dieselbe Kennzahl für gemeinsam nutzbaren Speicherbereich wieder verwendet werden.

Für den gemeinsam nutzbaren Speicher stehen maximal 150 Kennzahlen im BS2000 zur Verfügung. Pro Programm sind maximal 32 Aufrufe der Funktion `shmat()` möglich.

Um gemeinsam nutzbare Speicherbereiche mit der Steuerfunktion `SHM_LOCK` der Funktion `shmctl()` sperren zu können, muss bei `/START-PROGRAM` der Operand `RESIDENT-PAGES` angegeben werden.

Contingency- und STXIT-Routinen

Dieser Abschnitt gibt Hinweise, wie in C Contingency- bzw. STXIT-Routinen realisiert werden können.

Die für das Verständnis notwendige und ausführliche Beschreibung des Contingency-STXIT-Konzepts sowie der entsprechenden BS2000-Systemmakros finden Sie im Handbuch „BS2000/OSD-BC Makroaufrufe an den Ablaufteil“.

Die ausführliche Beschreibung der in diesem Abschnitt erwähnten Bibliotheksfunktionen (`signal()`, `raise()`, `alarm()`, `cenaco()`, `cdisco()`, `cstxit()`, `longjmp()`, `setjmp()`) finden Sie im Nachschlageteil dieses Handbuchs.

Achtung

Die Verwendung einiger C-Bibliotheksfunktionen innerhalb von `STXIT`-Routinen kann zu undefiniertem Verhalten führen. Die Konsistenz der Bibliotheksfunktionen kann bei asynchronen Unterbrechungen nicht immer gewährleistet werden. Zu undefiniertem Verhalten kommt es, wenn innerhalb der `STXIT`-Routine die gleiche bzw. eine zur gleichen Gruppe gehörende Bibliotheksfunktion (siehe Auflistung unten) ausgeführt werden soll, die durch das `STXIT`-Ereignis asynchron unterbrochen wurde.

„Kritische“ C-Bibliotheksfunktionen im Zusammenhang mit asynchronen Unterbrechungen sind:

– Dateizugriffsfunktionen zum Öffnen und Schließen von Dateien: `fopen()`, `freopen()`, `open()`, `creat()`, `fclose()`, `close()`
– Alle Dateizugriffs-, Dateiverwaltungs- und Ein-/Ausgabe-Funktionen, die auf die gleiche Datei angewendet werden
– Zufallsgeneratorfunktionen: `rand()`, `srand()`
– Zeitfunktionen: `localtime()`, `gmtime()`
– Funktionen zum An- und Abmelden von Contingency-Routinen: `cenaco()`, `cdisco()`
– `atexit()`
– `strtok()`
– `setlocale()`
– Ein-/Ausgabefunktionen aus der C++-Standardbibliothek

Die C-Bibliotheksfunktionen alarm(), raise(), signal()

Das Konzept von Contingency-Routinen bzw. STXIT-Contingency-Routinen ist für C-Programme vorrangig durch folgende C-Bibliotheksfunktionen abgedeckt:

`alarm()` Signal `SIGALRM` senden (`STXIT`-Ereignis `RTIMER`)

`raise()` Signale senden (simulierte `STXIT`-Ereignisse und benutzerdefinierte Ereignisse)

`signal()` Signalbearbeitungs-Routinen zuordnen

STXIT-Contingency-Routinen

Mit `alarm()`, `raise()` und `signal()` lassen sich folgende STXIT-Ereignisklassen behandeln:

- Programmüberprüfung (`PROCHK`)
- Intervallzeitgeber CPU-Zeit (`TIMER`)
- Ende der Programmlaufzeit (`RUNOUT`)
- nicht behebbarer Programmfehler (`ERROR`)
- Mitteilung an das Programm (`INTR`)
- nur im Dialog: BREAK/ESCAPE (`ESCPBRK`)
- `ABEND`
- Normale Programmbeendigung (`TERM`)
- Intervallzeitgeber Realzeit (`RTIMER`)

Die Ereignisklasse SVC-Unterbrechung wird derzeit nicht unterstützt.

Ereignisgesteuerte Routinen

Mit `signal()` und `raise()` lassen sich über zwei vom Benutzer definierbare Signale (`SIGUSR1`, `SIGUSR2`) zwei ereignisgesteuerte Routinen realisieren.

Die Ereignissteuerung über C-Bibliotheksfunktionen funktioniert nur innerhalb einer Task, d.h. die Kommunikation zwischen verschiedenen Tasks ist nicht möglich.
Diese ereignisgesteuerten Routinen sind deshalb intern nicht als Contingency-Routinen, sondern über eine `CALL`-Schnittstelle realisiert.

Freie Verwendung von Contingency-Routinen

Bei speziellen Anforderungen, die durch `signal()` und `raise()` nicht abgedeckt sind, können die entsprechenden BS2000-Funktionen für Ereignissteuerung frei programmiert werden. Solche Anforderungen sind z.B. eine größere Anzahl von Ereignissen (mit `raise()` und `signal()` lassen sich nur zwei Ereignisse selbst definieren) oder Inter-Task-Kommunikation (mit `raise()` und `signal()` ist Ereignissteuerung nur innerhalb einer Task möglich).

Funktionen zur eigentlichen Ereignissteuerung, wie etwa das Starten der ereignisgesteuerten Verarbeitung (Signale senden und empfangen) müssen in Assembler-Programmteilen mit den entsprechenden BS2000-Makroaufrufen (`POSSIG`, `SOLSIG`, `ENAEI`) realisiert werden.

Die Makros zum Anmelden, Abmelden und Beenden von Contingency-Prozessen (`ENACO`, `DISCO`, `RETCO`) dürfen jedoch nicht im Assembler-Programmteil verwendet werden. Statt dieser Makros müssen die C-Bibliotheksfunktion `cenaco()` bzw. `cdisco()` aufgerufen werden. `cenaco()` und `cdisco()` führen neben dem An- und Abmelden einer Contingency-Routine Aktionen durch, die für die Konsistenz-Sicherung des C-Laufzeitstacks notwendig sind.

Die Contingency-Routine selbst kann sowohl in C als auch in Assembler geschrieben werden. Die Beendigung dieser Routine muss mit einem "normalen" Rücksprung erfolgen (in C mit `return()` bzw. `longjmp()`, in Assembler mit `@EXIT`).

Contingency-Routine in C

Der Routine wird bei ihrem Anlauf ein Strukturparameter übergeben, der in der Include-Datei `cont.h` folgendermaßen deklariert ist:

```
struct contp
{
int     comess;         /* contingency message */
evcode indicat;        /* information indicator */
char    filler[2];      /* reserved for int. use */
evcode switchc;        /* event switch */
int     pcode;          /* post code */
int     reg4;           /* register 4 */
int     reg5;           /* register 5 */
int     reg6;           /* register 6 */
int     reg7;           /* register 7 */
int     reg8;           /* register 8 */
};

#define evcode     char
#define _normal    0       /* evceventnormal */
#define _abnormal 4       /* evceventabnormal */
```

```
#define _nmnpc     0       /* evcnocomessnopostcode */
#define _mnpc      4       /* evccomessnopostcode */
#define _nmpc      8       /* evcnocomesspostcode */
#define _mpc       12      /* evccomesspostcode */
#define _etnm      0       /* evcelapsedtimenocomess */
#define _etm       4       /* evcelapsedtimecomess */
#define _disnm     16      /* evceventdisablednocomess */
#define _dism      20      /* evceventdisabledcomess */
```

Wenn der oben beschriebene Strukturparameter ausgewertet werden soll, muss die C-Routine einen formalen Parameter für eine Struktur vom Typ `contp` vorsehen und ist dann etwa folgendermaßen aufgebaut:

```
#include <cont.h>

void controut (struct contp contpar)
{
...
   return ...;
}
```

Die C-Routine kann auf eine der folgenden zwei Arten beendet werden:

– mit der `return`-Anweisung. Dann wird das Programm an der unterbrochenen Stelle fortgesetzt.
– durch Aufruf der Funktion `longjmp()`. Dann wird das Programm bei der mit einem `setjmp`-Aufruf definierten Stelle fortgesetzt.

Contingency-Routine in Assembler

Die Contingency-Routine muss z.B. dann in Assembler geschrieben werden, wenn in ihr weitere BS2000-Makroaufrufe erfolgen sollen (etwa SOLSIG zur Erneuerung der Contingency-Routine).

Ein strukturiertes ILCS-Assemblerprogramm für eine Contingency-Routine hat etwa folgenden Aufbau:

```
PARLIST DSECT
COMESS   DS     F
IND      DS     C
FILLER   DS     CL2
EC       DS     C
...
CONTROUT @ENTR TYP=E,ILCS=YES
USING PARLIST,R1
...
SOLSIG
...
@EXIT
```

In der Contingency-Routine darf der RETCO-Makro nicht aufgerufen werden.
Die Rückkehr muss mit dem Makro @EXIT erfolgen.

Freie Verwendung von STXIT-Contingency-Routinen

Bei speziellen Anforderungen, die durch die Funktion signal() nicht abgedeckt sind, können STXIT-Contingency-Routinen in C frei programmiert werden. Solche Anforderungen sind z.B. umfangreichere Informationsübergaben oder mehr Fortsetzungs-Steuermöglichkeiten nach Ablauf der STXIT-Contingency-Routine.

Die Definition einer frei programmierten STXIT-Contingency-Routine muss durch Aufruf der C-Bibliotheksfunktion cstxit() erfolgen.

Die Ereignisklasse SVC-Unterbrechung kann auch bei Verwendung der cstxit-Funktion nicht realisiert werden.

Der STXIT-Contingency-Routine wird bei ihrem Anlauf eine Struktur übergeben, die in der Include-Datei stxit.h folgendermaßen deklariert ist:

```
struct stxcontp
{
int      *intwghtp;     /* pointer to interrupt weight */
jmp_buf *termlabp;     /* pointer to termination label */
int      *regsp;        /* pointer to register save area */
};
```

Aufbau der STXIT-Contingency-Routine

Um die oben beschriebene Struktur benutzen zu können, muss die Routine einen formalen Parameter für eine Struktur vom Typ stxcontp vorsehen und ist dann etwa folgendermaßen aufgebaut:

```
#include <stxit.h>

void stxrout(stxcontpar)
struct stxcontp stxcontpar;
{
/* ... */
}
```

Diese Routine kann auf drei verschiedene Arten beendet werden:

– mit der return-Anweisung, das Programm wird an der unterbrochenen Stelle fortgesetzt,
– durch Aufruf der Funktion longjmp() mit einer durch einen setjmp-Aufruf versorgten Variablen vom Typ jmp_buf, das Programm wird bei der mit einem setjmp-Aufruf definierten Stelle fortgesetzt oder
– durch Aufruf der Funktion longjmp() mit dem in der stxcontp-Struktur übergebenen Termination-Label.

Die Rückkehr aus der STXIT-Contingency-Routine mit einem `longjmp`-Aufruf ist bei der Ereignisklasse `TERM` nicht möglich, da bei Eintritt des Ereignisses (`TERM-SVC`) die Einträge für die C-Funktionen einschließlich der `main`-Funktion im C-Laufzeitstack bereits abgebaut sind.

Threadsichere C-Laufzeitbibliothek durch Unterstützung von POSIX-Threads

Programme, die mit den im XPG5-Standard beschriebenen POSIX-Threads arbeiten, setzen voraus, dass die Funktionen des Laufzeitsystems threadsicher sind.

Zur Gewährleistung der Threadsicherheit der C-Laufzeitbibliothek muss der Zugriff auf globale Ressourcen (Dateien, globale Daten aus den C-Globals) verboten bzw. durch einen LOCK geschützt werden, so dass zu jedem Zeitpunkt maximal ein Thread auf diese Ressourcen zugreifen kann. Die Aufrufschnittstelle der Funktionen ändert sich dadurch nicht. Ein aufrufender Thread-1 kann jedoch durch einen Thread-2 blockiert werden, der die angeforderten Ressourcen gerade belegt. Erst wenn Thread-2 die Ressourcen freigibt, kann Thread-1 auf diese zugreifen.

Ab CRTE V2.3A wird deshalb zusätzlich eine threadsichere Variante ausgeliefert.

Im Einzelnen wird die Threadsicherheit der Laufzeit-Bibliothek durch folgende Mechanismen realisiert:

- exklusiver Zugriff auf Objekte vom Typ (FILE *)

  Alle Funktionen, die auf Objekte des Typs (FILE *) zugreifen, verhalten sich so, als würden sie intern die Funktionen `flockfile()` und `funlockfile()` verwenden, um exklusiv in den Besitz dieser (FILE *) Objekte zu kommen.

- exklusiver Zugriff auf globale Daten, die in den Globals verankert sind

  Funktionen, die auf globale Daten zugreifen, werden durch einen LOCK geschützt.

- errno ist thread-spezifisch

  `errno` zählt nicht mehr wie bisher zu den globalen Daten, sondern ist thread-spezifisch. Für jeden Thread eines Prozesses gilt daher, dass der Wert von `errno` nicht durch Funktionsaufrufe oder Wertzuweisungen an `errno` durch einen anderen Thread beeinflusst wird.

- POSIX-Thread-Funktionen

  POSIX-Thread-Funktionen realisieren den exklusiven Zugriff auf Objekte vom Typ (FILE*).

  Es gibt folgende Kategorien von POSIX-Thread-Funktionen:

  – POSIX-Thread-Funktionen, die reentrant sind (mit ergänzendem „_r“ im Funktionsnamen)
  – POSIX-Thread-Funktionen , die automatisch durch LOCK geschützt sind
  – POSIX-Thread-Funktionen zum Sperren und Entsperren von Objekten des Typs (FILE*)
  – POSIX-Thread-Funktionen zur expliziten Sperrung von Clients
  – POSIX-Thread-Funktionen mit Wirkung auf den Prozess oder auf einen Thread

  Die einzelnen POSIX-Thread-Funktionen sind ausführlich beschrieben im Kapitel „Funktionen und Variablen alphabetisch (a - m)“ (siehe Seite 165).

- erweiterte Header-Dateien

  Die folgenden Header-Dateien enthalten zur Unterstützung der POSIX-Threads zusätzliche Funktionsprototypen, Datentypen und Konstanten:

  – <dirent.h>
  – <grp.h>
  – <pthread.h>
  – <pwd.h>
  – <sched.h>
  – <signal.h>
  – <stdio.h>
  – <stdlib.h>
  – <string.h>
  – <time.h>
  – <unistd.h>

  Die einzelnen Header-Dateien sind ausführlich beschrieben im Kapitel „Include-Dateien“ (siehe Seite 961).

Einzelne Funktionen sind auch weiterhin nicht threadsicher und dürfen in Programmen mit Multithreading nicht verwendet werden. Bei der Beschreibung dieser Funktionen im Kapitel „Funktionen und Variablen alphabetisch (a - m)“ (siehe Seite 165) wird darauf hingewiesen.

Es wurden außerdem die Funktionen der Gruppe _POSIX_THREAD_SAVE_FUNCTIONS aufgenommen. Eine Auflistung dieser Funktionen finden auf Seite 144. Ausführlich beschrieben sind diese Funktionen im Kapitel „Funktionen und Variablen alphabetisch (a - m)“ (siehe Seite 165).

Programmierhinweise

Returnwerte und Ergebnisparameter

Returnwert Zeiger

```
<typ> *funct(...)
```

Etliche Funktionen, die einen Zeiger zurückliefern, schreiben ihr Ergebnis in einen C-internen Datenbereich, der bei jedem Aufruf einer solchen Funktion überschrieben wird. Weil dies eine häufige Fehlerquelle ist, wird bei Funktionen vom Datentyp Zeiger auf diesen Umstand hingewiesen.

Returnwert void *

```
void * funct(...)
```

Wenn der Funktionswert einer `void *`-Funktion einer Zeigervariablen zugewiesen wird, sollte der Typ mit dem cast-Operator explizit umgewandelt werden. Beim Aufruf aus C++-Quellen ist die explizite Typumwandlung obligatorisch.

Beispiel

```
long *long_ptr;
.
.
long_ptr (long *)calloc(20, sizeof(long));
```

Returnwert int

`int` *`funct()`*`;`

Funktionen zur Zeichenbearbeitung haben einen Returnwert vom Typ `int`, da sie `EOF` (=-1) zurückliefern können. Wenn die Funktion einen Returnwert vom Typ `char` zurückliefert, läuft ein Programm auf einen Fehler.

Ergebnisparameter Zeiger

```
<typ1> funct(<typ2> *variable)
```

Ergebnisparameter sind Variablen, deren Inhalt durch die Funktion verändert wird. D.h., in solche Variablen speichert die Funktion ein Ergebnis ab. Ergebnisparameter sind ohne den Zusatz `const` definiert.

Als Argument ist stets die Adresse, d.h. ein Zeiger, zu übergeben. Außerdem müssen Sie vor Aufruf der Funktion den Speicherplatz für das Ergebnis explizit bereitstellen.
Weil man dies häufig vergisst, wird in den jeweiligen Funktionsbeschreibungen darauf hingewiesen.

Beispiele

```
struct timeb tp;    /* Struktur */
ftime(&tp);

char erg;           /* char-Variable */
scanf("%c", &erg);

char array[10];     /* Zeichenketten-Variable */
scanf("%s", array);
```

Fehlerbehandlung

Es ist im Sinne effektiver Programmierung bei den meisten Funktionsaufrufen vorteilhaft zu prüfen, ob die Funktion erfolgreich ausgeführt wurde. Dies kann z.B. wie folgt geschehen:

```
if(fct(...) == error result){     /* Abfrage auf Fehler-Returnwert */
               perror("fct:");    /* Ausgabe von Fehlerinformationen */
               exit(error code); /* Reaktion auf den Fehler, hier z.B. */
       }                         /* Programmbeendigung */
else...
```

Wenn bei der Abarbeitung einer Funktion ein Fehler auftritt, wird dies in den meisten Fällen durch den Returnwert -1 oder den Nullzeiger angezeigt; Einzelheiten sind im Kapitel „Funktionen und Variablen alphabetisch (a - m)“ auf Seite 165ff zu finden. Sofern dies von der Funktion vorgesehen ist, wird im Fehlerfall zusätzlich die externe Variable `errno` gesetzt. Der Wert dieser Variablen ist nur nach dem Aufruf einer Funktion definiert, für die ausdrücklich angegeben wird, dass sie diese Variable besetzt, und bis zu ihrer Änderung durch einen nachfolgenden Funktionsaufruf. Die Variable `errno` sollte nur dann überprüft werden, wenn dies durch den Wert des Funktionsergebnisses angezeigt oder jeweils im Abschnitt „Hinweis“ für eine Funktion angegeben ist. Keine Bibliotheksfunktion in diesem Handbuch setzt `errno` gleich 0, um einen Fehler anzuzeigen.

`errno` wird bei erfolgreichen Funktionsaufrufen nicht zurückgesetzt. Bei einigen Funktionen kann nur durch Prüfen von `errno` festgestellt werden, ob die Funktion erfolgreich war.

Auf Grund der in `errno` gesetzten Fehlernummer werden intern Daten aufbereitet, die den Fehler näher spezifizieren. Mit der Funktion `perror()` kann die Fehlermeldung auf die Standard-Ausgabe ausgegeben werden. Diese Fehlermeldung beinhaltet einen kurzen Fehlertext, der den Fehler erläutert.

Wenn bei der Abarbeitung eines Funktionsaufrufs mehr als ein Fehler auftritt, kann ein beliebiger der von der Funktion vorgesehenen Fehler zurückgeliefert werden, da die Reihenfolge ihrer Entdeckung undefiniert ist.

Alle Fehlernummern, auf die `errno` gesetzt werden kann, sowie die dafür vorgesehenen Fehlerinformationen sind in der Include-Datei `errno.h` definiert. Eine vollständige Liste finden Sie in `errno.h`.

Wenn bei einer Funktion verschiedene Arten von Fehlern und damit Fehlernummern möglich sind, kann es sinnvoll sein, die `errno`-Variable auf die Fehlernummer abzufragen, um dann ggf. unterschiedlich darauf reagieren zu können. Jede Fehlernummer wird durch eine in `errno.h` definierte symbolische Konstante repräsentiert, z.B. bedeutet `ERANGE` Überlauffehler.

Eine Abfrage könnte etwa folgendermaßen aussehen, z.B. hier bei der Funktion `signal()`:

```
#include <errno.h>
...
errno = 0;
...
if(signal(sig, fct) == 1){    /* Abfrage des Fehlerergebnisses */
   if (errno == EFAULT)
   ...                        /* Reaktionen auf EFAULT */
   else if(errno == EINVAL)
   ...                        /* Reaktionen auf EINVAL */
}
else...
```

Der Abschnitt „Fehler“ bei jeder Funktionsbeschreibung im Kapitel „Funktionen und Variablen alphabetisch (a - m)“ auf Seite 165 gibt an, unter welchen Bedingungen ein Fehler auftritt.

Testmöglichkeiten

Wenn das Programm mit der Option `TEST-SUPPORT=YES` übersetzt wird, können Sie alle Möglichkeiten von AID zum Testen Ihrer Programme benutzen (siehe Handbuch "AID - Testen von C/C++Programmen"). Ausnahme: keine Unterstützung von AID beim Zugang zu POSIX über rlogin oder telnet (AID funktioniert nur im Blockterminal-Modus).

Funktionen und Variablen thematisch

In diesem Abschnitt finden Sie eine Zusammenstellung der Funktionen nach thematischen Gesichtspunkten.

Dateibearbeitung

Dateiverwaltung

Dateizugriff

64-Bit-Funktionen zur Unterstützung von NFS V3.0

Ein-/Ausgabe

Prozesse

Prozessverwaltung

Prozesssteuerung und Signale

Interprozesskommunikation

Diagnose und Meldungen

Unterstützung von POSIX-Threads

Reentrante POSIX-Thread-Funktionen (Gruppe _POSIX_THREAD_SAVE_FUNCTIONS)

Bei diesen Funktionen mit Suffix „_r“ im Funktionsnamen handelt es sich um die reentrante Variante der entsprechenden Funktion ohne Suffix „_r“. Da diese Funktionen auch für das Arbeiten ohne Threads nützlich sind, werden sie auch in der nicht threadfesten Variante des CRTE ($.SYSLNK.CRTE) ausgeliefert.

„asctime_r - Datum und Uhrzeit threadsicher in Zeichenkette umwandeln“ auf Seite 179

„ctime_r - Datum und Uhrzeit threadsicher in Zeichenkette umwandeln“ auf Seite 238

„getgrgid_r - Gruppendateieintrag für eine Gruppen-ID threadsicher ermitteln“ auf Seite 431

„getgrnam_r - Gruppendateieintrag für Gruppenname threadsicher ermitteln“ auf Seite 433

„getlogin_r - Benutzerkennung threadsicher ermitteln“ auf Seite 439

„getpwnam_r - Benutzernamen threadsicher ermitteln“ auf Seite 453

„gmtime_r - Datum und Uhrzeit threadsicher in UTC umwandeln“ auf Seite 472

„localtime_r - Datum und Uhrzeit threadsicher in Zeichenkette umwandeln“ auf Seite 551

„rand_r - Pseudo-Zufallszahlen (int) threadsicher generieren“ auf Seite 670

„readdir_r - aus Dateiverzeichnis threadsicher lesen“ auf Seite 677

„strtok_r - Zeichenkette threadsicher in Tokens zerlegen“ auf Seite 820

„ttyname_r - Pfadnamen eines Terminals threadsicher ermitteln“ auf Seite 876

Bei der Arbeit mit Threads sind diese Funktionen anstelle der korrespondierenden Funktionen, die kein Suffix „_r“ enthalten, zu verwenden. Der Einsatz der genannten Funktionen ist jedoch auch in einer Nicht-Thread-Umgebung vorteilhaft.

POSIX-THREAD-Funktionen zum Sperren und Entsperren von Objekten des Typs (FILE*)

„flockfile, ftrylockfile, funlockfile - Funktionen zum Sperren der Standardein/-ausgabe“ auf Seite 311

POSIX-Thread-Funktionen zur expliziten Sperrung von Clients

Die folgenden Funktionen sind identisch zu den entsprechenden Funktionen ohne „_unlocked“ im Namen:

„getc_unlocked, getchar_unlocked, putc_unlocked, putchar_unlocked - Standardeingabe/-ausgabe mit expliziter Sperrung durch den Client“ auf Seite 412

Der Anwender muss hierbei selbst die Threadsicherheit garantieren, indem er die verwendeten Objekte des Typs (FILE*) durch den Aufruf der Funktionen flockfile bzw. ftrylockfile sperrt und durch Aufruf der Funktion funlockfile entsperrt.

POSIX-Thread-Funktionen mit Wirkung auf den Prozess oder auf einen Thread

Mit der Realisierung der POSIX-Threads ist zu unterscheiden zwischen

– Funktionen, die sich (wie bisher) auf den Prozess und damit auf alle zum Prozess gehörenden Threads auswirken, und

– Funktionen, die sich nur auf einen speziellen Thread beziehen

Bei Signalen ist ebenfalls zu unterscheiden, ob sie an den (gesamten) Prozess oder an einen bestimmten Thread geschickt werden.

Es handelt sich um folgende Funktionen::

„abort - Prozess abbrechen“ auf Seite 169

„alarm - Alarmsignal steuern“ auf Seite 176

„atexit - Prozessendefunktion registrieren“ auf Seite 183

„exit, _exit - Prozess beenden“ auf Seite 269

„fcntl - offene Datei steuern“ auf Seite 283

„fork - neuen Prozess erzeugen“ auf Seite 325

„getcontext, setcontext - Benutzerkontext anzeigen oder ändern“ auf Seite 415

„getpid - Prozessnummer ermitteln“ auf Seite 448

„getrlimit - Grenzwert für ein Betriebsmittel ermitteln“ auf Seite 456

„getpriority - Prozesspriorität abrufen“ auf Seite 450

„kill - Signal an Prozess oder Prozessgruppe senden“ auf Seite 528

„lockf - Dateiabschnitt sperren“ auf Seite 552

„msgrcv - Nachricht aus Warteschlange empfangen“ auf Seite 617

„msgsnd - Nachricht an Warteschlange senden“ auf Seite 620

„nice - Priorität eines Prozesses ändern“ auf Seite 630

„open - Datei öffnen“ auf Seite 633

„pause - Prozess bis zum Empfang eines Signals anhalten“ auf Seite 645

„raise - Signal an aufrufenden Prozess senden“ auf Seite 668

„read - Bytes aus Datei lesen“ auf Seite 672

„semop - Semaphor-Operationen durchführen“ auf Seite 719

„setcontext - Benutzerkontext ändern“ auf Seite 724

„setlocale - Lokalität ändern oder ermitteln“ auf Seite 728

„sigaction - Signalbehandlung ermitteln oder ändern“ auf Seite 749

„sigpause - Signal aus Signalmaske entfernen und Prozess deaktivieren“ auf Seite 770

„sigpending - blockierte Signale ermitteln“ auf Seite 771

„sigsetjmp - Marke für nichtlokalen Sprung durch Signal setzen“ auf Seite 775

„sigsuspend - auf Signal warten“ auf Seite 779

„sleep - Prozess für festgesetzte Zeitspanne anhalten“ auf Seite 781

„usleep - Prozess für festgesetzte Zeitspanne anhalten“ auf Seite 891

„wait, waitpid - auf Halt oder Ende eines Kindprozesses warten“ auf Seite 906

„wait3 - auf Zustandsänderung von Kindprozessen warten“ auf Seite 910

„waitid - auf Zustandsänderung von Kindprozessen warten“ auf Seite 911

„write - Bytes in Datei schreiben“ auf Seite 953

Bei folgenden Funktionen wird beim `EPIPE`-Fehler das Signal `SIGPIPE` nicht an den Prozess, sondern an den aufrufenden Thread gesendet:

„fclose - Datenstrom schließen“ auf Seite 281

„fflush - Datenstrom leeren“ auf Seite 297

„fputc - Byte in Datenstrom schreiben“ auf Seite 343

„fputwc - Langzeichen in Datenstrom schreiben“ auf Seite 346

„fseek - Lese-/Schreibzeiger im Datenstrom auf aktuellen Wert positionieren“ auf Seite 366

„write - Bytes in Datei schreiben“ auf Seite 953

Funktionen, die nicht threadsicher sind

Alle Funktionen, die in der C-Laufzeitbibliothek definiert sind, werden threadsicher ausgeliefert. Eine Ausnahme bilden lediglich die folgendenen Funktionen:

„asctime - Datum und Uhrzeit in Zeichenkette umwandeln“ auf Seite 178 [1]

„basename - letztes Element eines Pfadnamens zurückgeben“ auf Seite 188

„brk, sbrk - Größe des Datensegments verändern“ auf Seite 190

„chroot - Root-Verzeichnis ändern“ auf Seite 217

„ctime - Datum und Uhrzeit in Zeichenkette umwandeln“ auf Seite 237 [1]

„cuserid - Benutzerkennung ermitteln“ auf Seite 239

„dbmclearerr - Funktion zur Verwaltung von dbm-Datenbasen“ auf Seite 241

„dirname - Vaterverzeichnis zu einem Pfadnamen liefern“ auf Seite 245

„ecvt, fcvt, gcvt - Gleitpunktzahl in Zeichenkette umwandeln“ auf Seite 251

„endgrent, getgrent, setgrent - Gruppenverwaltung“ auf Seite 254

„endpwent, getpwent, setpwent - Benutzerkatalog verwalten“ auf Seite 256

„endutxent, getutxent, getutxid, getutxline, pututxline, setutxent, utmpx-Einträge verwalten“ auf Seite 258

„fcvt - Gleitpunktzahl in Zeichenkette umwandeln“ auf Seite 289

„gamma - Logarithmus der Gamma-Funktion berechnen“ auf Seite 408

„gcvt - Gleitpunktzahl in Zeichenkette umwandeln“ auf Seite 409

„getdtablesize - Größe der Deskriptor-Tabelle abrufen“ auf Seite 426

„getenv - Wert einer Umgebungsvariablen ermitteln“ auf Seite 427

„getgrent - Gruppendatei-Eintrag bestimmen“ auf Seite 429

„getpwent - Benutzerdaten aus dem Benutzerkatalog lesen“ auf Seite 451

„getutxent, getutxid, getutxline - auf utmpx-Eintrag zugreifen“ auf Seite 466

„getgrgid - Gruppendateieintrag für Gruppennummer ermitteln“ auf Seite 430 [2]

„getgrnam - Gruppendateieintrag für Gruppenname ermitteln“ auf Seite 432 [2]

„getlogin - Benutzerkennung ermitteln“ auf Seite 438 [2]

„getpagesize - aktuelle Seitengröße ausgeben“ auf Seite 445

[1] reentrante Funktion (Extension „_r“) nutzen

[2] reentrante Funktion (Extension „_r“) nutzen

[1] reentrante Funktion `rand_r()` nutzen

[2] Das Ergebnis eines Aufrufs dieser Funktionen ist undefiniert, wenn die Struktur `jmp_buf` nicht im aufrufenden Thread initialisiert wurde.

[3] reentrante Funktion `rand_r()` nutzen

Hinweis Wenn Sie irgendeine der `_POSIX_THREAD_SAFE_FUNCTIONS` oder `_POSIX_THREADS` Schnittstellen benutzen, müssen Sie die Funktionen `ctermid()` und `tmpnam()` mit einem Parameter ungleich dem Nullzeiger aufrufen, um threadsicher zu sein. Andernfalls wird das Ergebnis in einen internen statischen Bereich geschrieben, was zu undefiniertem Verhalten führen kann.

[4] Das Ergebnis eines Aufrufs dieser Funktionen ist undefiniert, wenn die Struktur `jmp_buf` nicht im aufrufenden Thread initialisiert wurde.

[5] Funktion `pthread_sigmask` nutzen

[6] reentrante Funktion (Extension „_r“) nutzen

Speicherverwaltung und Speicheroperationen

Systemumgebung

Zeichen und Zeichenketten

Einzelne Zeichen bearbeiten

Zeichenketten bearbeiten

Zeichen und Zeichenketten umwandeln

„wctob - Langzeichen in (1-Byte) Multibyte-Zeichen umwandeln“ auf Seite 944

Umwandlung von Größen

Reguläre Ausdrücke

Zeitfunktionen

Mathematische Funktionen

Arithmetik mit ganzen Zahlen

Arithmetik mit Gleitkommazahlen

Trigonometrische, hyperbolische und Arcus-Funktionen

„atan - Arcustangens berechnen“ auf Seite 181

„atan2 - Arcustangens von x/y berechnen“ auf Seite 182

„cos - Cosinus berechnen“ auf Seite 226

„cosh - Cosinus hyperbolicus berechnen“ auf Seite 227

„sin - Sinus berechnen“ auf Seite 780

„sinh - Sinus hyperbolicus berechnen“ auf Seite 780

„tan - Tangens berechnen“ auf Seite 845

„tanh - Tangens hyperbolicus berechnen“ auf Seite 845

Zufallszahlen

„drand48 - Pseudo-Zufallszahlen zwischen 0.0 und 1.0 generieren“ auf Seite 247

„erand48 - Pseudo-Zufallszahlen zwischen 0.0 und 1.0 mit Startwert generieren“ (siehe drand48)

„initstate, random, setstate, srandom - Pseudozufallszahlen generieren“ auf Seite 481

„jrand48 - Pseudo-Zufallszahlen zwischen -2^{31} und 2^{31} mit Startwert generieren“ (siehe drand48)

„lcong48 - Pseudo-Zufallszahlen (signed long int) generieren“ (siehe drand48)

„lrand48 - Pseudo-Zufallszahlen zwischen 0 und 2^{31} generieren“ (siehe drand48)

„mrand48 - Pseudo-Zufallszahlen zwischen -2^{31} und 2^{31} generieren“ (siehe drand48)

„nrand48 - Pseudo-Zufallszahlen zwischen 0 und 2^{31} mit Startwert generieren“ (siehe drand48)

„random - Pseudo-Zufallszahlen erzeugen“ auf Seite 671

„seed48 - Startwert (int) für Pseudo-Zufallszahlen setzen“ auf Seite 711“ (siehe drand48)

„srand - Pseudo-Zufallszahlen (int) mit Startwert generieren“ auf Seite 783

„srand48 - Startwert (double) für Pseudo-Zufallszahlen setzen“ (siehe drand48)

Such- und Sortierverfahren

Terminalschnittstelle und Datenübertragung

„tcsetpgrp - Vordergrund-Prozessgruppennummer setzen“ auf Seite 855

„bsearch(), hsearch(), lsearch(), search.h.“ auf Seite 874

„unlockpt - Lock von Master/Slave Pseudoterminalpaar aufheben“ auf Seite 890

Datenbankfunktionen

„dbm_clearerr, dbm_close, dbm_delete, dbm_error, dbm_fetch, dbm_firstkey, dbm_nextkey, dbm_open, dbm_store - Funktionen zur Verwaltung von dbm-Datenbasen“ auf Seite 241

Listenbearbeitung

„insque, remque - Element in Queue einfügen oder aus Queue entfernen“ auf Seite 483

Makros für die POSIX-IO

Bei Funktionen der C-Bibliothek, die mit Dateien arbeiten, muss vor der Ausführung der eigentlichen Funktionalität zunächst ermittelt werden, ob es sich um eine Datei im POSIX-Dateisystem oder um eine BS2000-Datei handelt. Wenn Sie schon genau wissen, dass Sie nur mit Dateien aus dem POSIX-Dateisystem arbeiten, könnte dieser Aufwand eingespart und somit eine bessere Performance erzielt werden. Mit CRTE V02.3A soll deshalb für folgende Makros eine spezielle Makrovariante für die Arbeit mit Dateien des POSIX-Dateisystems ausgeliefert werden:

getc (p):Zeichen aus einer Datei einlesen

getchar():Zeichen von Standardeingabe einlesen

putc(x, p):Zeichen in Datei schreiben

putchar(x)Zeichen auf Standardausgabe ausgeben

clearerr (p):Dateiende- und Fehlerflag löschen

feof(p): Test auf Dateiende

ferror(p):Test auf Dateifehler

fileno(p):Dateideskriptor ermitteln

Die Makros sind wie bisher im Header <`stdio.h`> enthalten. Damit die POSIX-spezifische Ausprägung generiert wird, muss der Anwender das Define `__POSIX_MACROS` vor setzen, bevor er <`stdio.h`> einbindet.

Funktionen und Variablen alphabetisch (a - m)

Dieses Kapitel enthält in alphabetischer Reihenfolge die Beschreibungen aller Funktionen bzw. Makros und externen Variablen, die vom C-Laufzeitsystem sowohl im POSIX-Subsystem als auch im BS2000/OSD unterstützt werden.

Strukturmittel

Nach der Überschrift, die den jeweiligen symbolischen Namen und eine stichwortartige Beschreibung der Funktionalität enthält, folgen immer dieselben Unterabschnitte:

Syntax
Syntax des Funktionsaufrufs oder der Variablendeklaration und der Include-Datei, in der die jeweilige Schnittstelle definiert bzw. deklariert ist.

Eine Syntaxzeile kann zusätzlich wie folgt gekennzeichnet sein:

Optional
Eine so gekennzeichnete Include-Anweisung muss in neuerstelltem Quellcode nicht angegeben werden. Aus bestehendem Quellcode muss sie nicht gelöscht werden. Das Ende eines so gekennzeichneten Abschnittes wird durch die Endemarke ❑ markiert.

Beschreibung
Beschreibung der Funktionalität einer Funktion bzw. eines Makros oder einer externen Variablen und Erläuterung der anzugebenden Argumente.

Returnwert
Aufzählung und Beschreibung der möglichen Returnwerte einer Funktion.

Nicht jede Funktion liefert einen Returnwert zurück. In solchen Fällen und bei der Beschreibung von externen Variablen gibt es keinen Abschnitt „Returnwert“.

Fehler
Aufzählung und Beschreibung der symbolischen Fehlernummern, die bei einem fehlerhaften Aufruf oder Ablauf einer Funktion in der externen Variablen `errno` abgelegt werden.

Nicht jede Funktion legt bei einem Fehler auch eine Fehlernummer in `errno` ab. In solchen Fällen und bei der Beschreibung von externen Variablen gibt es keinen Abschnitt „Fehler“.

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

Siehe auch Querverweise auf Funktionsbeschreibungen, Include-Dateien, Abschnitte im Konzept-Kapitel oder andere Handbücher.

Nicht weiter gekennzeichnete Textabschnitte beschreiben XPG4-konforme Implementierungen. Für Abweichungen vom Standard gibt es folgende Kennzeichnungen:

BS2000
Informationen über Erweiterungen des C-Laufzeitsystems, die sich auf Funktionalität beziehen im Zusammenhang mit dem Zugriff auf das DVS und auf C-Laufzeitversionen bis V2.1C (BS2000-Funktionalität). Das Ende eines so gekennzeichneten Abschnittes wird durch die Endemarke ❑ markiert.
Wenn eine Funktion eine Erweiterung ist, die aus der bisherigen C-Bibliothek (BS2000) übernommen wurde, ist sie in der Überschrift mit *(BS2000)* gekennzeichnet.

Erweiterung
Informationen über Erweiterungen des C-Laufzeitsystems. Das Ende eines so gekennzeichneten Abschnittes wird durch die Endemarke ❑ markiert.
Wenn eine Funktion eine Erweiterung ist, wie sie auf vielen UNIX-Systemen unterstützt wird, ist sie mit *(Erweiterung)* gekennzeichnet.

Einschränkung

Informationen über derzeitige Einschränkungen des C-Laufzeitsystems gegenüber dem XPG4. Das Ende eines so gekennzeichneten Abschnittes wird durch die Endemarke ❑ markiert.

a64l, l64a - Konvertierung einer Zeichenkette in 32-Bit-Integerzahl

Syntax

```
#include <stdlib.h>

long a64l (const char *s);
char *l64a (long value);
```

Beschreibung

Diese Funktionen werden zum Verwalten von Zahlen verwendet, die in ASCII-Zeichen zur Basis 64 gespeichert sind. Diese Zeichen definieren eine Notation, mit der lange ganze Zahlen durch maximal sechs Zeichen dargestellt werden können; jedes Zeichen stellt eine 'Ziffer' in einer Schreibweise gemäß Basis 64 dar.

Die für die Darstellung von 'Ziffern' verwendeten Zeichen sind `.` für 0, `/` für 1, `0` bis einschließlich `9` für 2-11, `A` bis einschließlich `Z` für 12-37 und `a` bis einschließlich `z` für 38-63.

`a64l()` erwartet einen Zeiger auf eine mit Null-Byte abgeschlossene Basis-64-Darstellung und gibt den entsprechenden `long`-Wert zurück. Wenn die Zeichenkette, auf die `s` zeigt, mehr als sechs Zeichen enthält, verwendet `a64l()` die ersten sechs Zeichen. War die übergebene Zeichenkette leer, ist der Returnwert 0L.

`a64l()` durchläuft die Zeichenkette von links nach rechts (mit der kleinsten signifikanten Ziffer links) und decodiert jedes Zeichen als 6-Bit Zahl zur Basis 64. Wenn der Typ `long` mehr als 32 Bit enthält, erhält das Resultat ein Vorzeichen. Das Verhalten von `a64l()` ist undefiniert, wenn `s` der Nullzeiger ist oder wenn die Zeichenkette, auf die `s` zeigt, nicht durch einen vorhergehenden Aufruf von `l64a()` erzeugt wurde.

`l64a()` erwartet ein `long`-Argument und gibt einen Zeiger auf die entsprechende Basis-64-Darstellung zurück. Wenn das Argument 0 ist, gibt `l64a()` einen Zeiger auf eine Nullzeichenkette zurück. Das Verhalten von `l64a()` ist undefiniert, wenn der Wert des Arguments negativ ist.

Returnwert `a64l()`:

Ganzzahliger Wert vom Typ `long`	für Zeichenketten, die eine wie oben beschriebene Struktur haben.
`0L`	für leere Zeichenketten.
undefiniert	falls *s* der Nullzeiger ist oder wenn die Zeichenkette nicht durch einen vorhergehenden Aufruf von `l64a()` erzeugt wurde. `errno` wird gesetzt, um den Fehler anzuzeigen.

`l64a()`:

Zeiger auf eine Zeichenkette mit der Basis-64-Darstellung
für *value* > 0
Zeiger auf leere Zeichenkette
für *value* = 0
undefiniert für *value* < 0

Fehler `a64l()` schlägt fehl, wenn gilt:

`ERANGE` Das Resultat ist nicht darstellbar.

Hinweis Der von `l64a()` zurückgegebene Wert ist ein Zeiger in einen statischen Puffer, dessen Inhalt bei jedem Aufruf überschrieben wird.

Wenn der Typ `long` mehr als 32 Bit enthält, belegt das Ergebnis von `a64l(l64a(l))` die 32 niederwertigen Bits.

Siehe auch `strtoul()`, `stdlib.h`

abort - Prozess abbrechen

Syntax

```
#include <stdlib.h>

void abort(void);
```

Beschreibung

Wenn die Funktion mit POSIX-Funktionalität aufgerufen wird, verhält sie sich XPG5-konform, wie folgt:

– Wenn das Signal `SIGABRT` nicht abgefangen wird und die Signalbehandlung nicht zurückkehrt, bewirkt `abort()` eine anormale Prozessbeendigung: Das Signal `SIGABRT` wird an den aufrufenden Prozess gesendet, als ob `raise()` mit `SIGABRT` aufgerufen worden wäre. Vor dem Prozessabbruch werden offene Datenströme und Meldungskatalog-Deskriptoren geschlossen, als ob `fclose()` aufgerufen worden wäre. Anschließend werden die für `SIGABRT` voreingestellten Signalaktionen durchgeführt (siehe `signal.h`).

– Der Status, den `abort()` an die Funktionen `wait()` oder `waitpid()` liefert, ist der eines Prozesses, der durch das Signal `SIGABRT` beendet wurde. Wenn das Signal `SIGABRT` blockiert oder ignoriert wird, setzt sich `abort()` darüber hinweg.

– Prozessendefunktionen, die mit `atexit()` registriert wurden, werden nicht aufgerufen.

Werden Threads verwendet, so wirkt sich die Funktion auf den Prozess oder auf einen Thread wie folgt aus:

– Abbrechen des Prozesses und damit aller seiner Threads.

– BS2000
Wenn die Funktion mit BS2000-Funktionalität aufgerufen wird, verhält sie sich abweichend, wie folgt:

– Wenn das Programm keine Signalbehandlungsfunktion vorsieht oder wenn eine solche Funktion zur Unterbrechungsstelle zurückkehrt, wird der Prozess mit `_exit(-1)` abgebrochen. ❑

Hinweis

Das Abfangen des Signals ist deshalb vorgesehen, damit der Anwendungsprogrammierer einen Prozess mit portablen Mitteln abbrechen kann. Damit ist er unabhängig von störenden Einflüssen proprietärer Bibliotheksfunktionen.

Wenn `SIGABRT` weder abgefangen noch ignoriert wird und das aktuelle Dateiverzeichnis das Schreibrecht hat, kann auch ein Speicherabzug erzeugt werden.

Siehe auch

`atexit()`, `exit()`, `kill()`, `raise()`, `signal()`, `stdlib.h`, Abschnitt „Signale“ auf Seite 115.

abs - ganzzahligen Absolutwert berechnen

Syntax #include <stdlib.h>

int abs(int *i*);

Beschreibung

`abs()` berechnet den Absolutwert einer ganzen Zahl *i*.

Returnwert Absolutwert für *i* bei Erfolg.

Hinweis Der Absolutwert der betragsmäßig größten darstellbaren negativen Zahl ist nicht darstellbar. Wird als Argument *i* die betragsmäßig größte negative Zahl (-2^{31}) als Parameter angegeben, wird das Programm mit Fehler beendet.

Siehe auch `cabs()`, `fabs()`, `labs()`, `stdlib.h`.

access - Zugriffsrechte auf eine Datei prüfen

Syntax

```
#include <unistd.h>

int access(const char *path, int amode);
```

Beschreibung

`access()` prüft die Zugriffsrechte der durch das Argument *path* angegebenen Datei entsprechend dem Bitmuster in *amode*. Dabei wird die reale Benutzernummer an Stelle der effektiven und die reale Gruppennummer an Stelle der effektiven verwendet.

Für *amode* können folgende symbolische Konstanten angegeben werden:

`R_OK`	Leserecht prüfen
`W_OK`	Schreibrecht prüfen
`X_OK`	Durchsuchrecht prüfen
`F_OK`	Existenz der Datei prüfen

Der Wert von *amode* ist entweder das bitweise Inklusiv-ODER der zu prüfenden Zugriffsrechte (`R_OK, W_OK , X_OK`) oder die Prüfung auf Existenz `F_OK` (siehe auch `unistd.h`).

Erweiterung
Für *amode* können zusätzliche Werte gültig sein, z.B. wenn ein System erweiterte Zugriffssteuerung hat. ❑

Ein Prozess mit besonderen Rechten kann zwar Dateien durchsuchen, ohne dass das Bit für das Durchsuchrecht gesetzt ist, es wird aber bei Abfrage von `X_OK` kein Erfolg gemeldet.

Returnwert

0	Geforderter Zugriff ist erlaubt.
-1	Zugriff ist nicht erlaubt, `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler

`access()` schlägt fehl, wenn gilt:

`EACCES`	Die Schutzbiteinstellung der Datei erlaubt den geforderten Zugriff nicht, oder für eine Komponente des Pfades existiert kein Durchsuchrecht.

Erweiterung

`EFAULT`	*path* ist eine ungültige Adresse.
`EINTR`	Während des `access`-Systemaufrufs wurde ein Signal abgefangen. ❑
`EINVAL`	Es wurde versucht, auf eine BS2000-Datei zuzugreifen.
`ELOOP`	Die maximale Anzahl der symbolischen Verweise in *path* ist überschritten, oder die maximale Anzahl der symbolischen Verweise ist durch `MAXSYMLINKS` in der Include-Datei `sys/param.h` beschrieben.

ENAMETOOLONG
: Die Länge des Arguments *path* überschreitet {PATH_MAX} oder eine Komponente des Pfadnamens ist länger als {NAME_MAX}.

ENOENT
: Das Argument *path* zeigt auf den Namen einer nicht existierenden Datei oder auf eine leere Zeichenkette.

ENOTDIR
: Eine Komponente des Pfades ist kein Dateiverzeichnis.

EROFS
: Schreibzugriff wurde für eine Datei auf einem Nur-Lesen-Dateisystem angefordert.

Hinweis `access()` wird nur für POSIX-Dateien ausgeführt.

Siehe auch `chmod()`, `stat()`, `unistd.h`.

acos - Arcuscosinus berechnen

Syntax #include <math.h>

double acos(double *x*);

Beschreibung

`acos()` ist die Umkehrfunktion zu `cos()` und berechnet zu einer Gleitpunktzahl *x* aus dem Intervall [-1.0, +1.0] den entsprechenden Winkel im Bogenmaß.

Returnwert arcuscosinus(*x*)
bei Erfolg. Es wird eine Gleitpunktzahl vom Typ `double` aus [0, π] zurückgegeben.

0 wenn *x* außerhalb des Bereichs [-1.0, +1.0] liegt. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `acos()` schlägt fehl, wenn gilt:

`EDOM` Der Wert von *x* liegt nicht im Intervall [-1.0, +1.0].

Hinweis Um einen Fehler sicher abzufangen, sollte `errno` vor Aufruf von `acos()` auf 0 gesetzt werden. Ist nach der Ausführung `errno` ≠ 0, so ist ein Fehler aufgetreten.

Siehe auch `asin()`, `atan()`, `atan2()`, `cos()`, `sin()`, `tan()`, `math.h`.

acosh, asinh, atanh - inverse Hyperbelfunktionen

Syntax #include <math.h>

double acosh (double *x*);
double asinh (double *x*);
double atanh (double *x*);

Beschreibung

`acosh()`, `asinh()` und `atanh()` berechnen jeweils den inversen Hyperbel-Kosinus, den inversen Hyperbel-Sinus bzw. den inversen Hyperbel-Tangens zum Argument *x*.

Returnwert `acosh()`:

`Arch(`*x*`)`	bei Erfolg.
0.0	falls *x* < 1.0. `errno` wird gesetzt, um den Fehler anzuzeigen.

`asinh()` :

`Arsh(`*x*`)`	Die Funktion ist immer erfolgreich.

`atanh()`:

`Arth(`*x*`)`	bei Erfolg.
0.0	falls | *x*| > 1.0. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `acosh()` schlägt fehl, wenn gilt:

`EDOM`	*x* < 1.0.

`atanh()` schlägt fehl, wenn gilt:

`EDOM`	| *x*| > 1.0.

Siehe auch `cosh()`, `sinh()`, `tanh()`, `math.h`.

advance - Muster mit regulärem Ausdruck vergleichen

Syntax #include <regexp.h>

int advance(const char **string*, const char **exbuf*);

Beschreibung

Siehe `regexp()`.

Hinweis Diese Funktion wird zukünftig vom X/Open-Standard nicht mehr unterstützt.

alarm - Alarmsignal steuern

Syntax

#include <unistd.h>

Optional
#include <signal.h> ❑

unsigned int alarm(unsigned int *seconds*);

Beschreibung

`alarm()` veranlasst das System, dem aufrufenden Prozess das Signal `SIGALRM` zu senden, nachdem die Zeitspanne *seconds* in Echtzeit-Sekunden vergangen ist (siehe auch `signal.h`).

Wenn *seconds* gleich 0 ist, wird eine evtl. vorangegangene Alarmanforderung gelöscht.

Alarmanforderungen werden nicht auf den Stack geschrieben: Nur `SIGALRM` kann auf diese Art erzeugt werden. Wenn das Signal `SIGALRM` noch nicht erzeugt wurde, dann verursacht der Aufruf eine Neufestsetzung des Zeitpunkts, zu dem `SIGALRM` erzeugt wird.

Wechselwirkungen zwischen `alarm()` und den Funktionen `setitimer()`, `ualarm()` oder `usleep()` sind undefiniert.

Werden Threads verwendet, so wirkt sich die Funktion auf den Prozess oder auf einen Thread wie folgt aus:

– Ein `SIGALRM`-Signal wird für den Prozess generiert, wenn die angegebene Zeit abgelaufen ist.

BS2000
– Wenn das Signal nicht abgefangen wird (siehe auch `signal()`), wird das Programm mit `exit(-1)` beendet. ❑

Returnwert

Restzeit in Sekunden, bis `SIGALRM` erzeugt worden wäre wenn es einen vorangegangenen `alarm`-Aufruf mit Restzeit in der Alarmuhr gegeben hat.

0 wenn es keinen vorangegangenen `alarm`-Aufruf gegeben hat.

`alarm()` ist immer erfolgreich.

Hinweis

`fork()` löscht anstehende Alarmanforderungen im Sohnprozess. Ein neues Prozessabbild, das durch eine der `exec`-Funktionen erzeugt wurde, übernimmt die bis zu einem Alarmsignal verbleibende Zeit der Alarmuhr aus dem alten Prozessabbild.

Vergabeverzögerungen für den Prozessor können verhindern, dass der Prozess das Signal sofort nach seiner Erzeugung behandelt.

BS2000
`SIGALRM` entspricht der STXIT-Ereignisklasse RTIMER (Intervallzeitgeber Realzeit). ❑

Siehe auch `exec()`, `fork()`, `getitimer()`, `pause()`, `sigaction()`, `ualarm()`, `usleep()`, `signal.h`, `unistd.h`, Abschnitt „Signale“ auf Seite 115.

altzone - Variable für Zeitzone *(Erweiterung)*

Syntax
```
#include <time.h>

extern long int altzone;
```

Beschreibung

Die externe Variable `altzone` beinhaltet die Differenz in Sekunden zwischen UTC (Universal Time Coordinated, 1. Januar 1970) und der alternativen Zeitzone.

`altzone` ist standardmäßig 0 (UTC).

`altzone` wird von `tzset()` gesetzt.

Siehe auch `asctime()`, `ctime()`, `daylight`, `environ`, `gmtime()`, `localtime()`, `setlocale()`, `timezone`, `tzname`, `tzset()`, `time.h`.

ascii_to_ebcdic - ASCII- zu EBCDIC-Zeichenketten konvertieren

(Erweiterung)

Syntax
```
int ascii_to_ebcdic(char *in, char *out);
```

Beschreibung

`ascii_to_ebcdic` konvertiert ASCII- zu EBCDIC-Zeichenketten. Dabei ist *in* die Eingabezeichenkette im ASCII-Code und *out* die Ausgabezeichenkette im EBCDIC-Code. Der Puffer muss vom Aufrufer zur Verfügung gestellt werden.

Die Zeichen der Eingabezeichenkette werden als ASCII-Zeichen interpretiert und in die entsprechenden Zeichen des EBCDIC-Code umgesetzt.

Returnwert

0	bei Erfolg.
1	bei Fehler.

Siehe auch `ebcdic_to_ascii`.

asctime - Datum und Uhrzeit in Zeichenkette umwandeln

Syntax #include <time.h>

char *asctime(const struct tm **timeptr*);

Beschreibung

`asctime()` wandelt eine gemäß der Struktur `tm` (s.u.) aufgeschlüsselte Zeitangabe in eine EBCDIC-Zeichenkette um.

Mit **timeptr* gibt man diese Struktur gemäß der Include-Datei `time.h` an:

```
struct tm
{
          int   tm_sec;        /* Sekunden [0,61] */
          int   tm_min;        /* Minuten [0,59] */
          int   tm_hour;       /* Stunden [0,23] */
          int   tm_mday;       /* Tag des Monats [1,31] */
          int   tm_mon;        /* Monate ab Jahresbeginn [0,11]*/
          int   tm_year;       /* Jahre seit 1900 */
          int   tm_wday;       /* Wochentag [0,6] Sonntag=0 */
          int   tm_yday;       /* Tage seit dem 1. Januar [0,365] */
          int   tm_isdst;      /* Sommerzeitanzeige (immer 0) */
};
```

`asctime()` ist nicht threadsicher. Verwenden Sie bei Bedarf die reentrante Funktion `asctime_r()`.

Returnwert Zeiger auf die erzeugte EBCDIC-Zeichenkette
bei Erfolg. Die Ergebniszeichenkette hat die Länge 26 (einschließlich Nullbyte) und das Format einer Datumsangabe mit Uhrzeit in Englisch:

wochentag monat tag std:min:sek jahr

z.B. `Thu Jun 30 15:20:54 1994\n\0`

Hinweis `Die Funktionen asctime(), ctime(), gmtime()` und `localtime()` schreiben ihr Ergebnis in einen C-internen Datenbereich. Bei jedem Aufruf einer Funktion dieser Gruppe wird daher dieser Datenbereich überschrieben.

Eine Struktur vom Typ `tm` wird von den Funktionen `gmtime()` und `localtime()` geliefert. Diese Funktionen werden aus Kompatibilitätsgründen weiter angeboten. Sie unterstützen weder lokalisierte Daten- noch Zeitformate, d.h. regionale Besonderheiten der Darstellung des Datums oder von Zeitangaben. Um portabel zu sein, sollten Anwendungen statt dessen die Funktion `strftime()` benutzen.

Siehe auch `asctime_r(), clock(), ctime(), difftime(), gmtime(), localtime(), mktime(), strftime(), time(), utime(), time.h`.

asctime_r - Datum und Uhrzeit threadsicher in Zeichenkette umwandeln

Syntax #include <time.h>

char *asctime_r(const struct tm **tm*, char **buf*);

Beschreibung

`asctime_r()` wandelt den Zeitwert, auf den *tm* zeigt, in genau dieselbe Zeitform wie `asctime()` um und schreibt das Ergebnis in den Datenbereich, auf den *buf* zeigt (mit zumindest 26 Bytes).

Returnwert Zeiger auf die Zeichenkette, auf die *buf* zeigt,
bei Erfolg.

Nullzeiger bei Fehler.

Siehe auch `asctime(), ctime(), ctime_r(), localtime(), localtime_r(), time()`.

asin - Arcussinus berechnen

Syntax #include <math.h>

double asin(double *x*);

Beschreibung

`asin()` ist die Umkehrfunktion zu `sin()` und berechnet zur Gleitpunktzahl *x* aus dem Intervall [-1.0, +1.0] den entsprechenden Winkel im Bogenmaß.

Returnwert

arcussinus(*x*)	bei Erfolg. Es wird eine Gleitpunktzahl vom Typ `double` aus [-π/2, +π/2] zurückgegeben.
0.0	für Werte von *x* außerhalb des Intervalls [-1.0, +1.0]. `errno` wird gesetzt, um den Fehler anzuzeigen.
0.0	bei Resultatunterlauf.

Fehler `asin()` schlägt fehl, wenn gilt:

`EDOM`	Der Wert von *x* liegt nicht im Intervall [-1.0, +1.0].

Hinweis Um einen Fehler sicher abzufangen, sollte `errno` vor Aufruf von `asin()` auf 0 gesetzt werden. Ist nach der Ausführung `errno` ≠ 0, so ist ein Fehler aufgetreten.

Siehe auch `acos()`, `atan()`, `atan2()`, `cos()`, `isnan()`, `sin()`, `tan()`, `math.h`.

asinh - inverse Hyperbel-Sinusfunktion

Syntax #include <math.h>

double asinh (double *x*);

Beschreibung

Siehe `acosh()`.

assert - Diagnosemeldungen ausgeben

Syntax #include <assert.h>

void assert(int *expression*);

Beschreibung

`assert()` ist als Makro realisiert. Es stellt fest, ob der Ausdruck *expression* an einer bestimmten Programmstelle falsch (0) ist. Im Fehlerfall schreibt `assert()` einen Kommentar über den fehlgeschlagenen Aufruf auf `stderr` und ruft `abort()` auf. Die Meldung enthält den Argumenttext, den Quelldateinamen (_ _FILE_ _) und die Zeilenummer (_ _LINE_ _).

Hinweis `assert`-Aufrufe werden nicht ausgeführt, wenn `NDEBUG` definiert wird. Dazu gibt es folgende Möglichkeiten:

- beim Compileraufruf durch eine Präprozessor-Option (siehe C- und C++-Benutzerhandbücher)
- im Quellcode durch die Präprozessoranweisung `#define NDEBUG` vor der Anweisung `#include <assert.h>`

Siehe auch `abort()`, _ _FILE_ _, _ _LINE_ _, `stderr()`, `assert.h`.

atan - Arcustangens berechnen

Syntax #include <math.h>

double atan(double *x*);

Beschreibung

`atan()` ist die Umkehrfunktion zu `tan()` und berechnet zur Gleitpunktzahl *x* den entsprechenden Winkel im Bogenmaß.

Returnwert arcustangens(*x*)

bei Erfolg. Es wird eine Gleitpunktzahl vom Typ `double` aus dem Intervall $[-\pi/2, +\pi/2]$ zurückgegeben.

Siehe auch `acos()`, `asin()`, `atan2()`, `cos()`, `sin()`, `tan()`, `math.h`.

atan2 - Arcustangens von x/y berechnen

Syntax #include <math.h>

double atan2(double *x*, double *y*);

Beschreibung

`atan2()` berechnet den Arcustangens von *x*/*y*. Die Vorzeichen der beiden Argumente bestimmen den Ergebnisquadranten.

x ist der Dividend des Ausdrucks, dessen Arcustangens berechnet werden soll.

y ist der Divisor des Ausdrucks, dessen Arcustangens berechnet werden soll.

Returnwert arcustangens(*x*/*y*)
wenn beide Argumente ungleich 0.0 sind.
Es wird eine Gleitpunktzahl vom Typ `double` aus dem Intervall [-π/2, +π/2] zurückgegeben.

-π/2 bzw. +π/2
wenn der Divisor 0.0 ist, abhängig vom Vorzeichen des Dividenden.

0 wenn der Dividend 0.0 ist.

π/2 wenn beide Argumente gleich 0.0 sind. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `atan2()` schlägt fehl, wenn gilt:

`EDOM` Beide Argumente sind gleich 0.0.

Hinweis Um einen Fehler sicher abzufangen, sollte `errno` vor Aufruf von `atan2()` auf 0 gesetzt werden. Ist nach der Ausführung `errno` ≠ 0, so ist ein Fehler aufgetreten.

Siehe auch `acos()`, `asin()`, `atan()`, `cos()`, `sin()`, `tan()`, `math.h`.

atanh - inverse Hyperbel-Tangensfunktion

Syntax #include <math.h>

double atanh (double *x*);

Beschreibung

Siehe `acosh()`.

atexit - Prozessendefunktion registrieren

Syntax

```
#include <stdlib.h>

int atexit(void (*func) (void));
```

Beschreibung

Mit `atexit()` wird eine Funktion *func*() registriert, die bei normaler Prozessbeendigung ohne Argumente aufgerufen werden soll. Registrierte Funktionen werden in der umgekehrten Reihenfolge ihrer Registrierung aufgerufen. Wird eine Funktion mehrmals registriert, wird sie auch mehrmals aufgerufen.

Die mit `atexit()` registrierten Funktionen werden nur aufgerufen, wenn der Prozess auf eine der folgenden Arten "normal" beendet wird:

– expliziter Aufruf von `exit()`

– Beendigung der `main`-Funktion ohne expliziten `exit`-Aufruf

BS2000

– Prozessbeendigung durch das C-Laufzeitsystem mit `exit(-1)`, das heißt bei Auftritt eines `raise`-Signals (nicht `SIGABRT`), das entweder nicht oder durch die voreingestellte Signalbehandlung über `SIG_DFL` behandelt wird (siehe `signal()`). ❑

Es können bis zu 40 Funktionen registriert werden.

Nach dem erfolgreichen Aufruf einer `exec`-Funktion sind die vorher mit `atexit()` registrierten Funktionen nicht mehr registriert.

Returnwert

0	bei erfolgreicher Registrierung der Funktion.
≠ 0	bei Fehler.

Hinweis

Damit alle registrierten Funktionen aufgerufen werden, muss der Anwender sicherstellen, dass registrierte Funktionen zurückkehren.

Die Funktion `sysconf()` liefert den Wert von `ATEXIT_MAX` zurück, der angibt, wie viele Funktionen insgesamt registriert werden können. Es gibt jedoch keine Möglichkeit (außer duch Mitzählen) herauszufinden, wie viele Funktionen bereits registriert wurden.

Siehe auch `bs2exit()`, `exit()`, `signal()`, `stdlib.h`.

atof - Zeichenkette in Gleitpunktzahl umwandeln

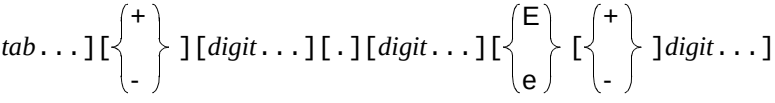
Syntax

```
#include <stdlib.h>

double atof(const char *str);
```

Beschreibung

`atof()` wandelt eine EBCDIC-Zeichenkette, auf die *str* zeigt, in eine Gleitpunktzahl vom Typ `double` um. Die umzuwandelnde Zeichenkette kann wie folgt aufgebaut sein:



Für *tab* sind alle Zwischenraumzeichen zulässig (siehe Definition bei `isspace()`).

Die Funktion `atof(str)` unterscheidet sich von `strtod(str,(char**)NULL)` nur durch die Fehlerbehandlung.

Returnwert

Gleitpunktzahl vom Typ `double`
für Zeichenketten, die eine wie oben beschriebene Struktur haben und einen Zahlenwert darstellen, der im zulässigen Gleitpunktbereich liegt.

Erweiterung
0 für Zeichenketten, die nicht der oben beschriebenen Syntax entsprechen.

`HUGE_VAL` für Zeichenketten, deren Zahlenwert außerhalb des zulässigen Gleitpunktbereichs liegt. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler

`atof()` schlägt fehl, wenn gilt:

`ERANGE` Der Returnwert verursacht einen Über- oder Unterlauf. ❑

Hinweis

`atof()` ist vollständig enthalten in `strtod()`. Die Funktion wird jedoch weiterhin angeboten, da sie in vielen existierenden Anwendungen eingesetzt ist.
Das Dezimalzeichen in der umzuwandelnden Zeichenkette wird durch die Lokalität (Kategorie `LC_NUMERIC`) beeinflusst. Voreingestellt ist der Punkt.

`atof()` erkennt auch Zeichenketten, die mit Ziffern beginnen, dann aber mit beliebigen Zeichen enden. `atof()` schneidet den Ziffernteil ab, wandelt ihn gemäß obiger Beschreibung um und ignoriert den Rest.

Siehe auch

`atoi()`, `atol()`, `strtod()`, `strtol()`, `strtoul()`, `stdlib.h`.

atoi - Zeichenkette in ganze Zahl umwandeln

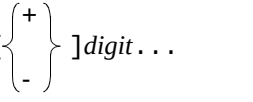
Syntax

```
#include <stdlib.h>

int atoi(const char *str);
```

Beschreibung

`atoi()` wandelt die EBCDIC-Zeichenkette, auf die *str* zeigt, in eine ganze Zahl um. Die umzuwandelnde Zeichenkette kann wie folgt aufgebaut sein:



Für *tab* sind alle Zwischenraumzeichen zulässig (siehe `isspace()`).

Die Funktion `atoi(str)` unterscheidet sich von `strtol(str,(char**)NULL)` nur durch die Fehlerbehandlung.

Returnwert

Ganzzahliger Wert vom Typ `int`
: für Zeichenketten, die eine wie oben beschriebene Struktur haben und einen Zahlenwert darstellen, der im zulässigen Integerbereich liegt.

0
: für Zeichenketten, die nicht der oben beschriebenen Syntax entsprechen.

`INT_MAX` bzw. `INT_MIN`
: bei Überlauf, abhängig vom Vorzeichen.

Hinweis

`atoi()` ist vollständig enthalten in `strtol()`. Die Funktion wird jedoch weiterhin angeboten, da sie in vielen existierenden Anwendungen eingesetzt ist.

`atoi()` erkennt auch Zeichenketten, die mit Ziffern beginnen, dann aber mit beliebigen Zeichen enden. `atoi()` schneidet den Ziffernteil ab, wandelt ihn gemäß obiger Beschreibung um und ignoriert den Rest.

Siehe auch `atof()`, `atol()`, `strtod()`, `strtol()`, `strtoul()`, `stdlib.h`.

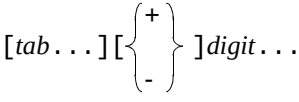
atol - Zeichenkette in ganze Zahl (long) umwandeln

Syntax #include <stdlib.h>

long int atol(const char *_str_);

Beschreibung

`atol()` wandelt eine EBCDIC-Zeichenkette, auf die *str* zeigt, in eine ganze Zahl vom Typ `long` um. Die umzuwandelnde Zeichenkette kann wie folgt aufgebaut sein:



Für *tab* sind alle Steuerzeichen für "Zwischenraum" zulässig (siehe Definition bei `isspace()`).

Die Funktion `atol(str)` unterscheidet sich von `strtol(str,(char**)NULL,10)` nur durch die Fehlerbehandlung.

Returnwert Ganzzahliger Wert vom Typ `long`
: für Zeichenketten, die eine wie oben beschriebene Struktur haben und einen Zahlenwert darstellen.

0
: für Zeichenketten, die nicht der oben beschriebenen Syntax entsprechen.

`LONG_MAX` bzw. `LONG_MIN`
: bei Überlauf, abhängig vom Vorzeichen.

Hinweis `atol()` ist vollständig enthalten in `strtol()`. Die Funktion wird jedoch weiterhin angeboten, da sie in vielen existierenden Anwendungen eingesetzt ist.

`atol()` erkennt auch Zeichenketten, die mit Ziffern beginnen, dann aber mit beliebigen Zeichen enden. `atol()` schneidet den Ziffernteil ab, wandelt ihn gemäß obiger Beschreibung um und ignoriert den Rest.

Siehe auch `atof()`, `atoi()`, `strtod()`, `strtol()`, `strtoul()`, `stdlib.h`.

atoll - Zeichenkette in ganze Zahl umwandeln (long long int)

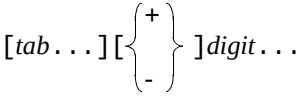
Syntax

```
#include <stdlib.h>

long long int atoll(const char *s);
```

Beschreibung

`atoll()` wandelt eine EBCDIC-Zeichenkette, auf die *s* zeigt, in eine ganze Zahl vom Typ `long long int` um. Die umzuwandelnde Zeichenkette kann wie folgt aufgebaut sein:



Für *tab* sind alle Steuerzeichen für „Zwischenraum“ zulässig (siehe Definition bei `isspace()`).

Returnwert

Ganzzahliger Wert vom Typ `long long int`
: für Zeichenketten, die eine wie oben beschriebene Struktur haben und einen Zahlenwert darstellen.

0
: für Zeichenketten, die nicht der oben beschriebenen Syntax entsprechen.

`LLONG_MAX` bzw. `LLONG_MIN`
: bei Überlauf, abhängig vom Vorzeichen.

Hinweis

`atoll()` erkennt auch Zeichenketten, die mit Ziffern beginnen, dann aber mit beliebigen Zeichen enden. `atoll()` schneidet den Ziffernteil ab, wandelt ihn gemäß obiger Beschreibung um und ignoriert den Rest.

Ist *zg* ein Nullzeiger und *base* gleich 10, unterscheidet sich `atoll()` von der Funktion `strtoll()` nur durch die Fehlerbehandlung.
`atoll(s)` entspricht `strtoll(s, (char **)NULL, 10)`.

Siehe auch

`atof()`, `atoi()`, `atol()`, `strtod()`, `strtol()`, `stroll()`, `strtoul()`, `stroull()`

basename - letztes Element eines Pfadnamens zurückgeben

Syntax #include <libgen.h>

char *basename (char **path*);

Beschreibung

Wenn man `basename()` einen Zeiger auf eine mit Null beendete Zeichenkette übergibt, die einen Pfadnamen enthält, gibt `basename()` einen Zeiger auf das letzte Element von *path* zurück. Abschließende `/`-Zeichen (Schrägstriche) werden gelöscht.

Wenn die übergebene Zeichenkette nur aus dem Zeichen '/' besteht, wird ein Zeiger auf die Zeichenkette '/' zurückgegeben.

Wenn *path* oder **path* null ist, wird ein Zeiger auf die Zeichenkette '`.`' zurückgegeben.

`basename()` ist nicht reentrant.

Returnwert Zeiger auf die letzte Komponente von *path*.

Beispiel

```
Eingabezeichenkette Ausgabezeiger

/usr/lib            lib
/usr/               usr
/                   /
```

Hinweis `basename()` arbeitet auf der übergebenen Zeichenkette. Die Zeichenkette wird ggf. verändert, indem abschließende Schrägstriche ('/') durch '\0' überschrieben werden.

Siehe auch `dirname()`, `libgen.h`.

bcmp - Speicherbereiche vergleichen

Syntax #include <strings.h>

int bcmp(const void *s1, const void *s2, size_t n);

Beschreibung

`bcmp()` vergleicht die ersten *n* Byte ab der Adresse im Speicher, auf die *s1* zeigt, mit dem über *s2* adressierten Speicherbereich. Es wird vorausgesetzt, dass beide Bereiche im Speicher mindestens *n* Byte lang sind.

Returnwert

0	Alle *n* Byte sind gleich, oder *n*=0.
≠ 0	Die beiden Speicherbereiche unterscheiden sich.

Hinweis Portable Anwendungen sollten statt `bcmp()` die Funktion `memcmp()` verwenden.

Siehe auch `memcmp()`, `strings.h`.

bcopy - Speicherbereich kopieren

Syntax #include <strings.h>

void bcopy(const void *s1, const void *s2, size_t n);

Beschreibung

`bcopy()` kopiert *n* Byte ab der Adresse im Speicher, auf die *s1* zeigt, in den über *s2* adressierten Speicherbereich. Sich überlagernde Bereiche werden korrekt bearbeitet.

Hinweis Portable Anwendungen sollten statt `bcopy()` die Funktion `memmove()` verwenden.

Die beiden folgenden Funktionsaufrufe sind nahezu äquivalent (Achtung: die Reihenfolge der Argumente *s1* und *s2* ist vertauscht!):

`bcopy(`*s1*`,` *s2*`,` *n*`)` ≅ `memmove(`*s2*`,` *s1*`,` *n*`)`

Siehe auch `memmove()`, `strings.h`.

brk, sbrk - Größe des Datensegments verändern

Syntax

```
#include <unistd.h>

int brk(void *addr);
void *sbrk(int incr);
```

Beschreibung

`brk()` und `sbrk()` werden zum dynamischen Ändern des Speicherplatzes verwendet, der dem Datensegment des aufrufenden Prozesses zugewiesen ist (vgl. `exec`). Die Änderung wird durch Rücksetzen des Speichergrenzwerts (’break value’) des Prozesses und Zuweisen eines entsprechenden Bereichs vorgenommen. Der Speichergrenzwert (’break value’) ist die erste nicht belegte Adresse oberhalb des Datensegments. Der Umfang des zugewiesenen Speicherplatzes erhöht sich mit der Vergrößerung des Speichergrenzwerts. Neu zugewiesener Speicherplatz wird auf null gesetzt. Wenn jedoch derselbe Speicherplatz demselben Prozess wieder zugewiesen wird, ist sein Inhalt undefiniert.

`brk()` setzt den Grenzwert auf *addr* und ändert den zugewiesenen Platz entsprechend.

`sbrk()` fügt *incr* Bytes zum Grenzwert hinzu und ändert den zugewiesenen Platz entsprechend. *incr* kann negativ sein. In diesem Fall wird der Umfang des zugewiesenen Speicherplatzes verringert. Der aktuelle Speichergrenzwert wird von `sbrk(0)` zurückgegeben.

Wenn eine Anwendung zusätzlich weitere Funktionen zur Speicherbereichsverwaltung einsetzt, wie z.B. `malloc()`, `mmap()` oder `free()`, ist das Verhalten von `brk()` und `sbrk()` undefiniert. Alle anderen Funktionen können diese weiteren Speicherverwaltungsfunktionen problemlos verwenden.

`brk()` und `sbrk()` sind nicht reentrant.

Returnwert

`brk()`:

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

`sbrk()`:

vorheriger Speichergrenzwert	bei Erfolg.
`(void*)-1`	bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler

`brk()` und `sbrk()` sind erfolglos und ändern den zugewiesenen Speicherplatz nicht, wenn gilt:

`ENOMEM`	Eine derartige Änderung würde dazu führen, dass mehr Speicherplatz zugewiesen wird, als durch die systembedingte maximale Prozessgröße zulässig ist (siehe `ulimit()`).

Hinweis Die Funktionen `brk()` und `sbrk()` wurden in speziellen Fällen benötigt, wo keine andere Speicherverwaltungsfunktion dieselben Möglichkeiten geboten hätte. Jetzt wird jedoch die Funktion `mmap()` empfohlen, da sie problemlos gleichzeitig mit allen anderen Speicherverwaltungsfunktionen eingesetzt werden kann.

Der Zeiger, der von `sbrk()` zurückgegeben wird, ist nicht für jede weitere Verwendung passend ausgerichtet.

Siehe auch `exec()`, `malloc()`, `mmap()`, `ulimit()`, `unistd.h`.

bs2exit - Programm mit MONJV beenden *(BS2000)*

Syntax #include <stdlib.h>

void bs2exit(int *status*, const char **monjv_rcode*);

Beschreibung

`bs2exit()` beendet das aufrufende Programm. Vorher werden alle vom Programm geöffneten Dateien geschlossen und folgende Meldungen auf `stderr` ausgegeben:

- "`CCM0998 used CPU-time` *t* `seconds`", falls in der `RUNTIME`-Option `CPU-TIME=YES` gesetzt ist.
- "`CCM0999 exit` *status*", falls *status* ≠ `EXIT_SUCCESS` (Wert 0) ist.
- "`CCM0999 exit FAILURE`", falls *status* = `EXIT_FAILURE` (Wert 9990888) ist.
- "`EXC0732 ABNORMAL PROGRAMM TERMINATION. ERROR CODE NRT0101`“

Die Zustandsanzeige der Monitor-Jobvariablen (1. - 3. Byte) wird entsprechend dem Argument *status* wie bei der Funktion `exit()` auf den Wert "`$A` " gesetzt, falls *status* = `EXIT_FAILURE`. Bei allen anderen Werten für *status* steht „`$T`“ in der Monitor-Jobvariablen.

Zusätzlich lässt sich mit *monjv_rcode* die Rückkehranzeige von `MONJV` (4. - 7. Byte) versorgen. Für *monjv_rcode* kann ein Zeiger auf eine 4 Byte lange Information (Rückkehranzeige) angegeben werden, die bei Programmbeendigung in `MONJV` aufgenommen wird.

Inhalt und Auswertung des Arguments *status* sind identisch mit `exit()`.

Hinweis Bei der Programmbeendigung mit `bs2exit()` werden die mit `atexit()` registrierten Beendigungsroutinen nicht aufgerufen (siehe `exit()`).

Um Monitor-Jobvariablen versorgen und abfragen zu können, muss das C-Programm mit folgendem Kommando gestartet werden:

`/START-PROG` *programm*`,MONJV=`*monjvname*

Der Inhalt der Jobvariablen lässt sich dann z.B. mit folgendem Kommando abfragen:

`/SHOW-JV JV-NAME(`*monjvname*`)`

Weitere Informationen zur Ablaufüberwachung mit `MONJV` finden Sie im Handbuch "Jobvariablen".

Siehe auch `exit()`, `_exit()`.

bs2fstat - BS2000-Dateinamen aus Katalog ermitteln *(BS2000)*

Syntax

#include <stdlib.h>

int bs2fstat(const char **pattern*, void (**function*)(const char **filename*, int *len*));

Beschreibung

`bs2fstat` liefert den vollqualifizierten Dateinamen (`:catid:$userid.dateiname`) von Dateien, die das Auswahlkriterium *pattern* erfüllen, sowie die Länge des jeweiligen Dateinamens inklusive des abschließenden Nullbytes (\0).

Für jede gefundene Datei ruft `bs2fstat` eine vom Benutzer bereitzustellende Funktion *function* auf und übergibt an diese als aktuelle Argumente den jeweiligen Dateinamen *filename* (Zeichenkette char *) und die Namenslänge *len* (ganze Zahl).

`const char` ***pattern** ist eine Zeichenkette, die das Auswahlkriterium für einen oder mehrere Dateinamen angibt.

pattern ist ein voll- oder teilqualifizierter Dateiname mit Wildcard-Syntax.

Außerdem können aus Kompatibilitätsgründen weitere Parameter angegeben werden, die die Auswahl der Dateien beeinflussen, z.B:

Datei- und Katalogeigenschaften (FCBTYPE, SHARE etc.)

Erstellungs- und Zugriffsdatum (CRDATE, EXDATE etc.)

Diese Parameter müssen in der Syntax des ISP-Kommandos FSTAT angegeben werden.

Beispielsweise liefert das Muster "*,`crdate=today`" die Namen aller Dateien, die am jeweils heutigen Tag erstellt bzw. verändert wurden.

`void` (***function**)(`const char` ***filename**, int *len*) ist eine vom Benutzer bereitzustellende Funktion mit den Parametern *filename* (Dateiname) und *len* (Namenslänge). Diese Parameter werden von `bs2fstat` bei jedem Funktionsaufruf mit aktuellen Werten versorgt. Die Funktionsaufrufe erfolgen durch `bs2fstat` automatisch (in einer `while`-Schleife).

Returnwert

0 bei Erfolg.

DMS-Fehlermeldungscode
bei Fehler.

Hinweis

Das Kennzeichen für DMS-Fehlermeldungen kann nur außerhalb der benutzereigenen Funktion *function* abgefragt werden, da bei erfolgloser Suche die Funktion nicht aufgerufen wird.

Siehe auch

`system()`, `stdio.h`.

bs2system - BS2000-Kommando ausführen *(Erweiterung)*

Syntax

```
#include <stdlib.h>

int bs2system(const char *command);
```

Beschreibung

`bs2system()` führt das BS2000-Kommando aus, das in der Zeichenkette *command* steht.

Returnwert

0	wenn das BS2000-Kommando erfolgreich ausgeführt wurde (Returnwert des entsprechenden BS2000-Kommandos: 0).
-1	wenn das BS2000-Kommando nicht erfolgreich ausgeführt wurde (Returnwert des BS2000-Kommandos: Fehlercode ≠ 0).
undefiniert	wenn nach dem BS2000-Kommando nicht in das Programm zurückverzweigt wird (siehe auch Hinweis).

Hinweis

`bs2system()` übergibt die Zeichenkette *command* unverändert dem BS2000-Kommandoprozessor MCLP als Eingabe (siehe auch Handbuch „BS2000/OSD-BC Makroaufrufe an den Ablaufteil“). Es erfolgt keine Umsetzung in Großbuchstaben. Deshalb muss das BS2000-Kommando in Großbuchstaben angegeben werden; es kann maximal 2048 Zeichen lang sein und muss nicht mit dem System-Schrägstrich (/) angegeben werden.

Nach einigen BS2000-Kommandos (START-PROG, LOAD-PROG, CALL-PROCEDURE, DO, HELP-SDF) wird nicht in das aufrufende Programm zurückverzweigt. Wenn ein Programm vorzeitige Programmbeendigungen zulässt, sollte es vor dem `bs2system`-Aufruf die Puffer leeren (`fflush()`) bzw. die Dateien schließen.

Siehe auch `system()`, `stdlib.h`.

bsd_signal - vereinfachte Signalbehandlung

Syntax

```
#include <signal.h>

void (*bsd_signal(int sig, void (*func)(int))) (int);
```

Beschreibung

Die Funktion `bsd_signal()` stellt eine teilkompatible Schnittstelle für Programme bereit, die für historische Systemschnittstellen geschrieben wurden (siehe unten „Hinweis“).

Der Funktionsaufruf `bsd_signal(`*sig*`,` *func*`)` wirkt, als ob er folgendermaßen implementiert wäre:

```
void (*bsd_signal(int sig, void (*func)(int)))(int)
{
     struct sigaction act, oact;

     act.sa_handler = func;
     act.sa_flags = SA_RESTART;
     sigemptyset(&act.sa_mask);
     sigaddset(&act.sa_mask, sig);
     if (sigaction(sig, &act, &oact) == -1)
          return(SIG_ERR);
     return(oact.sa_handler);
}
```

Die Ereignisbehandlungsfunktion sollte folgendermaßen deklariert werden:

```
void handler(int sig);
```

Dabei steht *sig* für die Signalnummer. Das Verhalten ist nicht definiert, wenn *func* eine Funktion ist, die mehr als ein Argument oder ein Argument eines anderen Typs hat.

Returnwert

Die vorausgegangene Aktion für *sig*	bei Erfolg.
`SIG_ERR`	bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler

Siehe `sigaction()`.

Hinweis

Diese Funktion ist ein direkter Ersatz für die BSD-Funktion `signal()` für einfache Anwendungen, für die eine Signalbehandlungsfunktion mit einem Argument installiert wird. Falls eine BSD-Signalbehandlungsfunktion installiert wird, die mehr als ein Argument erwartet, muss die Anwendung dahingehend geändert werden, dass sie `sigaction()` verwendet. Die Funktion `bsd_signal()` unterscheidet sich insofern von `signal()`, als das Flag `SA_RESTART` gesetzt ist und `SA_RESETHAND` gelöscht ist, wenn `bsd_signal()` verwendet wird. Der Status dieser Flags ist für `signal()` nicht angegeben.

Siehe auch

`sigaction()`, `sigaddset()`, `sigemptyset()`, `signal()`, `signal.h`.

bsearch - sortierten Vektor binär durchsuchen

Syntax #include <stdlib.h>

void *bsearch(const void **key*, const void **base*, size_t *nel*,
size_t *width*, int (**compar*) (const void *, const void *));

Beschreibung

`bsearch()` ist eine binäre Suchfunktion. `bsearch()` durchsucht *nel* Elemente eines Vektors *base* nach dem Wert im Datenelement *key*. Jedes Vektorelement ist *width* Bytes lang.

compar() ist eine vom Benutzer bereitzustellende Vergleichsfunktion, die von `bsearch()` jeweils mit zwei Argumenten aufgerufen wird, einem Zeiger auf *key* und einem Zeiger auf ein Vektorelement.

compar() muss eine ganze Zahl liefern, die kleiner, gleich oder größer als null ist, je nachdem, ob das erste Argument kleiner, gleich oder größer als das zweite Argument ist. Der Vektor muss die Elemente in der folgenden Reihenfolge enthalten: erst alle Elemente, die kleiner als *key* sind, dann alle Elemente, die gleich und schließlich alle Elemente, die größer als *key* sind.

Returnwert Zeiger auf das gesuchte Element
bei Erfolg. Wenn das gesuchte Element mehrmals vorhanden ist, ist nicht festgelegt, auf welches Element der Zeiger zeigt.

Nullzeiger wenn kein Element gefunden wurde.

Hinweis Die Zeiger auf *key* und das Element am Anfang des Vektors sollten vom Typ „Zeiger-auf-*Element*“ sein.

Die Vergleichsfunktion muss nicht jedes Byte vergleichen, deshalb können die Elemente zusätzlich zu den Vergleichswerten beliebige Daten enthalten.

In der Praxis sind die Elemente des Vektors meist entsprechend der Vergleichsfunktion sortiert.

Wenn die Anzahl von Elementen im Vektor kleiner als die für den Vektor reservierte Größe ist, sollte *nel* die kleinere Zahl sein.

BS2000
Wird für die Sortierung des Vektors z.B. die Funktion `qsort()` verwendet, ist es sinnvoll, dieselbe Vergleichsfunktion *compar*() zu verwenden, die von `bsearch()` benutzt wird. Die aktuellen Argumente von `qsort()` sind dann Zeiger auf zwei zu vergleichende Vektorelemente. ❑

Siehe auch `hsearch()`, `lsearch()`, `qsort()`, `tsearch()`, `stdlib.h`.

btowc - (ein-byte) Multibyte-Zeichen in Langzeichen umwandeln

Syntax
```
#include <stdio.h>
#include <wchar.h>

wint_t btowc(int c);
```

Beschreibung

`btowc()` konvertiert das Multibyte-Zeichen *c*, das aus einem Byte besteht und sich im „initial shift“-Zustand befinden muss, in ein Langzeichen.

Returnwert

Langzeichen	bei Erfolg.
`WEOF`	falls *c* den Wert EOF enthält oder `(unsigned char)c` kein gültiges (1-Byte) Multibyte-Zeichen im „initial shift“-Zustand darstellt.

Hinweis

In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Multibyte-Zeichen unterstützt.
Der Shift-Zustand des Multibyte-Zeichens wird ignoriert.

Siehe auch `mblen()`, `mbtowc()`, `wcstombs()`, `wctomb()`

bzero - Speicher mit X‘00‘ initialisieren

Syntax #include <strings.h>

void bzero(void *s, size_t n);

Beschreibung

`bzero()` überschreibt *n* Bytes ab der Adresse, auf die *s* zeigt, mit X‘00‘.

Hinweis Portable Anwendungen sollten statt `bzero()` die Funktion `memset()` verwenden.

Siehe auch `memset()`, `strings.h`.

cabs - Absolutwert einer komplexen Zahl berechnen *(BS2000)*

Syntax #include <math.h>

double cabs(_ _complex *z*);

Beschreibung

`cabs()` berechnet den Absolutwert der komplexen Zahl *z*.

`struct (_ _complex z)` ist eine komplexe Zahl *z* mit Realteil *x* und Imaginärteil *y*.

`_ _complex` ist ein in `math.h` vordefinierter Typ:

```
#typdef struct{double x, y;} _ _complex
```

Returnwert Absolutbetrag der komplexen Zahl *z* bei Erfolg.

Programmabbruch bei Überlauf (Signal `SIGFPE`).

Siehe auch `abs()`, `fabs()`, `labs()`, `sqrt()`, `math.h`.

calloc - Speicherbereich zuweisen

Syntax #include <stdlib.h>

void *calloc(size_t *nelem*, size_t *elsize*);

Beschreibung

`calloc()` beschafft zur Ausführungszeit ungenutzten Speicherplatz für einen Vektor mit *nelem* Elementen, wobei jedes Element *elsize* Byte beansprucht. `calloc()` initialisiert jedes Element des neuen Vektors mit binären Nullen.

`calloc()` ist Teil eines C-spezifischen Speicherverwaltungspaketes, das angeforderte und wieder freigegebene Speicherbereiche intern verwaltet. Neue Anforderungen werden zuerst aus bereits verwalteten Bereichen zu erfüllen versucht, dann erst vom Betriebssystem.

nelem ist ein ganzzahliger Wert, der die Anzahl der Vektorelemente angibt.
elsize ist ein ganzzahliger Wert, der die Größe eines Vektorelementes angibt.

Wenn Speicherbereiche durch aufeinander folgende Aufrufe von `calloc()` zugewiesen wurden, so ist die Anordnung dieser Bereiche im Speicher undefiniert. Der Zeiger, der bei erfolgreicher Allokierung zurückgegeben wird, ist auf Doppelwortgrenze ausgerichtet, so dass er einem Zeiger auf jeden Typ von Objekt zugewiesen werden kann. Nach der Zuweisung kann auf das Objekt oder auf einen Vektor solcher Objekte in dem neu zugewiesenen Speicherbereich zugegriffen werden (bis der Bereich explizit freigegeben oder erneut zugewiesen wird).

Returnwert Zeiger auf den neuen Speicherplatz
falls *nelem* und *elsize* ungleich 0 sind und genügend Speicherplatz vorhanden ist.

Nullzeiger wenn der Speicherplatz für die Anforderung nicht ausreicht. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `calloc()` schlägt fehl, wenn gilt:

`ENOMEM` Es ist nicht genügend Speicherplatz verfügbar.

Hinweis Der neue Datenbereich beginnt auf Doppelwortgrenze.
Um sicherzugehen, dass Sie die richtige Größe für ein Vektorelement anfordern, sollten Sie für die Berechnung von *elsize* den Operator `sizeof` verwenden.
Wird die Länge des zur Verfügung gestellten Speicherbereiches beim Schreiben überschritten, führt dies zu schwer wiegenden Fehlern im Arbeitsspeicher.

`calloc()` ist unterbrechungssicher, d.h. die Funktion kann in Signalbehandlungs- und Contingency-Routinen verwendet werden.

Siehe auch `free()`, `malloc()`, `realloc()`, `stdlib.h`.

catclose - Meldungskatalog schließen

Syntax

```
#include <nl_types.h>

int catclose(nl_catd catd);
```

Beschreibung

`catclose()` schließt den Meldungskatalog, der durch den Meldungskatalog-Deskriptor *catd* identifiziert wird. Wenn ein Dateideskriptor verwendet wird, um den Typ `nl_catd` zu definieren, wird auch dieser Dateideskriptor geschlossen.

Returnwert

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

Fehler

`catclose()` schlägt fehl, wenn gilt:

`EBADF`	Der Meldungskatalog-Deskriptor ist ungültig.
`EINTR`	`catclose()` wurde durch ein Signal unterbrochen.

Siehe auch `catgets()`, `catopen()`, `nl_types.h`, Abschnitt „Lokalität“ auf Seite 52.

catgets - Meldung lesen

Syntax #include <nl_types.h>

char *catgets(nl_catd *catd*, int *set_id*, int *msg_id*, const char **s*);

Beschreibung

`catgets()` versucht, die Meldung *msg_id* in der Menge *set_id* aus dem Meldungskatalog zu lesen, der durch *catd* identifiziert wird.

catd ist ein Meldungskatalog-Deskriptor, der durch einen vorausgegangenen Aufruf von `catopen()` erzeugt wurde.

s zeigt auf eine voreingestellte Meldungszeichenkette, die geliefert wird, wenn `catgets()` die angegebene Meldung nicht lesen kann.

Returnwert Zeiger auf einen internen Pufferbereich, der die mit X‘00‘ abgeschlossene Meldung enthält bei Erfolg.

s bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `catgets()` schlägt fehl, wenn gilt:

`EBADF` Der Meldungskatalog-Deskriptor ist zum Lesen ungültig.

`EINTR` Die Leseoperation wurde durch ein Signal unterbrochen, und keine Daten wurden übertragen.

Siehe auch `catopen()`, `nl_types.h`, Abschnitt „Lokalität“ auf Seite 52.

catopen - Meldungskatalog öffnen

Syntax #include <nl_types.h>

nl_catd catopen(const char **name*, int *oflag*);

Beschreibung

`catopen()` öffnet einen Meldungskatalog und liefert einen Meldungskatalog-Deskriptor zurück.

name gibt den Namen des zu öffnenden Meldungskatalogs an. Wenn *name* einen Schrägstrich / enthält, wird *name* als absoluter Pfadname interpretiert. Andernfalls wird die Umgebungsvariable `NLSPATH` ausgewertet, wobei *name* für `%N` eingesetzt wird (siehe auch Abschnitt „Lokalität“ auf Seite 52).

Wenn die Umgebungsvariable `NLSPATH` nicht existiert oder der Meldungskatalog unter irgendeiner in `NLSPATH` definierten Pfadkomponente nicht geöffnet werden kann, wird der voreingestellte Pfad verwendet (siehe `nl_types.h`).

Wenn *oflag* gleich `NL_CAT_LOCALE` ist, wird diese Voreinstellung durch die Kategorie `LC_MESSAGES` bestimmt.

Wenn *oflag* 0 ist, wird nur die Umgebungsvariable `LANG` ausgewertet unabhängig vom Inhalt der Kategorie `LC_MESSAGES` (siehe auch Abschnitt „Umgebungsvariablen“ auf Seite 71).

Ein Meldungskatalog-Deskriptor bleibt in einem Prozess so lange gültig, bis ihn der Prozess oder ein erfolgreicher Aufruf einer `exec`-Funktion schließt. Eine Änderung in der Kategorie `LC_MESSAGES` kann existierende offene Kataloge ungültig machen.

Wenn ein Dateideskriptor benutzt wird, um Meldungskatalog-Deskriptoren zu definieren, wird das Bit `FD_CLOEXEC` gesetzt (siehe auch `fcntl.h`).

Returnwert Meldungskatalog-Deskriptor
bei Erfolg. Dieser kann nun von `catgets()` und `catclose()` verwendet werden.

`(nl_catd)` -1 bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `catopen()` schlägt fehl, wenn gilt:

`EACCES` Eine Komponente des Meldungskatalog-Pfadpräfixes darf nicht durchsucht werden,
oder der Meldungskatalog darf nicht gelesen werden.

`EMFILE` Der Prozess verwendet mehr als `{OPEN_MAX}` Dateideskriptoren gleichzeitig.

ENAMETOOLONG
: Die Länge des Meldungskatalog-Pfadnamens überschreitet {PATH_MAX}, oder eine Komponente des Pfadnamens ist größer als {NAME_MAX} oder die Auflösung eines symbolischen Verweises erzeugt ein Zwischenergebnis, das länger ist als {PATH_MAX}.

ENFILE
: Zu viele Dateinamen sind aktuell im System offen.

ENOENT
: Der Meldungskatalog existiert nicht, oder *name* weist auf eine leere Zeichenkette.

ENOMEM
: Es ist nicht genügend Speicherplatz verfügbar.

ENOTDIR
: Eine Komponente des Meldungskatalog-Pfadnamens ist kein Verzeichnis.

Hinweis `catopen()` verwendet `malloc()`, um Speicherplatz für die internen Pufferbereiche zu reservieren. `catopen()` schlägt fehl, wenn nicht genügend Speicherplatz für die Unterbringung dieser Puffer verfügbar ist.

Portable Anwendungen müssen berücksichtigen, dass Meldungskatalog-Deskriptoren nach dem Aufruf einer `exec`-Funktion nicht mehr gültig sind.

Jede Anwendung muss den zugehörigen Meldungskatalog in einem der durch `DEF_NLSPATH` voreingestellten Dateiverzeichnisse so ablegen, dass er bei der Ersetzung von `%N` durch *name* gefunden wird (siehe auch `nl_types.h`).

Siehe auch `catclose()`, `catgets()`, `fcntl.h`, `nl_types.h`, Abschnitt „Lokalität“ auf Seite 52 und Abschnitt „Umgebungsvariablen“ auf Seite 71.

cbrt - Kubikwurzel

Syntax #include <math.h>

double cbrt (double *x*);

Beschreibung
`cbrt()` gibt die Kubikwurzel von *x* zurück.

Returnwert Kubikwurzel von *x*
bei Erfolg.

Siehe auch `math.h`.

cdisco - Contingency-Routine abmelden *(BS2000)*

Syntax #include <cont.h>

void cdisco(struct enacop **enacopar*);

Beschreibung

`cdisco()` meldet eine mit `cenaco()` definierte Contingency-Routine (TU bzw. P1) ab. Ausführliche Informationen zu Contingency-Routinen finden Sie im Abschnitt „Contingency- und STXIT-Routinen“ auf Seite 121 und im Handbuch „BS2000/OSD-BC Makroaufrufe an den Ablaufteil“.

Die Struktur `enacop` ist wie folgt in `cont.h` definiert:

```
struct enacop
{
  char resrv1 [7];        /* reserved for int. use   */
  char coname [54];       /* name of cont. routine   */
  char resrv2 [15];       /* reserved for int. use   */
  char level;             /* priority of cont.rout.  */
  int  (*econt)();        /* start adr of cont.rout. */
  int  comess;            /* contingency message     */
  char coidret [4];       /* contingency identifier  */
  errcod secind;          /* secondary indicator     */
  char resrv3 [2];        /* reserved for int. use   */
  errcod rcode1;          /* return code             */
};

#define errcod     char
#define _norm      0      /* normterm                */
#define _abnorm    4      /* abnormend               */
#define _enabled   4      /* codefenabled            */
#define _preven    12     /* coprevenabled           */
#define _parerr    16     /* coparerror              */
#define _maxexc    24     /* comaxexceed             */
```

`cdisco()` wertet nur die Strukturkomponente `coidret` (Kurzkennung des Contingency-Prozesses) aus.

Strukturkomponenten, die von `cdisco()` versorgt werden:

`secind` "Secondary Indicator", wie er nach Ausführung des `ENACO`-Makros im höchstwertigen Byte des Register 15 abgelegt wird (Werte 4 oder 20).

`rcode1` "Return Code", wie er nach Ausführung des `ENACO`-Makros im niedrigstwertigen Byte des Register 15 abgelegt wird (Werte 0 oder 4).

Siehe auch `cenaco()`.

ceil, ceilf, ceill - Gleitpunktzahl aufrunden

Syntax #include <math.h>

double ceil(double *x*);
float ceilf(float *x*);
long double ceill(long double *x*);

Beschreibung

`ceil()`, `ceilf()` und `ceill` runden eine Gleitpunktzahl *x* nach oben ganzzahlig auf.

Returnwert Kleinste ganze Zahl vom Typ `double` bzw. `float` bzw. `long double` (größer oder gleich *x*) bei Erfolg.

`HUGE_VAL` bei Überlauf.
`errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `ceil()` schlägt fehl, wenn gilt:

`ERANGE` Überlauf, der Returnwert ist zu groß.

Hinweis Der ganzzahlige Wert, der von `ceil()` bzw. `ceilf()` bzw. `ceill()` als `double` bzw. `float` bzw. `long double` zurückgegeben wird, kann nicht immer als `int` oder `long int` dargestellt werden. Das Ergebnis sollte stets überprüft werden, bevor es einer Variablen vom Typ `int` zugewiesen wird, um einen Integer-Überlauf abfangen zu können.

Um sicherzugehen, dass kein Fehler aufgetreten ist, sollte `errno` vor Aufruf von `ceil()`, `ceilf()` und `ceill()` auf 0 gesetzt werden. Ist nach der Ausführung `errno` ≠ 0, so ist ein Fehler aufgetreten.

Das Ergebnis von `ceil()`, `ceilf` und `ceill()` kann nur überlaufen, wenn für die Darstellung der Gleitpunktzahlen gilt: `DBL_MANT_DIG` > `DBL_MAX_EXP`.

Siehe auch `abs()`, `fabs()`, `floor()`, `floorf`, `floorl()`, `()isnan()`, `math.h`.

cenaco - Contingency-Routine definieren *(BS2000)*

Syntax #include <cont.h>

void cenaco(struct enacop **enacopar*);

Beschreibung

`cenaco()` definiert eine Contingency-Routine (TU bzw. P1), d.h. eine vom Anwender geschriebene Routine wird damit als Contingency-Routine angemeldet. Ausführliche Informationen zu Contingency-Routinen finden Sie im Abschnitt „Contingency- und STXIT-Routinen“ auf Seite 121 und im Handbuch „BS2000/OSD-BC Makroaufrufe an den Ablaufteil“.

Die Struktur `enacop` ist wie folgt in `cont.h` definiert:

```
struct enacop
{
  char resrv1 [7];        /* reserved for int. use   */
  char coname [54];       /* name of cont. routine   */
  char resrv2 [15];       /* reserved for int. use   */
  char level;             /* priority of cont.rout.  */
  int  (*econt)();        /* start adr of cont.rout. */
  int  comess;            /* contingency message     */
  char coidret [4];       /* contingency identifier  */
  errcod secind;          /* secondary indicator     */
  char resrv3 [2];        /* reserved for int. use   */
  errcod rcode1;          /* return code             */
};

#define errcod     char
#define _norm      0      /* normterm                */
#define _abnorm    4      /* abnormend               */
#define _enabled   4      /* codefenabled            */
#define _preven    12     /* coprevenabled           */
#define _parerr    16     /* coparerror              */
#define _maxexc    24     /* comaxexceed             */
```

Einige Strukturkomponenten müssen bzw. können Sie vor dem `cenaco`-Aufruf selbst versorgen, in anderen Einträgen legt `cenaco()` während des Ablaufs Informationen ab.

Einträge, die vom Anwender versorgt werden:

`coname` Name des Contingency-Prozesses. Der Name ist max. 54 Byte lang (ohne Nullbyte), muss in Großbuchstaben geschrieben und mit mindestens einem Leerzeichen abgeschlossen werden (ein Nullbyte unmittelbar hinter dem eigentlichen Namen wird vom System nicht als Endekriterium erkannt). Für die Versorgung von `coname` eignet sich z.B. die Funktion `strfill()`. Die Versorgung ist obligatorisch.

`level` Prioritätsstufe des Contingency-Prozesses. Die Versorgung ist obligatorisch. Es sind Werte von 1 - 126 zulässig.

`econt` Startadresse der Contingency-Routine. Die Versorgung ist obligatorisch.

`comess` Contingency-Message. Die Versorgung ist fakultativ. Der Wert wird als Parameter an die Contingency-Routine übergeben.

Einträge, die von `cenaco()` versorgt werden:

`coidret` Kurzkennung des Contingency-Prozesses. Diese Kurzkennung muss in weiteren Makros (z.B. `SOLSIG`) zur Bezeichnung des Contingency-Prozesses verwendet werden.

`secind` "Secondary Indicator", wie er nach Ausführung des `ENACO`-Makros im höchstwertigen Byte des Register 15 abgelegt wird (Werte 4 oder 20).

`rcode1` "Return Code", wie er nach Ausführung des `ENACO`-Makros im niedrigstwertigen Byte des Register 15 abgelegt wird (Werte 0 oder 4).

Hinweis Es können maximal 255 Contingency-Routinen definiert werden.

Siehe auch `cdisco()`, `cstxit()`, `signal()`, `alarm()`, `raise()`, `sleep()`.

cfgetispeed - Eingabe-Baudrate ermitteln

Syntax #include <termios.h>

speed_t cfgetispeed(const struct termios *_termios_p_);

Beschreibung

`cfgetispeed()` ermittelt die Eingabe-Baudrate aus der `termios`-Struktur, auf die *termios_p* zeigt. `cfgetispeed()` gibt nur den Wert aus der `termios`-Datenstruktur zurück.

Erweiterung
Da verschiedene Baudraten von der Hardware nicht unterstützt werden, ist es nur relevant, ob dieser Wert gleich null oder ungleich null ist. Weitere Details siehe `tcsetattr()`. ❑

Returnwert Eingabe-Baudrate vom Typ `speed_t`
bei Erfolg.

Siehe auch `cfgetospeed()`, `cfsetispeed()`, `cfsetospeed()`, `tcgetattr()`, `termios.h`, Abschnitt „Allgemeine Terminalschnittstelle“ auf Seite 96.

cfgetospeed - Ausgabe-Baudrate ermitteln

Syntax #include <termios.h>

speed_t cfgetospeed(const struct termios *_termios_p_);

Beschreibung

`cfgetospeed()` ermittelt die Ausgabe-Baudrate aus der `termios`-Struktur, auf die *termios_p* zeigt. `cfgetospeed()` gibt nur den Wert aus der `termios`-Datenstruktur zurück.

Returnwert Ausgabe-Baudrate vom Typ `speed_t`
bei Erfolg.

Erweiterung
Da verschiedene Baudraten von der Hardware nicht unterstützt werden, ist es nur relevant, ob dieser Wert gleich null oder ungleich null ist. Weitere Details siehe `tcsetattr()`. ❑

Siehe auch `cfgetispeed()`, `cfsetispeed()`, `cfsetospeed()`, `tcgetattr()`, `termios.h`, Abschnitt „Allgemeine Terminalschnittstelle“ auf Seite 96.

cfsetispeed - Eingabe-Baudrate festlegen

Syntax

#include <termios.h>

int cfsetispeed(struct termios **termios_p*, speed_t *speed*);

Beschreibung

`cfsetispeed()` setzt die Eingabe-Baudrate in der `termios`-Struktur, auf die *termios_p* zeigt, auf den Wert von *speed*.

`cfsetispeed()` hat keinen Einfluss auf Hardware-Baudraten, solange nicht ein nachfolgender erfolgreicher Aufruf von `tcsetattr()` mit derselben `termios`-Struktur erfolgt ist.

Erweiterung
Es wird nur der betreffende Wert in der `termios`-Struktur geändert. Da verschiedene Baudraten von der Hardware nicht unterstützt werden, ist es nur relevant, ob dieser Wert gleich null oder ungleich null ist. Es können jedoch die unter `termios.h` definierten Baudraten angegeben und in der `termios`-Struktur gespeichert werden. Werden Baudraten angegeben, die nicht in `termios.h` definiert sind, erfolgt keine Speicherung. Es wird -1 zurückgegeben und `errno` erhält den Wert `EINVAL`. Weitere Details siehe `tcsetattr()`.

Wird die Eingabe-Baudrate auf null gesetzt, erhält sie den Wert der Ausgabe-Baudrate. Versuche, nicht unterstützte Hardware-Baudraten einzustellen, werden ignoriert. Dies gilt sowohl für die Änderung von Baudraten, die nicht von der Hardware unterstützt werden, als auch für die Einstellung von Eingabe- und Ausgabe-Baudraten auf unterschiedliche Werte, wenn die Hardware dies nicht unterstützt. ❑

Returnwert

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

Fehler

`cfsetispeed()` schlägt fehl, wenn gilt:

`EINVAL`	*speed* entspricht keiner gültigen Baudrate (z. B. 9999) oder der Wert von *speed* liegt nicht im zulässigen Wertebereich, der in `termios.h` definiert ist.

Siehe auch

`cfgetispeed()`, `cfgetospeed()`, `cfsetospeed()`, `tcsetattr()`, `termios.h`, Abschnitt „Allgemeine Terminalschnittstelle“ auf Seite 96.

cfsetospeed - Ausgabe-Baudrate festlegen

Syntax #include <termios.h>

int cfsetospeed (struct termios *_termios_p_, speed_t _speed_);

Beschreibung

`cfsetospeed()` setzt die Ausgabe-Baudrate in der `termios`-Struktur, auf die *termios_p* zeigt, auf den Wert von *speed*.

`cfgetospeed()` hat keinen Einfluss auf Hardware-Baudraten, solange nicht ein nachfolgender erfolgreicher Aufruf von `tcsetattr()` mit derselben `termios`-Struktur erfolgt ist.

Erweiterung
Es wird nur der betreffende Wert in der `termios`-Struktur geändert. Da verschiedene Baudraten von der Hardware nicht unterstützt werden, ist es nur relevant, ob dieser Wert gleich null oder ungleich null ist. Es können jedoch die unter `termios.h` definierten Baudraten angegeben und in der `termios`-Struktur gespeichert werden. Werden Baudraten angegeben, die nicht in der `termios.h` definiert sind, erfolgt keine Speicherung. Es wird -1 zurückgegeben und `errno` erhält den Wert `EINVAL`. Weitere Details siehe `tcsetattr()`. Die Null-Baudrate B0 wird benutzt, um die Verbindung zu beenden. Falls B0 angegeben wird, werden die Kontroll-Leitungen des Modems nicht länger angesprochen, wodurch üblicherweise die Verbindung beendet wird. ❑

Returnwert

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

Fehler `cfsetospeed()` schlägt fehl, wenn gilt:

`EINVAL`	*speed* entspricht keiner gültigen Baudrate oder der Wert von *speed* liegt nicht im zulässigen Wertebereich, der in `termios.h` definiert ist.

Siehe auch `cfgetispeed()`, `cfgetospeed()`, `cfsetispeed()`, `tcsetattr()`, `termios.h`, Abschnitt „Allgemeine Terminalschnittstelle“ auf Seite 96.

chdir - aktuelles Dateiverzeichnis wechseln

Syntax

```
#include <unistd.h>

int chdir(const char *path);
```

Beschreibung

`chdir()` macht das Verzeichnis, auf das *path* verweist, zum aktuellen Dateiverzeichnis. Dies ist der Startpunkt für Pfadsuchen nach Pfadnamen, die nicht mit `/` beginnen.

path zeigt auf den Pfadnamen eines Verzeichnisses.

Returnwert

0	bei Erfolg. Das angegebene Verzeichnis ist nun das aktuelle Arbeitsverzeichnis.
-1	bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `chdir()` schlägt fehl, wenn gilt:

`EACCES`	Für eine Komponente des Pfadnamens gibt es kein Durchsuchrecht.
`Erweiterung EFAULT`	*path* ist eine ungültige Adresse.
`EINTR`	Ein Signal wurde während des Systemaufrufs `chdir()` abgefangen.
`EIO`	Während des Lesens im oder Schreibens in das Dateisystem trat ein Ein- oder Ausgabefehler auf.
`ELOOP`	Während der Übersetzung von *path* waren zu viele symbolische Verweise vorhanden. ❑
`ENAMETOOLONG`	Die Länge von *path* ist größer als `{PATH_MAX}`, oder die Länge einer Komponente von *path* ist größer als `{NAME_MAX}`, während `{POSIX_NO_TRUNC}` aktiv ist.
`ENOENT`	Eine Komponente von *path* existiert nicht oder ist ein leerer Pfadname.
`ENOTDIR`	Eine Komponente des Pfadnamens ist kein Dateiverzeichnis.

Hinweis Die Änderung des aktuellen Dateiverzeichnisses wirkt für die Dauer des aktuellen Programmes (bzw. der aktuellen Shell). Wird ein Programm oder eine Shell neu gestartet, dann ist wieder das Home-Verzeichnis als aktuelles Dateiverzeichnis eingestellt.

Um ein Verzeichnis zum aktuellen Dateiverzeichnis zu machen, muss ein Prozess Ausführrechte (Suchen) für das Verzeichnis haben.

`chdir()` wirkt nur in dem jeweils aktiven Prozess und nur bis zur Beendigung des aktiven Programms.

`chdir()` wird nur für POSIX-Dateien ausgeführt.

Siehe auch `chroot()`, `fchdir()`, `getcwd()`, `unistd.h`.

chmod - Dateizugriffsrechte ändern

Syntax

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

Optional
#include <sys/types.h> ❑

int chmod(const char *path, mode_t mode);
```

Beschreibung

`chmod()` ändert `S_ISUID`, `S_ISGID` und die Schutzbits der Datei, die durch *path* angesprochen wird, in die entsprechenden Bits von *mode* um. Dazu muss die effektive Benutzernummer des Prozesses zum Eigentümer der Datei passen oder Sonderrechte besitzen.

`S_ISUID`, `S_ISGID` und die Schutzbits einer Datei werden in `sys/stat.h` beschrieben.

Wenn der aufrufende Prozess keine Sonderrechte besitzt und die Gruppennummer der Datei nicht zur effektiven Gruppennummer oder einer der weiteren passt und die Datei eine normale Datei ist, dann wird das Bit `S_ISGID` (Setze Gruppennummer bei Ausführung) in den Zugriffsrechten der Datei bei einer erfolgreichen Rückkehr von `chmod()` gelöscht.

Im C-Laufzeitsystem wird `chmod()` auch für offene Dateien ausgeführt. Andere X/Open-kompatible Systeme können für diesen Fall andere Vorgaben definieren.

Bei erfolgreicher Beendigung markiert `chmod()` das Feld `st_ctime` der Datei zum Aktualisieren.

Returnwert

0	bei Erfolg. Die Zugriffserlaubnis der angegebenen Datei ist entsprechend gesetzt.
-1	bei Fehler. Der Dateimodus wird nicht verändert. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler

`chmod` schlägt fehl, wenn gilt:

`EACCES`	Für eine Komponente des Pfadnamen-Anfangs existiert kein Durchsuchrecht.
Erweiterung	
`EFAULT`	*path* weist über den zugewiesenen Adressraum hinaus.
`EINTR`	Ein Signal wurde während der Ausführung des Systemaufrufs abgefangen. ❑
`EINVAL`	Der Wert von *mode* ist ungültig.
	Es wurde versucht, auf eine BS2000-Datei zuzugreifen.

Erweiterung

EIO	Ein Ein-/Ausgabe-Fehler trat während des Lesens oder Schreibens im Dateisystem auf.
ELOOP	Während der Übersetzung von *path* waren zu viele symbolische Verweise vorhanden. ❑
ENAMETOOLONG	Die Länge von *path* überschreitet {PATH_MAX}, oder eine Komponente des Pfadnamens ist länger als {NAME_MAX}.
ENOENT	*path* zeigt auf den Namen einer nicht existierenden Datei oder auf die leere Zeichenkette.
ENOTDIR	Eine Komponente von *path* ist kein Dateiverzeichnis.
EPERM	Die effektive Benutzernummer entspricht nicht dem Eigentümer der Datei, und der Prozess besitzt auch keine Sonderrechte.
EROFS	Die genannte Datei befindet sich in einem nur zum Lesen eingehängten Dateisystem.

Hinweis chmod() wird nur für POSIX-Dateien ausgeführt.

Siehe auch chown(), fchmod(), mkdir(), mkfifo(), open(), stat(), sys/types.h, sys/stat.h.

chown - Eigentümer und Gruppe einer Datei ändern

Syntax

#include <unistd.h>

Optional
#include <sys/types.h> ❑

int chown(const char **path*, uid_t *owner*, gid_t *group*);

Beschreibung

path zeigt auf einen Pfadnamen, der eine Datei bezeichnet. Die Benutzer- und Gruppennummer der benannten Datei werden auf die numerischen Werte gesetzt, die in *owner* und *group* enthalten sind.

Nur Prozesse, deren effektive Benutzernummer gleich der Benutzernummer der Datei ist oder die Sonderrechte haben, können die Benutzer- oder Gruppennummer einer Datei ändern. Wenn `{_POSIX_CHOWN_RESTRICTED}` für *path* aktiv ist, dann gilt:

– Die Änderung der Benutzernummer ist auf Prozesse mit Sonderrechten beschränkt.

– Die Änderung der Gruppennummer ist einem Prozess, dessen effektive Benutzernummer gleich der Benutzernummer der Datei ist, der aber keine Sonderrechte hat, nur dann erlaubt, wenn *owner* gleich der Benutzernummer der Datei und *group* entweder gleich der effektiven Benutzernummer des Prozesses oder aber gleich einer seiner weiteren Gruppennummern ist.

Wenn *path* eine normale Datei bezeichnet, dann werden die Bits `S_ISUID` und `S_ISGID` der Datei bei erfolgreicher Rückkehr von `chown()` gelöscht, solange der Aufruf nicht von einem Prozess mit Sonderrechten erfolgte. Ist dies der Fall, werden unter POSIX diese Bits nicht geändert. Wenn `chown()` erfolgreich für eine Datei aufgerufen wird, die keine normale Datei ist, dann können diese Bits gelöscht werden. Diese Bits sind in `sys/stat.h` definiert.

Wenn *owner* oder *group* als `(uid_t)-1` oder `(gid_t)-1` angegeben werden, dann wird die entsprechende Nummer der Datei nicht geändert.

Nach erfolgreicher Beendigung markiert `chown()` das Feld `st_ctime` der Datei zum Aktualisieren.

Returnwert

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

Fehler

`chown()` schlägt fehl, wenn gilt:

`EACCES`	Für eine Komponente von *path* existiert kein Durchsuchrecht.
Erweiterung `EFAULT`	Es wurde eine ungültige Adresse als Argument übergeben.

EINTR	Während des chown-Aufrufs wurde ein Signal abgefangen. ❑
EINVAL	Der Wert der angegebenen Benutzer- oder Gruppennummer wird nicht unterstützt, z.B. wenn der Wert kleiner als 0 ist, oder es wurde versucht, auf eine BS2000-Datei zuzugreifen.
Erweiterung	
EIO	Während des Lesens oder Schreibens im Dateisystem trat ein Ein-/Ausgabefehler auf.
ELOOP	Während der Übersetzung von *path* waren zu viele symbolische Verweise vorhanden. ❑
ENAMETOOLONG	Die Länge des Arguments *path* überschreitet {PATH_MAX}, oder eine Pfadnamen-Komponente ist länger als {NAME_MAX}.
ENOENT	*path* zeigt auf eine Datei, die nicht existiert oder auf eine leere Zeichenkette.
ENOTDIR	Eine Komponente von *path* ist kein Dateiverzeichnis.
EPERM	Die effektive Benutzernummer passt nicht zum Eigentümer der Datei oder der aufrufende Prozess besitzt keine Sonderrechte, obwohl {_POSIX_CHOWN_RESTRICTED} anzeigt, dass Sonderrechte gefordert werden.
EROFS	Die bezeichnete Datei befindet sich in einem nur zum Lesen eingehängten Dateisystem.

Hinweis chown() wird nur für POSIX-Dateien ausgeführt.

Siehe auch chmod(), sys/stat.h, sys/types.h, unistd.h.

chroot - Root-Verzeichnis ändern

Syntax

```
#include <unistd.h>

int chroot(const char *path);
```

Beschreibung

path zeigt auf einen Pfadnamen, der ein Dateiverzeichnis bezeichnet. Die Funktion `chroot()` bewirkt, dass das bezeichnete Dateiverzeichnis zum Root-Verzeichnis wird, dem Anfangspunkt für alle Pfadnamen, die mit dem Zeichen '/' beginnen. Das aktuelle Dateiverzeichnis des Benutzers wird durch `chroot()` nicht beeinflusst.

Der Prozess muss Sonderrechte haben, um das Root-Verzeichnis ändern zu können. Der Eintrag .. im Root-Verzeichnis wird so interpretiert, dass er das Root-Verzeichnis selbst bedeutet. Daher kann .. nicht dazu verwendet werden, auf Dateien außerhalb des beim Root-Verzeichnis beginnenden Unterbaums zuzugreifen.

`chroot()` ist nicht reentrant.

Returnwert

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

Fehler

`chroot()` schlägt fehl, wenn gilt:

`EACCES`	Für eine Komponente von *path* existiert kein Durchsuchrecht.
Erweiterung `EFAULT`	Es wurde eine ungültige Adresse als Argument übergeben.
`EINTR`	Ein Signal wurde während des Systemaufrufs `chroot()` abgefangen.
`ELOOP`	Während der Übersetzung von *path* waren zu viele symbolische Verweise vorhanden. ❑
`ENAMETOOLONG`	Die Länge des Arguments *path* überschreitet `{PATH_MAX}`, oder eine Pfadnamen-Komponente ist länger als `{NAME_MAX}`.
`ENOENT`	*path* zeigt auf den Namen eines Dateiverzeichnisses, das nicht existiert, oder auf die leere Zeichenkette.
`ENOTDIR`	Eine Komponente des Pfadnamens *path* ist kein Dateiverzeichnis.
`EPERM`	Die effektive Benutzernummer ist nicht die eines Prozesses mit Sonderrechten.

Hinweis

`chroot()` wird nur für POSIX-Dateiverzeichnisse ausgeführt.
`chroot()` wirkt nur im jeweils aktiven Prozess und nur bis zur Beendigung des Prozesses.

Siehe auch `chdir()`, `unistd.h`.

clearerr - Dateiende- und Fehlerkennzeichen zurücksetzen

Syntax #include <stdio.h>

void clearerr(FILE **stream*);

Beschreibung

`clearerr()` setzt das Dateiende- und Fehlerkennzeichen für den Datenstrom, auf den *stream* zeigt, zurück.

BS2000

`clearerr()` ist sowohl als Makro als auch als Funktion realisiert.

`clearerr()` ist auch auf Dateien mit Satz-Ein-/Ausgabe anwendbar. ❑

Hinweis Ob `clearerr()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

Siehe auch `feof()`, `ferror()`, `stdio.h`.

clock - CPU-Zeitverbrauch eines Prozesses ermitteln

Syntax

```
#include <time.h>

clock_t clock(void);
```

Beschreibung

Je nach Wahl der Funktionalität verhält sich `clock()` unterschiedlich wie folgt (siehe Abschnitt „Umfang der unterstützten C-Bibliothek“ auf Seite 18):

Bei POSIX-Funktionalität gibt `clock()` die seit dem ersten `clock`-Aufruf vergangene CPU-Zeit zurück.

BS2000
Bei BS2000-Funktionalität gibt `clock()` die CPU-Zeit seit Programmbeginn zurück. ❑

Returnwert

Betrag der verbrauchten CPU-Zeit seit dem letzten `clock`-Aufruf
bei Erfolg. Beim ersten `clock`-Aufruf ist der Returnwert 0.

BS2000
CPU-Zeit seit Programmbeginn
bei Erfolg. ❑

`(clock_t)-1` wenn die CPU-Zeit nicht verfügbar oder darstellbar ist. Dies gilt für POSIX- und BS2000-Funktionalität.

Hinweis

Der Returnwert von `clock()` wird in zehntausendstel Sekunden definiert. Dies geschieht aus Kompatibilitätsgründen zu Systemen, die CPU-Takte mit hohen Auflösungen haben. Deshalb kann der Returnwert von `clock()` auf manchen Systemen in 0 überlaufen. Auf einem System mit `clock_t`-Werten von 32 Bit geschieht dies nach 2147 Sekunden bzw. 36 Minuten.

Wenn die CPU-Zeit in Sekunden angegeben werden soll, muss der Returnwert von `clock()` durch den Wert des Makros `CLOCKS_PER_SEC` dividiert werden (siehe `time.h`).

Siehe auch

`asctime()`, `cputime()`, `ctime()`, `difftime()`, `gmtime()`, `localtime()`, `mktime()`, `strftime()`, `strptime()`, `system()`, `time()`, `times()`, `utime()`, `wait()`, `time.h`, Abschnitt „Umfang der unterstützten C-Bibliothek“ auf Seite 18.

close - Datei schließen

Syntax #include <unistd.h>

int close(int *fildes*) ;

Beschreibung

fildes ist ein Dateideskriptor, der von einem Systemaufruf `creat()`, `open()`, `dup()`, `fcntl` oder `pipe()` geliefert wurde. `close()` schließt die durch *fildes* angegebene Datei. Alle dem Prozess zugeordneten bestehenden Satzsperren der von *fildes* angegebenen Datei werden aufgehoben.

Wird die Funktion `close()` durch ein Signal unterbrochen, das abgefangen werden soll, dann liefert sie -1 und setzt *errno* auf `EINTR`; der Zustand von *fildes* ist danach unbestimmt.

Sind alle Dateideskriptoren, die einer Pipe oder einer FIFO-Gerätedatei zugeordnet waren, geschlossen, werden alle Daten verworfen, die noch in dieser Pipe oder FIFO enthalten waren.

Sind alle Dateideskriptoren, die einer Dateibeschreibung zugeordnet waren, geschlossen, wird die Dateibeschreibung freigegeben.

Ist der Verweiszähler der Datei gleich 0, und sind alle Dateideskriptoren zu dieser Datei geschlossen, wird der von dieser Datei belegte Platz freigegeben. Es kann nicht länger darauf zugegriffen werden.

Erweiterung
Wenn ein Datenstrom geschlossen wird und für den aufrufenden Prozess vorher registriert wurde, dass ihm ein `SIGPOLL`-Signal (siehe `signal()` und `sigset()`) bei Ereignissen in Zusammenhang mit dieser Datei geschickt wird, wird die Registrierung des aufrufenden Prozesses für Ereignisse in Zusammenhang mit dieser Datei aufgehoben. Das letzte `close()` auf eine Datei bewirkt, dass die zu *fildes* gehörende Datei beseitigt wird. Wenn `O_NDELAY` und `O_NONBLOCK` nicht gesetzt sind und keine Signale für die Datei abgesetzt wurden, wartet `close()` bis zu 15 Sekunden auf jedes Modul und jeden Treiber, damit in der Warteschlange stehende Ausgaben vor Beseitigen der Datei gemacht werden können. Wenn `O_NDELAY` oder `O_NONBLOCK` gesetzt wurde oder wenn Signale vorliegen, wartet `close()` nicht auf die Beendigung der Ausgabe und löscht die Datei sofort. ❑

Returnwert

0	bei Erfolg. Die angegebene Datei ist geschlossen.
-1	bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `close()` schlägt fehl, wenn gilt:

`EBADF`	*fildes* ist kein gültiger Dateideskriptor, oder die BS2000-Datei ist in diesem Prozess nicht zugreifbar.
`EINTR`	`close()` wurde durch ein Signal unterbrochen.

Hinweis Wurde die Datei mit `fopen()` geöffnet, muss sie statt mit `close()` mit `fclose()` geschlossen werden.

Bei Beendigung eines Programms (normal oder mit `exit()`) werden automatisch alle offenen Dateien geschlossen.

Ob `close()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

Siehe auch `creat()`, `dup()`, `fcntl()`, `lseek()`, `open()`, `read()`, `tell()`, `write()`, `unistd.h`.

closedir - Dateiverzeichnis schließen

Syntax #include <dirent.h>

Optional
#include <sys/types.h> ❑

int closedir(DIR **dirp*);

Beschreibung

`closedir()` schließt den Dateiverzeichnisstrom, der durch *dirp* angegeben wird. Nach der Rückkehr zeigt der Wert von *dirp* nicht mehr auf ein zugreifbares Objekt des Typs `DIR`. Der in der `DIR`-Struktur verwendete Dateideskriptor wird geschlossen.

Returnwert

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

Fehler `closedir()` schlägt fehl, wenn gilt:

`EBADF`	Das Argument *dirp* bezieht sich nicht auf einen offenen Dateiverzeichnisstrom.
`EINTR`	`closedir()` wurde von einem Signal unterbrochen.

Hinweis `closedir()` wird nur für POSIX-Dateien ausgeführt.

Siehe auch `opendir()`, `dirent.h`, `sys/types.h`.

closelog, openlog, setlogmask, syslog - Systemprotokoll steuern

Syntax

#include <syslog.h>

void closelog(void)

void openlog(const char **ident*, int *logopt*, int *facility*);

int setlogmask(int *maskpri*);

void syslog(int *priority*, const char **message*, ... /* Argumente */);

Beschreibung

`syslog()` schreibt eine Meldung *message* in die Datei `/var/adm/messages`. Die Meldung besteht aus einem Meldungskopf und dem Meldungstext. Der Meldungskopf enthält die Angabe des Meldungsgewichts, die Zeitangabe, das Kennzeichen `syslog` und optional die Prozess-Id.

Der Meldungstext wird aus dem Argument *message* und den darauf folgenden Argumenten erzeugt, als ob diese Argumente an `printf()` übergeben worden seien, außer dass `%m` im Formatstring, auf den *message* zeigt, durch die Fehlermeldung ersetzt wird, die dem aktuellen Wert von `errno` entspricht. Ein nachgestelltes NEWLINE wird ggf. hinzugefügt.

Werte für *priority* werden durch inklusives ODER aus Meldungsgewicht und ggfs. Funktionswert gebildet. Wenn für *facility* keine Funktion angegeben wurde, wird die vordefinierte Standardfunktion verwendet.

Mögliche Werte für das Meldungsgewicht sind:

`LOG_EMERG`	Eine „panic“-Bedingung. Diese Bedingung wird normalerweise an alle Benutzer übertragen.
`LOG_ALERT`	Eine Bedingung, die sofort korrigiert werden sollte, beispielsweise eine beschädigte Systemdatenbank.
`LOG_CRIT`	Schwerwiegende Bedingung, beispielsweise Festplattenfehler.
`LOG_ERR`	Fehler.
`LOG_WARNING`	Warnmeldungen.
`LOG_NOTICE`	Bedingungen, bei denen es sich nicht um Fehlerbedingungen handelt, die jedoch spezielle Schritte erfordern.
`LOG_INFO`	Informationsmeldungen.
`LOG_DEBUG`	Meldungen mit Informationen, die normalerweise nur beim Testen eines Programms verwendet werden.

facility gibt an, welche Anwendung oder welche Systemkomponente die Meldung erzeugt hat. Mögliche Werte sind:

LOG_USER	Von wahlfreien Benutzerprozessen erstellte Meldungen. Dies ist die standardmäßige Funktions-ID, wenn keine andere angegeben ist.
LOG_LOCAL0	Reserviert für lokale Verwendung.

LOG_LOCAL1	Reserviert für lokale Verwendung.
LOG_LOCAL2	Reserviert für lokale Verwendung.
LOG_LOCAL3	Reserviert für lokale Verwendung.
LOG_LOCAL4	Reserviert für lokale Verwendung.
LOG_LOCAL5	Reserviert für lokale Verwendung.
LOG_LOCAL6	Reserviert für lokale Verwendung.
LOG_LOCAL7	Reserviert für lokale Verwendung.

`openlog()` setzt Prozessattribute, die nachfolgende Aufrufe von `syslog()` steuern. Das Argument *ident* ist eine Zeichenkette, die jeder Meldung vorangestellt wird. *logopt* ist ein Bitfeld, in dem Protokolloptionen angezeigt werden. Werte für *logopt* werden durch bitweises inklusives ODER aus beliebig vielen der folgenden Werte erzeugt. Folgendes sind aktuelle Werte für *logopt*:

`LOG_PID`	Protokolliert die Prozess-ID mit jeder Meldung. Dies ist nützlich für die Identifizierung spezieller Prozesse.
`LOG_CONS`	Gibt Meldungen auf der Systemkonsole aus, wenn sie nicht in die Datei `/var/adm/messages` geschrieben werden können. Diese Option kann in Prozessen, die nicht über ein Steuerungsterminal verfügen, sicher verwendet werden, da `syslog()` vor dem Öffnen der Konsole einen Kindprozess erzeugt.
`LOG_NDELAY`	Öffnet die Datei `/var/adm/messages` bei Ausführung von `openlog()`. Normalerweise wird das Öffnen verzögert, bis die erste Meldung protokolliert wird. Diese Option ist für Programme nützlich, die bei der Zuordnung von Dateideskriptoren auf die Reihenfolge achten müssen.
LOG_ODELAY	Verzögert das Öffnen, bis `syslog()` aufgerufen wird.
`LOG_NOWAIT`	Wartet nicht auf Kindprozesse, die zum Protokollieren von Meldungen auf der Konsole aufgespaltet wurden. Diese Option sollte von Prozessen verwendet werden, die mit Hilfe von SIGCHLD eine Benachrichtigung über das Beenden eines Kindprozesses ausgeben, da `syslog()` andernfalls möglicherweise durch das Warten auf einen Kindprozess blockiert wird, dessen Endestatus bereits erreicht ist.

Das Argument *facility* codiert eine Standardfunktion, die allen Meldungen zugeordnet werden soll, die nicht über eine bereits codierte explizite Funktion verfügen. Die Voreinstellung für die Standardfunktion ist `LOG_USER`.

Die Funktionen `openlog()` und `syslog()` können Dateideskriptoren zuordnen. Es ist nicht notwendig, `openlog()` vor `syslog()` aufzurufen.

Die Funktion `closelog()` schließt alle offenen Dateideskriptoren, die durch vorhergehende Aufrufe von `openlog()` und `syslog()` zugeordnet wurden.

`setlogmask()` setzt die Protokollprioritätsmaske für den aktuellen Prozess auf *maskpri* und gibt die vorherige Maske zurück. Wenn *maskpri* den Wert 0 hat, wird die aktuelle Protokollprioritätsmaske nicht geändert. Aufrufe von `syslog()` durch den aktuellen Prozess, deren Priorität nicht in *maskpri* angegeben ist, werden zurückgewiesen. Die Maske für eine be-

stimmte Priorität *pri* wird vom Makro LOG_MASK(*pri*) berechnet. Die Maske für alle Prioritäten bis zu *toppri* einschließlich werden im Makro LOG_UPTO(*toppri*) angegeben. Standardmäßig können alle Prioritäten protokolliert werden.

Symbolische Konstanten, die als Werte für *logopt*, *facility*, *priority* und *maskpri* verwendet werden können, sind in der Include-Datei syslog.h definiert.

Returnwert setlogmask():

Vorherige Protokollprioritätsmaske.

Siehe auch printf(), syslog.h.

compile - regulären Ausdruck übersetzen

Syntax #include <regexp.h>

int compile(char **instring*, char **exbuf*, const char **endbuf*, int *eof)*;

Beschreibung

Siehe regexp().

Hinweis Diese Funktion wird zukünftig vom X/Open-Standard nicht mehr unterstützt.

confstr - Zeichenketten-Wert einer Systemvariablen ermitteln

Syntax

```
#include <unistd.h>

size_t confstr(int name, char *buf, size_t len);
```

Beschreibung

Mit `confstr()` können die aktuellen Zeichenketten-Werte einer konfigurierbaren Systemvariablen *name* ermittelt werden. Verwendung und Zweck entsprechen `sysconf()`; `confstr()` wird jedoch nur für Zeichenketten-Werte und nicht für numerische Werte verwendet.

Die Implementierung unterstützt nur den Wert `_CS_PATH` für *name*, der in `unistd.h` definiert ist.

Wenn *len* ungleich 0 ist und *name* einen von der Konfiguration definierten Wert hat, kopiert `confstr()` diesen Wert in den Puffer mit *len* Byte, auf den *buf* zeigt. Wenn die zurückzugebende Zeichenkette mehr als *len* Byte hat (einschließlich abschließendem Nullbyte), so schneidet `confstr()` die Zeichenkette auf *len*-1 Byte ab und fügt ein abschließendes Nullbyte an das Ergebnis an. Die Anwendung kann erkennen, dass die Zeichenkette abgeschnitten wurde, wenn sie den von `confstr()` zurückgelieferten Wert mit *len* vergleicht.

Wenn *len* gleich 0 und *buf* ein Nullzeiger ist, gibt `confstr()` den Integer-Wert wie unten definiert zurück. Es wird aber keine Zeichenkette zurückgegeben. Wenn *len* gleich 0, aber *buf* kein Nullzeiger ist, ist kein Returnwert definiert.

Returnwert

Puffergröße für den Wert von *name*
: wenn *name* einen von der Konfiguration definierten Wert hat. Wenn dieser Returnwert größer als *len* ist, wird die in *buf* zurückgelieferte Zeichenkette abgeschnitten.

0
: wenn *name* keinen von der Konfiguration definierten Wert hat. `errno` wird nicht gesetzt.

: wenn *name* einen ungültigen Wert hat. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler

`confstr()` schlägt fehl, wenn gilt:

`EINVAL`
: Der Wert von *name* ist ungültig.

Hinweis

Eine Anwendung kann zwischen einem ungültigen Wert für *name* und einem Namen unterscheiden, der einer konfigurierbaren Umgebungsvariablen entspricht, die keinen von der Konfiguration definierten Wert hat. Dazu muss abgefragt werden, ob `errno` verändert wurde. Dies entspricht dem Verhalten von `sysconf()`.

Ursprünglich wurde `confstr()` benötigt, um den von der Konfiguration vordefinierten Wert für die Umgebungsvariable `PATH` abzufragen. Da der Benutzer `PATH` erweitern kann, müssen Anwendungen den Wert der vom System zur Verfügung gestellten Umgebungsvariablen `PATH` feststellen können, der den richtigen Suchpfad für die XPG4-Kommandos enthält.

Siehe auch `sysconf()`, `pathconf()`, `unistd.h`.

cos - Cosinus berechnen

Syntax #include <math.h>

double cos(double *x*);

Beschreibung

`cos()` berechnet für die Gleitpunktzahl *x*, die den Winkel im Bogenmaß angibt, d.h. die trigonometrische Funktion Cosinus.

Returnwert cos(*x*)

bei Erfolg. Der Returnwert ist eine Gleitpunktzahl im Intervall [-1.0, +1.0].

Siehe auch `acos()`, `asin()`, `atan()`, `atan2()`, `sin()`, `tan()`, `math.h`.

cosh - Cosinus hyperbolicus berechnen

Syntax
```
#include <math.h>
double cosh(double x);
```

Beschreibung
`cosh()` berechnet den Cosinus hyperbolicus für die Gleitpunktzahl *x*.

Returnwert

cosh(*x*)	bei Erfolg.
`HUGE_VAL`	bei Überlauf. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `cosh()` schlägt fehl, wenn gilt:

`ERANGE`	Der Wert von *x* verursacht einen Überlauf.

Siehe auch `acos()`, `asin()`, `atan()`, `cos()`, `sinh()`, `tanh()`, `math.h`.

cputime - CPU-Zeitverbrauch einer Task ermitteln *(BS2000)*

Syntax
```
#include <stdlib.h>
int cputime(void);
```

Beschreibung
`cputime()` liefert den CPU-Zeitverbrauch der aktuellen Task (seit `LOGON`).

Returnwert CPU-Zeitverbrauch in zehntausendstel Sekunden.

Siehe auch `clock()`, `stdlib.h`, Abschnitt „Umfang der unterstützten C-Bibliothek“ auf Seite 18.

creat - neue Datei erzeugen oder vorhandene überschreiben

Name **creat, creat64**

Syntax

```
#include <fcntl.h>

Optional
#include <sys/types.h>
#include <sys/stat.h>

int creat(const char *path, mode_t mode);
int creat64(const char *path, mode_t mode);

BS2000
int creat(const char *path, int mode);
```

Beschreibung

Wenn POSIX-Dateien erzeugt werden, verhält sich die Funktion XPG-konform wie folgt:

`creat()` erstellt eine neue Datei oder bereitet eine vorhandene Datei vor, neu beschrieben zu werden. Die Datei wird durch den Pfadnamen angegeben, auf den *path* zeigt.

Wenn die Datei vorhanden ist, wird ihre Länge auf 0 abgeschnitten; Modus und Eigentümer bleiben unverändert.

Wenn die Datei nicht existiert, wird die Datei-Eigentümernummer auf die effektive Benutzernummer des Prozesses gesetzt. Die Gruppennummer der Datei wird auf die effektive Gruppennummer des Prozesses gesetzt; wenn aber das `S_ISGID`-Bit im übergeordneten Verzeichnis gesetzt ist, dann wird die Gruppennummer der Datei vom übergeordneten Verzeichnis geerbt. Die Zugriffserlaubnis-Bits des Dateimodus werden auf den Wert von *mode* wie folgt geändert:

– Wenn die Gruppennummer der neuen Datei nicht zur effektiven oder einer der zusätzlichen Gruppennummern passt, wird das `S_ISGID`-Bit gelöscht.

– Alle Bits, die in der Dateityp-Erstellungsmaske des Prozesses gesetzt sind, werden gelöscht (siehe `umask()`).

– Das Bit für die Sicherung des Textsegments nach der Ausführung wird gelöscht (siehe `chmod()`).

Nach erfolgreicher Beendigung wird ein Dateideskriptor mit reinem Schreibrecht zurückgegeben, und die Datei wird zum Schreiben geöffnet, auch wenn der Modus das Schreiben nicht zulässt. Der Lese-/Schreibzeiger wird auf den Anfang der Datei gesetzt. Die Datei bleibt standardmäßig bei `exec`-Systemaufrufen geöffnet (siehe `fcntl()`). Eine neue Datei kann mit einem Modus geöffnet werden, der Schreiben nicht zulässt.

Der Aufruf `creat(`*path*, *mode*`)` entspricht dem Aufruf von

`open(`*path*`, O_WRONLY | O_CREAT | O_TRUNC,` *mode*`)`

Es besteht kein funktionaler Unterschied zwischen `creat()` und `creat64()`, außer dass in der mit dem Filedescriptor verknüpften Dateibeschreibung das Kennzeichen für eine große Datei hinterlegt wird, dh. es wird das O_LARGEFILE Bit gesetzt. Es wird eine Dateikennzahl zurückgegeben, die dazu verwendet werden kann, die Datei über 2GB hinaus zu vergrößern.

BS2000
Wenn BS2000-Dateien erzeugt werden, ist Folgendes zu beachten:

path kann sein:

– jeder gültige BS2000-Dateiname

– "`link=`*linkname*"
linkname bezeichnet einen BS2000-Linknamen.

mode wird ignoriert. Zur Erstellung von portierbaren Programmen ist *mode* jedoch notwendig, da damit im UNIX-Betriebssystem die Schutzbitvergabe geregelt wird.

Der BS2000-Dateiname bzw. -Linkname kann in Klein- und Großbuchstaben geschrieben werden, er wird automatisch in Großbuchstaben umgesetzt.

Wird eine nicht vorhandene Datei angelegt, wird standardmäßig folgende Datei erzeugt:
Bei KR-Funktionalität (nur bei C/C++ Versionen kleiner V3 vorhanden) eine SAM-Datei mit variabler Satzlänge und Standardblocklänge,
bei ANSI-Funktionalität eine ISAM-Datei mit variabler Satzlänge und Standardblocklänge.

Bei Verwendung eines Linknamens lassen sich mit dem `ADD-FILE-LINK`-Kommando folgende Dateiattribute ändern: Zugriffsmethode, Satzlänge, Satzformat, Blocklänge und Blockformat.
Wird eine bereits existierende Datei auf die Länge 0 verkürzt, bleiben die Katalogeigenschaften dieser Datei erhalten.

Es können maximal `_NFILE` Dateien gleichzeitig geöffnet sein. `_NFILE` ist in `stdio.h` mit 2048 definiert.

Returnwert

Dateideskriptor	bei Erfolg.
-1	bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen. Es wird keine Datei geöffnet oder modifiziert.

Fehler `creat()` schlägt fehl, wenn gilt:

`EACCES` Eine Komponente des Pfades darf nicht durchsucht werden.

Die Datei ist nicht vorhanden, und das Dateiverzeichnis, in dem die Datei angelegt werden soll, lässt das Schreiben nicht zu.
Die Datei ist vorhanden, und die Schreiberlaubnis wird verweigert.

Erweiterung
`EAGAIN` Die Datei ist vorhanden, obligatorisches Sperren von Dateien und Dateisätzen ist gesetzt, und in der Datei sind noch Dateisatzsperren vorhanden (siehe `chmod()`).

`EEXIST` `O_CREAT` und `O_EXCL` sind gesetzt, und der Dateiname existiert schon.

Erweiterung
`EFAULT` *path* weist über den zugewiesenen Adressraum des Prozesses hinaus.

`EINTR` Ein Signal wurde während des Systemaufrufs `creat()` abgefangen.

`EISDIR` Die angegebene Datei ist ein Dateiverzeichnis.

Erweiterung
`ELOOP` Während der Übersetzung von *path* waren zu viele symbolische Verweise vorhanden.

`EMFILE` Der Prozess hat zu viele Dateien geöffnet (siehe `getrlimit()`).

`ENAMETOOLONG`
Die Länge des *path*-Arguments überschreitet `{PATH_MAX}`, oder eine *path*-Komponente ist länger als `{NAME_MAX}`.

`ENFILE` Die Systemdatei-Tabelle ist voll.

`ENOENT` Eine Komponente des Pfadnamens existiert nicht, oder *path* zeigt auf eine leere Zeichenkette.

`ENOSPC` Das Dateisystem hat keine Indexeinträge mehr.

`ENOTDIR` Eine Komponente des Pfadnamens ist kein Dateiverzeichnis.

`ENXIO` Die angegebene Datei ist eine Gerätedatei für ein zeichen- oder blockorientiertes Gerät, und das dieser Datei zugewiesene Gerät existiert nicht.

`EROFS` Die angegebene Datei steht in einem schreibgeschützten Dateisystem.

`ETXTBSY` Die Datei ist eine reine Programmdatei, die gerade ausgeführt wird.

Hinweis Ob eine BS2000- oder eine POSIX-Datei erzeugt wird, hängt von der Programmumgebung ab.

Siehe auch `chmod()`, `close()`, `dup()`, `fcntl()`, `getrlimit()`, `lseek()`, `open()`, `read()`, `umask()`, `write()`, `stat()`, `fcntl.h`, `sys/stat.h`, `sys/types.h`.

crypt - Zeichenkette algorithmisch verschlüsseln

Syntax #include <unistd.h>

char *crypt(const char **key*, const char **salt*);

Beschreibung

`crypt()` ist eine Verschlüsselungsfunktion für Zeichenketten. Die Funktion verwendet einen Einweg-Verschlüsselungsalgorithmus mit Variationen, die die Anwendung von Hardware-Implementierungen für eine Schlüsselsuche verhindern sollen.

key ist die zu verschlüsselnde Eingabefolge, zum Beispiel das Passwort eines Benutzers. *salt* ist eine Zeichenkette der Länge zwei aus den Zeichen (a-z, A-Z, 0-9, . , /).
Diese Zeichenkette wird zur Veränderung des Verschlüsselungsalgorithmus auf eine von 4096 verschiedenen Arten verwendet; danach wird die Eingabefolge als Schlüssel zum wiederholten Verschlüsseln einer konstanten Zeichenkette benutzt. Der jeweils zurückgegebene Wert zeigt auf die verschlüsselte Zeichenkette.

Returnwert Zeiger auf verschlüsselte Zeichenkette.
Die ersten beiden Zeichen der verschlüsselten Zeichenkette sind die Zeichen von *salt*.

Nullzeiger bei Fehler.
`errno` wird gesetzt, um den Fehler anzuzeigen.

Hinweis Das Ergebnis von `crypt()` zeigt auf statische Daten, die bei jedem Aufruf überschrieben werden.

Siehe auch `encrypt()`, `setkey()`, `unistd.h`.

cstxit - STXIT-Routine definieren *(BS2000)*

Syntax #include <stxit.h>

void cstxit(struct stxitp *stxitpar*);

Beschreibung

`cstxit()` definiert eine STXIT-Routine, d.h. eine vom Anwender geschriebene Routine wird damit als STXIT-Routine angemeldet.

Ausführliche Informationen zur Programmierung von STXIT-Routinen finden Sie im Abschnitt „Contingency- und STXIT-Routinen“ auf Seite 121 und im Handbuch „BS2000/OSD-BC Makroaufrufe an den Ablaufteil“.

Die Struktur `stxit` ist in `stxit.h` wie folgt definiert:

```
struct stxitp
{
  addr     bufadr;      /* Adresse der Mitteilung an das Programm (OPINT) */
  enum err_set retcode; /* Returncode */
  struct cont contp;    /* Adresse der STXIT-Routinen */
  struct nest nestp;    /* max. Schachtelungstiefe */
  struct stx  stxp;     /* Steuerung des cstxit-Aufrufs */
  struct diag diagp;    /* Diagnosesteuerung */
  struct type typep;    /* Parameterübergabe-Modus */
};


struct cont             /* Adresse der STXIT-Routine für */
{                       /* die jeweilige Ereignisklasse */
  int  (*prchk) ();
  int  (*timer) ();
  int  (*opint) ();
  int  (*error) ();
  int  (*runout) ();
  int  (*brkpt) ();
  int  (*abend) ();
  int  (*pterm) ();
  int  (*rtimer) ();
};

struct nest             /* max. Schachtelungstiefe für */
{                       /* die jeweilige Ereignisklasse */
  char prchk;
  char timer;
  char opint;
  char error;
  char runout;
```

```
  char brkpt;
  char abend;
  char pterm;
  char rtimer;
  char filler;
};

struct stx              /* Steuerung des cstxit-Aufrufs für */
{                       /* die jeweilige Ereignisklasse */
  stx_set prchk;
  stx_set timer;
  stx_set opint;
  stx_set error;
  stx_set runout;
  stx_set brkpt;
  stx_set abend;
  stx_set pterm;
  stx_set rtimer;
  stx_set filler;
};

struct diag             /* Diagnosesteuerung für die */
{                       /* jeweilige Ereignisklasse */
  diag_set prchk;
  diag_set timer;
  diag_set opint;
  diag_set error;
  diag_set runout;
  diag_set brkpt;
  diag_set abend;
  diag_set pterm;
  diag_set rtimer;
  diag_set filler;
};

struct type             /* Parameterübergabe-Modus für */
{                       /* jeweilige Ereignisklasse */
  type_set prchk;
  type_set timer;
  type_set opint;
  type_set error;
  type_set runout;
  type_set brkpt;
  type_set abend;
  type_set pterm;
  type_set rtimer;
  type_set filler;
};
```

```
#define stx_set      char
#define old_stx      0
#define new_stx      4
#define del_stx      8

#define diag_set     char
#define ful_diag     0
#define min_diag     4
#define no_diag      8

#define err_set      char
#define no_err       0
#define par_err      4
#define stx_err      8
#define mem_err      12


#define type_set     char
#define par_opt      0
#define par_std      4
```

Steuerung des `cstxit`-Aufrufs:

Über diese Information wird der Ablauf des `cstxit`-Aufrufs gesteuert. Es wird festgelegt, welche Aktionen für die jeweilige Ereignisklasse durchgeführt werden.

`old_stx` Für die entsprechende Ereignisklasse ergibt sich keine Änderung. Eine vorher zugeordnete STXIT-Routine bleibt erhalten. Die restlichen Informationen für diese Ereignisklasse werden nicht ausgewertet.

`new_stx` Für die entsprechende Ereignisklasse wird eine neue STXIT-Routine zugeordnet. In diesem Fall werden die restlichen Informationen für diese Ereignisklasse ausgewertet. Insbesondere muss die Adresse der Routine im entsprechenden Eintrag von `contp` stehen.

`del_stx` Für die entsprechende Ereignisklasse wird die bisher zugeordnete STXIT-Routine gelöscht. Die restlichen Informationen für diese Ereignisklasse werden nicht ausgewertet.

Diagnosesteuerung:

`ful_diag`, `min_diag`, `no_diag` Die Parameter zur Diagnosesteuerung werden aus Kompatibilitätsgründen syntaktisch akzeptiert, jedoch wegen der Umstellung auf ILCS nicht mehr ausgewertet. Die angemeldete Routine wird ohne vorhergehende Diagnosemeldung aktiviert.

Parameterübergabe-Modus:

`par_opt` Die Parameter werden in den Registern 1-4 übergeben.

`par_std` Die Parameter werden in einer Parameterliste übergeben.
Für C ist nur dieser Wert zulässig.

Returncode:

`no_err` Die STXIT-Routine wurde ordnungsgemäß definiert.

`par_err` Die Parameterstruktur *stxitpar* wurde falsch versorgt.

`stx_err` Fehler beim Anmelden der STXIT-Routine.

`mem_err` Fehler bei der Speicherplatzanforderung (beim Anmelden der STXIT-Routine).

Hinweis Die Parameterstruktur *stxitpar* müssen Sie selbst versorgen.

Für die standardmäßige Initialisierung steht ein in der Include-Datei `stxit.h` definierter Prototyp (`stxit_pr`) zur Verfügung. Diesen Prototyp können Sie auf eine selbstdefinierte Struktur vom Typ `stxitp` kopieren und brauchen dann nur die Felder für diejenigen Ereignisklassen zu versorgen, bei denen die Zuordnung einer STXIT-Routine geändert werden soll.

Bei der Ereignisklasse `INTR` ist die Adresse zu versorgen (*stxitpar*`.`*bufadr*), bei der die Mitteilung an das Programm bereitgestellt werden soll. Die STXIT-Contingency-Routine kann dann die Mitteilung von dieser Adresse abholen und auswerten.

Siehe auch `alarm()`, `cenaco()`, `raise()`, `signal()`, `sleep()`.

ctermid - Pfadname für steuerndes Terminal erzeugen

Syntax

```
#include <stdio.h>

char *ctermid(char *s);
```

Beschreibung

`ctermid()` erzeugt eine Zeichenkette, die auf das aktuelle steuernde Terminal des aktuellen Prozesses verweist, wenn sie als Pfadname verwendet wird.

Wenn *s* ein Nullzeiger ist, wird die Zeichenkette in einem internen statischen Bereich gespeichert, dessen Inhalt beim nächsten Aufruf von `ctermid()` überschrieben und dessen Adresse zurückgegeben wird. Andernfalls wird angenommen, dass *s* auf einen Zeichenvektor mit wenigstens `L_ctermid` Elementen zeigt; der Pfadname wird in dieses Feld geschrieben und der Wert von *s* wird zurückgegeben. Die Konstante `L_ctermid` ist in der Include-Datei `stdio.h` definiert.

Returnwert Abweichend vom XPG4 wird immer `/dev/tty` zurückgegeben.

Hinweis Der Unterschied zwischen `ctermid()` und `ttyname()` ist der, dass `ttyname()` als Argument einen Dateideskriptor benötigt und den Pfadnamen des Terminals liefert, das diesem Dateideskriptor zugeordnet ist, `ctermid()` aber eine Zeichenkette (wie z.B. `/dev/tty`) liefert, die auf das aktuelle steuernde Terminal verweist, wenn sie als Pfadname benutzt wird. Daher ist `ttyname()` nur von Nutzen, wenn der Prozess bereits wenigstens eine Datei für ein Terminal geöffnet hat.

Siehe auch `ttyname()`, `stdio.h`.

ctime - Datum und Uhrzeit in Zeichenkette umwandeln

Syntax

```
#include <time.h>

char *ctime(const time_t *clock);
```

Beschreibung

`ctime()` wandelt die mit *clock* angegebene Zeit in eine lokale Zeitangabe um. Die Funktion gibt einen Zeiger auf eine 26 Zeichen lange Zeichenkette zurück (siehe Returnwert).

Mit *clock* gibt man die Zeit in Sekunden seit 00:00:00 UTC (Universal Time Coordinated, 1. Januar 1970) an.

Ein Aufruf von `ctime()` hat die gleiche Wirkung wie `asctime(localtime(`*clock*`))`.

`ctime()` ist nicht threadsicher. Verwenden Sie bei Bedarf die reentrante Funktion `ctime_r()`.

BS2000

`ctime()` interpretiert *clock* (siehe Returnwerte von `mktime()` und `time()`) als Anzahl der Sekunden, die seit dem 1. Januar 1950 00:00:00 vergangen sind, berechnet daraus die Ortszeit (MEZ) und wandelt das Ergebnis in eine EBCDIC-Zeichenkette um. ❑

Returnwert

Zeiger auf eine Zeichenkette bei Erfolg. Die Ergebniszeichenkette hat die Länge 26 (einschließlich Nullbyte) und das Format einer Datumsangabe mit Uhrzeit in Englisch:

wochentag monat tag std:min:sek jahr

z.B. `Wed Dec 14 15:20:54 1988\n\0`

Hinweis

Die Returnwerte für `ctime()` zeigen auf statische Daten, die bei jedem Aufruf überschrieben werden.

Siehe auch

`altzone`, `asctime()`, `ctime_r()`, `daylight`, `gmtime()`, `localtime()`, `timezone`, `tzname`, `tzset()`, `time.h`.

ctime_r - Datum und Uhrzeit threadsicher in Zeichenkette umwandeln

Syntax #include <time.h>

char *ctime_r(const time_t **clock*, char **buf*);

Beschreibung

`ctime_r()` wandelt den Zeitwert, auf den *clock* zeigt, in genau dieselbe Zeitform wie `ctime()` um und schreibt das Ergebnis in den Datenbereich, auf den *buf* zeigt (mit zumindest 26 Bytes).

Returnwert Zeiger auf die Zeichenkette, auf die *buf* zeigt, bei Erfolg.

Nullzeiger bei Fehler. (`errno` wird gesetzt, um den Fehler anzuzeigen.)

Siehe auch `asctime(), asctime_r(), ctime(), localtime(), localtime_r(), time()`.

cuserid - Benutzerkennung ermitteln

Syntax

```
#include <stdio.h>

char *cuserid(char *s);
```

Beschreibung

`cuserid()` erzeugt eine Zeichenkettendarstellung des Namens, der der realen Benutzernummer des aktuellen Prozesses zugeordnet ist (Benutzerkennung).

Wenn *s* der Nullzeiger ist, wird diese Zeichenkette in einem Bereich erzeugt, der statisch sein und daher durch nachfolgende Aufrufe von `cuserid()` überschrieben werden kann. Die Adresse dieses Bereichs wird zurückgeliefert.

Wenn *s* nicht der Nullzeiger ist, wird vorausgesetzt, dass *s* auf einen Vektor von mindestens `{L_cuserid}` Bytes zeigt. Die Zeichenkette der Benutzerkennung wird in diesem Vektor abgelegt. Die symbolische Konstante `{L_cuserid}` ist in `stdio.h` definiert und besitzt einen Wert größer als 0.

`cuserid()` ist nicht threadsicher.

Returnwert

s wenn *s* kein Nullzeiger ist. Wenn die Benutzerkennung nicht gefunden werden kann, wird **s* das Nullbyte zugewiesen.

Adresse des Puffers, der die Benutzerkennung enthält
wenn *s* der Nullzeiger ist und die Benutzerkennung nicht gefunden werden kann.

Nullzeiger wenn *s* der Nullzeiger ist und die Benutzerkennung nicht gefunden werden kann.

Hinweis

Die Funktionalität von `cuserid()`, die im POSIX.1-1988-Standard und XPG3 definiert wurde, unterscheidet sich von vorhergehenden Implementierungen und XPG3. Aus ISO POSIX-1-Standard wurde die `cuserid`-Funktion entfernt. Gemäß XPG4 sind beide Funktionalitäten noch erlaubt, aber gleichzeitig wird darauf hingewiesen, dass sie **zukünftig nicht mehr unterstützt** werden.

Die XPG2-Funktionalität kann mit folgender Syntax erreicht werden:

```
getpwuid(getuid())
```

Die XPG3-Funktionalität kann mit folgender Syntax erreicht werden:

```
getpwuid(geteuid())
```

Siehe auch `getlogin()`, `getpwnam()`, `getpwuid()`, `getuid()`, `geteuid()`, `stdio.h`, Handbuch „POSIX Grundlagen (BS2000/OSD)“.

__DATE__ - Makro für Übersetzungsdatum

Syntax __DATE__

Beschreibung

Dieses Makro generiert das Übersetzungsdatum einer Quelldatei als Zeichenkette in der Form: "*dd Mmm yyyy*\0"

Dabei bedeuten:

dd Tag (bei Tagen < 10 ohne führende Null)

Mmm Monatsname in Englisch (Abkürzung wie bei `asctime()`)

yyyy Jahr

Hinweis Dieses Makro muss in keiner Include-Datei definiert werden. Sein Name wird vom Compiler erkannt und ersetzt.

Siehe auch `asctime()`, `__TIME__`.

daylight - Sommerzeitvariable

Syntax #include <time.h>

extern int daylight;

Beschreibung

Die externe Variable `daylight` zeigt an, ob die Sommerzeit ausgegeben werden soll. `daylight` ist ungleich null, wenn eine alternative Zeitzone existiert. Die Zeitzonennamen sind in der externen Variablen `tzname` enthalten, die standardmäßig wie folgt gesetzt wird.

```
char *tzname[2] = { "GMT", "     " };
```

Die Funktionen `ctime()`, `localtime()`, `gmtime()` und `asctime()` kennen die Eigenarten dieser Konvertierungen für verschiedene Zeitperioden in den Vereinigten Staaten (insbesondere in den Jahren 1974, 1975 und 1987). Sie behandeln die neue Sommerzeit, beginnend mit dem ersten Sonntag im April 1987.

Hinweis Der Systemverwalter muss das Start- und Enddatum der Sommerzeit jährlich ändern, wenn das Format des Julianischen Kalenders verwendet wird.

Siehe auch `altzone`, `asctime()`, `ctime()`, `gmtime()`, `localtime()`, `timezone`, `tzname`, `tzset()`, `time.h`.

dbm_clearerr, dbm_close, dbm_delete, dbm_error, dbm_fetch, dbm_firstkey, dbm_nextkey, dbm_open, dbm_store - Funktionen zur Verwaltung von dbm-Datenbasen

Syntax

```
#include <ndbm.h>

int dbm_clearerr(DBM *db);

void dbm_close(DBM *db);

int dbm_delete(DBM *db, datum key);

int dbm_error(DBM *db);

datum dbm_fetch(DBM *db, datum key);

datum dbm_firstkey(DBM *db);

datum dbm_nextkey(DBM *db);

DBM *dbm_open(const char *file, int open_flags, mode_t file_mode);

int dbm_store(DBM *db, datum key, datum content, int store_mode);
```

Beschreibung

Diese Funktionen verwalten Paare aus Schlüssel und zugehörigem Inhalt (*key*/*content*) von mindestens 1024 Byte in einer Datenbasis. Die Funktionen bearbeiten sehr große Datenbasen (mit einer Milliarde Blöcken) und greifen auf ein mit einem Schlüssel versehenes Objekt in einem oder zwei Zugriff(en) auf das Dateisystem zu. Dieses Paket ersetzt die frühere `dbm`-Bibliothek, die nur jeweils eine Datenbasis verwalten kann.

key und *content* werden von der Typdefinition (`typedef`) *datum* beschrieben. *datum* gibt eine Zeichenkette von *dsize* Byte an, auf die *dptr* zeigt. Sowohl beliebige binäre Daten als auch normale ASCII-Zeichenketten sind zulässig.

Die Datenbasis wird in zwei Dateien gespeichert. Bei einer Datei handelt es sich um ein Verzeichnis mit dem Suffix `.dir`, das eine Bitmaske enthält. Die zweite Datei mit dem Suffix `.pag` enthält die Daten.

`dbm_open()` öffnet eine Datenbasis. Das Argument *file* muss den Pfadnamen der Datenbank enthalten. Hierdurch werden die Dateien *file*`.dir` und *file*`.pag` geöffnet und/oder erstellt, abhängig vom Argument *open_flags*. Die Bedeutung von *open_flags* entspricht der von *oflag* der Funktion `open()` (siehe Seite 633), außer dass bei den Dateien der Datenbank, die `WRITE-ONLY` geöffnet werden, Schreib- **und** Lesezugriff erlaubt ist. *file_mode* hat dieselbe Bedeutung wie das dritte Argument von `open()`.
`dbm_open()` gibt einen Zeiger auf eine Struktur vom Typ `DBM` zurück. Dieser Zeiger muss von allen übrigen Funktionen dieser Gruppe als Argument *db* übergeben werden.

`dbm_close()` schließt eine Datenbasis.

`dbm_fetch()` liest einen Satz aus der Datenbasis. *key* ist vom Typ `datum` und muss den Wert des entsprechenden Schlüssels des Satzes, der gelesen werden soll, enthalten.

`dbm_store()` schreibt einen Satz in die Datenbasis. *key* ist vom Typ `datum` und muss den Wert des entsprechenden Schlüssels des Satzes, der geschrieben werden soll, enthalten. Unter diesem Schlüssel kann der Satz später wieder gelesen, geändert oder gelöscht werden. *content* ist ebenfalls vom Typ `datum` und enthält den Inhalt des Satzes, der geschrieben werden soll. Das Argument *store_mode* kann entweder `DBM_INSERT` oder `DBM_REPLACE` lauten. Bei `DBM_INSERT` werden nur neue Einträge in die Datenbasis aufgenommen; ein bereits vorhandener Eintrag mit gleichem Schlüssel wird nicht geändert. Bei `DBM_REPLACE` wird ein bestehender Eintrag ersetzt, wenn er den gleichen Schlüssel hat. Bei `DBM_INSERT` dagegen wird ein bestehender Eintrag mit gleichem Schlüssel nicht ersetzt. Wenn der angegebene Schlüssel in der Datenbasis nicht gefunden wird, fügt `dbm_store()` den Satz in die Datenbasis ein, unabhängig davon, ob *store_mode* auf `DBM_INSERT` oder `DBM_REPLACE` gesetzt ist.

`dbm_delete()` löscht einen Satz und den zugehörigen Schlüssel aus der Datenbasis. *key* ist vom Typ `datum` und muss den Wert des entsprechenden Schlüssels des Satzes, der gelöscht werden soll, enthalten.

`dbm_firstkey()` gibt den ersten Schlüssel in der Datenbasis zurück.

`dbm_nextkey()` gibt den jeweils nächsten Schlüssel in der Datenbasis zurück. Um mit `dbm_nextkey()` arbeiten zu können, muss zuvor `dbm_firstkey()` aufgerufen worden sein. Aufeinander folgende Aufrufe von `dbm_nextkey()` geben jeweils den nächsten Schlüssel zurück, bis alle Schlüssel der Datenbasis abgearbeitet sind.

Die Funktion `dbm_error()` gibt die Fehlerbedingung der Datenbank zurück. Das Argument *db* ist ein Zeiger auf eine Datenbankstruktur, die von einem Aufruf von `dbm_open()` zurückgegeben wurde.

Die Funktion `dbm_clearerr()` löscht die Fehlerbedingung der Datenbank. Das Argument *db* ist ein Zeiger auf eine Datenbankstruktur, die von einem Aufruf von `dbm_open()` zurückgegeben wurde.

`dbm_clearerr()` ist nicht threadsicher.

Returnwert `dbm_open()`:

Zeiger auf eine Struktur vom Typ `DBM`
bei Erfolg.
(DBM *)0 bei Fehler.

`dbm_store()`:

0	bei Erfolg.
1	falls *flags* den Wert `DBM_INSERT` hat und die Datenbasis bereits einen Satz mit dem angegebenen Schlüssel enthält.
Negativwert	bei Fehler.

`dbm_fetch()`:

`datum content`	bei Erfolg.
`dptr` = Nullzeiger	falls der angegebene Schlüssel nicht in der Datenbasis gefunden wurde oder bei Fehler.

`dbm_delete()`:

0	bei Erfolg.
Negativwert	bei Fehler.

`dbm_firstkey()`, `dbm_nextkey()`:

`datum key`	bei Erfolg.
`dptr` = Nullzeiger	falls das Ende der Datenbasis erreicht ist oder bei Fehler. Im Fehlerfall wird zusätzlich die Fehleranzeige der Datenbasis gesetzt.

`dbm_error()`:

0	wenn die Fehlerbedingung nicht gesetzt ist.
≠ 0	wenn die Fehlerbedingung gesetzt ist.

`dbm_clearerr()`:

Der Returnwert ist undefiniert.

Hinweis Der folgende Code geht die gesamte Datenbasis durch:

```
for (key = dbm_firstkey(db); key.dptr != NULL; key = dbm_nextkey(db))
```

Die `dbm_` Funktionen, die in dieser Bibliothek zur Verfügung gestellt werden, können keinesfalls mit den Funktionen eines allgemeinen Datenbankverwaltungssystems verglichen werden. Sie ermöglichen keine mehrfachen Suchschlüsselworte pro Eintrag, sie bieten keinen Schutz gegen Mehrfach-Zugriff (d.h. sie sperren keine Sätze oder Dateien) und sie stellen auch nicht die Vielzahl anderer Datenbankfunktionen bereit, die in leistungsstarken

Datenbankverwaltungssystemen angeboten werden. Erstellen und Aktualisieren von Datenbasen mit diesen Funktionen geschieht auf Grund der Datenkopien nach Hash-Kollisionen relativ langsam. Die `dbm_` Funktionen sind nützlich für Anwendungen, die ohne viel Aufwand relativ statische Informationen verwalten wollen, die über einen einzigen Schlüssel indiziert werden.

Die *dptr*-Zeiger, die von diesen Funktionen zurückgegeben werden, zeigen auf statischen Speicher, der durch nachfolgende Aufrufe geändert werden kann.

`dbm_delete()` stellt den Dateibereich zwar nicht physisch wieder her, macht ihn aber zur weiteren Verwendung verfügbar.

Wird die Datenbasis durch Aufrufe von `dbm_store()` oder `dbm_delete()` verändert, während die Datenbasis sequenziell mit den Funktionen `dbm_firstkey()` und `dbm_nextkey()` durchgegangen wurde, so empfiehlt es sich, mit einem Aufruf von `dbm_firstkey()` auf den Anfang der Datenbasis zurückzusetzen.

Siehe auch `open()`, `ndbm.h`.

difftime - Differenz zwischen zwei Kalenderdaten berechnen

Syntax #include <time.h>

double difftime(time_t *time1*, time_t *time0*);

Beschreibung

`difftime()` berechnet die Differenz zwischen zwei Kalenderdaten.

time1 und *time0* sind Zeitwerte vom Typ `time_t`. Diese Zeitwerte werden von den Funktionen `mktime()` und `time()` geliefert.

Returnwert *time1* - *time0* bei Erfolg. Differenz, angegeben in Sekunden, vom Typ `double`.

Siehe auch `ctime()`, `mktime()`, `time()`, `time.h`.

dirname - Vaterverzeichnis zu einem Pfadnamen liefern

Syntax

```
#include <libgen.h>

char *dirname(char *path);
```

Beschreibung

`dirname()` ermittelt zu dem Pfadnamen, auf den **path* zeigt, das übergeordnete Verzeichnis und liefert einen Zeiger auf eine Zeichenkette zurück, die den Namen dieses Vaterverzeichnisses bzw. den String "." enthält. Dabei werden abschließende Gegenstriche (‚/') am Ende des Pfadnamens nicht als Teil des Pfades gewertet.

Enthält *path* keinen Gegenschrägstrich (‚/'), liefert `dirname()` einen Zeiger auf die Zeichenkette "." zurück.

Wenn *path* ein Nullzeiger ist oder auf einen leeren String zeigt, gibt `dirname()` ebenfalls einen Zeiger auf die Zeichenkette "." zurück.

`dirname()` ist nicht reentrant.

Returnwert Zeiger auf den Namen des Vaterverzeichnisses oder

Zeiger auf Zeichenkette "."

Wenn *path* keinen Gegenschrägstrich enthält,
path ein Nullzeiger ist oder auf einen leeren String zeigt.

Beispiel

Eingabewert in *path*	Rückgabewert
`"/usr/lib"`	`"/usr"`
`"/usr/"`	`"/"`
`"usr"`	`"."`
`"/"`	`"/"`
`"."`	`"."`
`".."`	`"."`

Der folgende Code-Fragment liest einen Pfadnamen, macht das Vaterverzeichnis zum aktuellen Arbeitsverzeichnis und öffnet die Datei:

```
char path(MAXPATHLEN), *pathcopy;
int fd;
fgets(path, MAXPATHLEN, stdin);
pathcopy = strup(path);
chdir(dirname(pathcopy));
fd = open(basename(path), O_RDONLY);
```

Hinweis `dirname()` kann die Zeichenkette *path* verändern. Der Returnwert von `dirname()` kann in einen statischen Bereich zeigen, der von einem nachfolgenden Aufruf von `dirname()` überschrieben wird.

`dirname()` und `basename()` ergeben zusammen einen vollständigen Pfadnamen. `dirname(path)` ermittelt den Pfadnamen des Verzeichnisses, in dem sich `basename(path)` befindet.

Siehe auch `basename()`, `libgen.h`.

div - ganze Zahl dividieren

Syntax #include <stdlib.h>

div_t div(int *numer*, int *denom*);

Beschreibung

`div()` berechnet den Quotienten und den Rest der Division *numer* / *denom*.

Das Vorzeichen des Quotienten ist gleich dem Vorzeichen des algebraischen Quotienten. Die Größe des Quotienten ist die größte ganze Zahl, die kleiner oder gleich dem absoluten Wert des algebraischen Quotienten ist.

Der Rest ergibt sich aus der Gleichung:

quotient * *divisor* + *rest* = *dividend*

Returnwert Struktur vom Typ `div_t`
bei Erfolg. Die Struktur enthält sowohl den Quotienten `quot` als auch den Rest `rem` als ganzzahlige Werte.

Siehe auch `ldiv()`, `stdlib.h`.

drand48 - Pseudo-Zufallszahlen zwischen 0.0 und 1.0 generieren

Syntax

```
#include <stdlib.h>

double drand48 (void);
double erand48 (unsigned short int xsubi[3]);
long int jrand48 (unsigned short int xsubi[3]);
void lcong48 (unsigned short int param[7]);
long int lrand48 (void);
long int mrand48 (void);
long int nrand48 (unsigned short int xsubi[3]);
unsigned short int *seed48 (unsigned short int seed16v[3]);
void srand48 (long int seedval);
```

Beschreibung

Diese Familie von Funktionen erzeugt Pseudo-Zufallszahlen unter Verwendung eines Algorithmus der linearen Kongruenz und von ganzzahliger 48-Bit-Arithmetik.

`drand48()` und `erand48()` liefern nichtnegative Gleitpunktzahlen doppelter Genauigkeit, die gleichmäßig über das Intervall [0.0, 1.0] verteilt sind.

`lrand48()` und `nrand48()` liefern nichtnegative Ganzzahlen vom Typ `long`, die gleichmäßig über das Intervall $[0, 2^{31}]$ verteilt sind.

`mrand48()` und `jrand48()` liefern vorzeichenbehaftete Ganzzahlen vom Typ `long`, die gleichmäßig über das Intervall $[-2^{31}, 2^{31}]$ verteilt sind.

`srand48()`, `seed48()` und `lcong48()` sind Initialisierungsprozeduren, von denen eine vor dem Aufruf von entweder `drand48()`, `lrand48()` oder `mrand48()` aufgerufen werden sollte. Obwohl nicht empfohlen wird `drand48()`, `lrand48()` oder `mrand48()` ohne vorhergehenden Initialisierungsaufruf aufzurufen, werden für diesen Fall automatisch voreingestellte Anfangswerte zur Verfügung gestellt.

`erand48()`, `nrand48()` und `rand48()` benötigen keinen vorausgehenden Initialisierungsaufruf.

Alle Prozeduren arbeiten mit einer Sequenz von ganzzahligen 48-Bit Werten Xi, die der linear kongruenten Formel entsprechend gebildet werden:

$$X_{n+1} = (aX_n + c) \bmod m \qquad n \geq 0$$

Für den Parameter `m` gilt: $m = 2^{48}$; es wird 48-Bit Ganzzahlarithmetik betrieben. Sofern nicht `lcong48()` aufgerufen wurde, ist der Wert des Faktors a und der additiven Konstanten c gegeben durch:

$$a = 5DEECE66D_{16} = 273673163155_8$$
$$c = B_{16} = 13_8$$

Jedes Ergebnis einer der Funktionen `drand48()`, `erand48()`, `lrand48()`, `nrand48()`, `mrand48()` oder `jrand48()` entsteht folgendermaßen: Zuerst wird das nächste 48-Bit Reihenelement *Xi* berechnet. Dann werden, je nach Datentyp der Rückgabevariablen, entsprechend viele Bits aus dem höchstwertigen Anteil von *Xi* (äußere linke Bits) kopiert und in das Ergebnis umgewandelt.

Die Funktionen `drand48()`, `lrand48()` und `mrand48()` speichern den zuletzt erzeugten 48-Bit Wert *Xi* in einem internen Puffer; dies ist auch der Grund, weshalb sie vor dem ersten Aufruf initialisiert werden müssen. Die Funktionen `erand48()`, `nrand48()` und `jrand48()` erwarten vom aufrufenden Programm die Bereitstellung von Speicherplatz für die sukzessiven Werte *Xi* in Form eines Vektors, der beim Aufruf als Parameter übergeben wird. Deswegen brauchen diese Funktionen nicht initialisiert zu werden; das aufrufende Programm muss lediglich den benötigten Anfangswert von *Xi* in dem Vektor ablegen und diesen als Argument übergeben.

Durch die Verwendung verschiedener Argumente erlauben es die Funktionen `erand48()`, `nrand48()` und `jrand48()`, in getrennten Modulen größerer Programme mehrere unabhängige Folgen von Pseudo-Zufallszahlen zu generieren, d.h., die Reihenfolge der Zahlen in einer Folge ist nicht abhängig davon, wie oft die Routinen für die Generierung von Zahlen in anderen Folgen aufgerufen wurden.

Die Initialisierungsfunktion `srand48()` setzt die höherwertigen 32 Bit von *Xi* auf den Wert der `{LONG_BIT}` Bits ihres Arguments. Die niederwertigen 16 Bit von *Xi* werden mit dem willkürlichen Wert $330E_{16}$ belegt.

Die Initialisierungsfunktion `seed48()` setzt den Wert von *Xi* auf den 48-Bit Wert, der im übergebenen Vektor angegeben wird. Zusätzlich wird der frühere Wert von *Xi* in einem internen 48-Bit Puffer abgespeichert, der nur von `seed48()` verwendet und dessen Adresse von `seed48()` zurückgegeben wird. Dieser zurückgegebene Zeiger kann ignoriert werden, wenn er nicht benötigt wird. Er ist jedoch nützlich, falls ein Programm zu irgendeinem späteren Zeitpunkt von einer gegebenen Stelle aus neu gestartet werden soll. Es kann den Zeiger dazu benutzen, den letzten Wert von *Xi* zu ermitteln und abzuspeichern, um dann durch `seed48()` diesen Wert für eine Reinitialisierung beim Neustart zu verwenden.

Die Initialisierungsfunktion `lcong48()` erlaubt dem Benutzer, die Voreinstellungswerte von *Xi*, des Faktors *a* und der additiven Konstanten *c* festzulegen. Die Vektorelemente *param*`[0]` bis *param*`[2]` legen *Xi* fest, *param*`[3]` bis *param*`[5]` legen den Faktor *a* und *param*`[6]` legt die 16-Bit Additionskonstante *c* fest. Nach dem Aufruf von `lcong48()` wird ein nachfolgender Aufruf von entweder `srand48()` oder `seed48()` diesen "Standard"-Faktor *a* und die additive Komponente *c* wiederherstellen, wie oben angegeben.

Returnwert Siehe oben im Abschnitt „Beschreibung“.

Siehe auch `rand()`, `stdlib.h`.

dup, dup2 - Dateideskriptor duplizieren

Syntax

#include <unistd.h>

int dup(int *fildes*);

int dup2(int *fildes*, int *fildes2*);

Beschreibung

fildes ist ein Dateideskriptor, der von einem Systemaufruf `creat()`, `open()`, `dup()`, `fcntl()` oder `pipe()` geliefert wurde. `dup()` gibt einen neuen Dateideskriptor zurück, der mit dem Original-Dateideskriptor Folgendes gemeinsam hat:

– dieselbe offene Datei oder Pipe

– denselben Lese-/Schreibzeiger

– denselben Zugriffsmodus (Lesen, Schreiben oder Schreiben/Lesen)

fildes2 ist eine nichtnegative ganze Zahl, die kleiner als `{OPEN-MAX}` ist. `dup2()` veranlasst *fildes2*, auf dieselbe Datei wie *fildes* zu verweisen. Wenn *fildes2* bereits auf eine offene Datei verweist, außer auf *fildes*, wird die offene Datei erst geschlossen. Wenn *fildes2* auf *fildes* verweist oder wenn *fildes* kein gültiger Dateideskriptor ist, wird *fildes* nicht zuerst geschlossen.

Die Funktionen `dup()` und `dup2()` bieten eine alternative Schnittstelle der Funktion `fcntl()` zu dem Kommando `F_DUPFD`. Der Aufruf

```
fid = dup (fildes);
```

ist äquivalent zu:

```
fid = fcntl (fildes, F_DUPFD, 0);
```

Der Aufruf

```
fid = dup2 (fildes, fildes2);
```

ist äquivalent zu:

```
close (fildes2);
fid = fcntl (fildes, F_DUPFD, fildes2);
```

mit folgender Ausnahme:

Ist *fildes* ein gültiger Dateideskriptor und gleich *fildes2*, dann liefert `dup2()` *fildes2* ohne *fildes2* zu schließen.

Returnwert nichtnegative Zahl (Dateideskriptor)
bei Erfolg.

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

Fehler `dup()` und `dup2()` schlagen fehl, wenn gilt:

`EBADF` *fildes* ist kein gültiger Dateideskriptor, oder *fildes2* ist negativ oder größer oder gleich `{OPEN_MAX}`.

`EINTR` `dup2()` wurde durch ein Signal unterbrochen.

Erweiterung
`EINVAL` *fildes* und *fildes2* bezeichnen BS2000-Dateien. ❑

`EMFILE` Die Anzahl der durch den Prozess verwendeten Dateideskriptoren überschreitet `{OPEN_MAX}`, oder es sind keine Dateideskriptoren *fildes2* verfügbar.

Hinweis `dup()` und `dup2()` werden nur für POSIX-Dateien ausgeführt.

Siehe auch `close()`, `fcntl()`, `open()`, `unistd.h`.

ebcdic_to_ascii - EBCDIC- zu ASCII-Zeichenketten konvertieren

(Erweiterung)

Syntax int ebcdic_to_ascii(char **in*, char **out*);

Beschreibung
`ebcdic_to_ascii` konvertiert EBCDIC- zu ASCII-Zeichenketten. Dabei ist *in* die Eingabezeichenkette im EBCDIC-Code und *out* die Ausgabezeichenkette im ASCII-Code. Der Puffer muss vom Aufrufer zur Verfügung gestellt werden.

Die Zeichen der Eingabezeichenkette werden als EBCDIC-Zeichen interpretiert und in die entsprechenden Zeichen des ASCII-Code umgesetzt.

Returnwert 0 bei Erfolg.

1 bei Fehler.

Siehe auch `ascii_to_ebcdic`.

ecvt, fcvt, gcvt - Gleitpunktzahl in Zeichenkette umwandeln

Syntax

#include <stdlib.h>

char *ecvt(double *value*, int *ndigit*, int **decpt*, int **sign*);

char *fcvt (double *value*, int *ndigit*, int **decpt*, int **sign*);

char *gcvt (double *value*, int *ndigit*, char **buf*);

Beschreibung

`ecvt()` wandelt einen Gleitpunktwert *value* in eine Zeichenkette aus *ndigit* EBCDIC-Ziffern um und liefert als Ergebnis einen Zeiger auf diese Zeichenkette. Das Ausgabeformat entspricht dem `%f`-Format von `printf()`.

Die Zeichenkette beginnt mit der ersten Ziffer ungleich 0 aus dem umzuwandelnden Gleitpunktwert, d.h. führende Nullen werden nicht übernommen.

Dezimalzeichen und ein ggf. negatives Vorzeichen sind nicht Bestandteil der Zeichenkette. `ecvt()` liefert jedoch die Position des Dezimalzeichens und das Vorzeichen in Ergebnisparametern zurück.

value ist ein Gleitpunktwert, der für die Ausgabe aufbereitet werden soll.

ndigit ist die Anzahl der Ziffern in der Ergebniszeichenkette (gerechnet ab der ersten Ziffer ungleich 0 aus dem umzuwandelnden Gleitpunktwert). Ist *ndigit* kleiner als die Ziffernzahl von *value*, wird die niedrigste Stelle gerundet. Ist *ndigit* größer, wird rechtsbündig mit Nullen aufgefüllt. Die Genauigkeit der umgewandelten Zahl wird begrenzt durch die maximale Anzahl signifikanter Ziffern, die im Typ `double` darstellbar sind.

decpt ist der Zeiger auf eine ganze Zahl, die die Position des Dezimalzeichens in der Ergebniszeichenkette angibt. Wenn **decpt* eine positive Zahl ist, wird die Position des Dezimalzeichens relativ zum Beginn der Ergebniszeichenkette angegeben. Wenn **decpt* eine negative Zahl bzw. 0 ist, steht das Dezimalzeichen links vor der ersten Ziffer. Kann der ganz-zahlige Anteil von *value* nicht vollständig mit *ndigit* Ziffern dargestellt werden, ist **decpt* größer als *ndigit*.

sign ist der Zeiger auf eine ganze Zahl, die das Vorzeichen der Ergebniszeichenkette angibt. Wenn **sign* gleich 0 ist, ist das Vorzeichen positiv. Wenn **sign* ungleich 0 ist, ist das Vorzeichen negativ.

`fcvt()` ist identisch zu `ecvt()`, außer dass mit *ndigit* die Anzahl der Ziffern nach dem Dezimalzeichen angegeben wird.

Ist *ndigit* kleiner als die Ziffernzahl von *value* nach dem Dezimalzeichen, wird die niedrigste Stelle gerundet. Ist *ndigit* größer, wird rechtsbündig mit Nullen aufgefüllt.

`gcvt()` wandelt einen Gleitpunktwert *value* in eine Zeichenkette aus EBCDIC-Ziffern entsprechend dem `%g`-Format von `printf()` um und schreibt die aufbereitete Zeichenkette in einen Vektor, auf den *buf* zeigt. Als Ergebnis wird ein Zeiger auf diesen Bereich geliefert. Es werden *ndigit* signifikante Ziffern erzeugt (obere Grenze für *ndigit* ist die Anzahl signifikanter Stellen, die der Genauigkeit des Typs `double` entspricht). Ist *ndigit* kleiner als die Ziffernzahl von *value*, wird die niedrigste Stelle gerundet. Ist *ndigit* größer, endet die Zeichenkette mit der letzten Ziffer ungleich 0. Falls *value* eine ganze Zahl darstellt, wird *buf* rechtsbündig mit Nullen aufgefüllt.

Außerdem enthält die Zeichenkette das Minuszeichen, falls der Wert < 0 ist, und das Dezimalzeichen, falls *value* keine ganze Zahl ist. Das verwendete Dezimalzeichen ergibt sich aus der aktuellen Lokalität und wird dort durch die Kategorie `LC_NUMERIC` bestimmt. Wenn durch `setlocale()` nicht explizit die Lokalität geändert wurde, gilt der Standardwert „POSIX“. In der POSIX-Lokalität ist das Dezimalzeichen ein Punkt (.).

Je nach Aufbau des umzuwandelnden Gleitpunktwertes entspricht das Ausgabeformat

– dem `%f`-Format von `printf()`.

– oder dem `%e`-Format von `printf()` (Exponential-Schreibweise / wissenschaftliche Notation).

ndigit ist die Anzahl der Ziffern in der Ergebniszeichenkette (gerechnet ab der ersten Ziffer ungleich 0 aus dem umzuwandelnden Gleitpunktwert).

**buf* ist der Zeiger auf die umgewandelte Zeichenkette.
Der Speicherbereich, auf den *buf* zeigt, sollte mindestens (*ndigit* + 4) Bytes groß sein!

`ecvt()`, `fcvt()` und `gcvt()` sind nicht threadsicher.

Returnwert `ecvt()`, `fcvt()`:
Zeiger auf die umgewandelte EBCDIC-Zeichenkette
bei Erfolg. Die Zeichenkette wird mit dem Nullbyte (`\0`) abgeschlossen.

`gcvt()`:
**buf* bei Erfolg. Die Zeichenkette wird mit dem Nullbyte (`\0`) abgeschlossen.

Hinweis Falsche Parameter, etwa ein `integer`- statt `double`-Wert, führen zum Programmabbruch.

Portable Anwendungen sollten statt `ecvt()`, `fcvt()` und `gcvt()` die Funktion `sprintf()` verwenden.

`ecvt()` und `fcvt()`: Das Ergebnis wird in einem C-internen Datenbereich abgelegt, der bei jedem nachfolgenden Aufruf einer dieser Funktionen überschrieben wird.

Siehe auch `printf()`, `setlocale()`, `sprintf()`, `stdlib.h`.

_edt - EDT aufrufen *(BS2000)*

Syntax #include <stdlib.h>

void _edt(void);

Beschreibung

`_edt` ruft den BS2000-Dateibearbeiter EDT auf. Nach ordnungsgemäßer Beendigung des Dateibearbeiters fährt das Programm mit der nächsten C-Anweisung nach dem `_edt`-Aufruf fort.

Hinweis Programme, die `_edt` aufrufen, benötigen beim Ablauf Module aus der Modulbibliothek `EDTLIB` (standardmäßig auf der `$TSOS`-Kennung). Beim Binden ist eine `RESOLVE`-Anweisung auf diese Bibliothek abzusetzen.

encrypt - Zeichenkette blockweise verschlüsseln

Syntax #include <unistd.h>

void encrypt(char *block*[64], int *edflag*);

Beschreibung

`encrypt()` ermöglicht den Zugriff auf einen Verschlüsselungsalgorithmus. Der Schlüssel *key*, der von `setkey()` erzeugt worden ist, wird verwendet, um die Zeichenkette *block* mittels `encrypt()` zu verschlüsseln.

block ist ein Zeichenfeld der Länge 64, das nur Zeichen mit den Werten 0 und 1 enthält. Das Argumentfeld wird in ein ähnliches Feld geändert, das die Bits des Arguments enthält, nachdem die unter Verwendung des von `setkey()` gesetzten Schlüssels durch den Verschlüsselungsalgorithmus verändert wurden.
Wenn *edflag* null ist, wird das Argument verschlüsselt. Das Argument kann nicht entschlüsselt werden, falls dies versucht wird (*edflag* = 1), wird `errno` auf `ENOSYS` gesetzt.

Fehler `encrypt()` schlägt fehl, wenn gilt:

`ENOSYS` Das System unterstützt die Funktionalität nicht.

Hinweis Da `encrypt()` keinen Returnwert zurückgibt, können Fehler nur wie folgt festgestellt werden: `errno` wird auf 0 gesetzt; anschließend wird die Funktion aufgerufen und `errno` geprüft. Wenn `errno` ungleich 0 ist, muss ein Fehler aufgetreten sein.

Siehe auch `crypt()`, `setkey()`, `unistd.h`.

endgrent, getgrent, setgrent - Gruppenverwaltung

Syntax

```
#include <grp.h>

void endgrent (void);

void setgrent (void);

struct group *getgrent (void);
```

Beschreibung

`getgrent()` gibt einen Zeiger auf ein Objekt mit nachstehender Struktur zurück, die die einzelnen Felder einer Zeile der Datei `/etc/group` enthält. Jede Zeile enthält ein Objekt der Struktur `group` (Gruppen-Struktur), die in der Include-Datei `grp.h` deklariert ist, mit folgenden Elementen:

```
struct        group {
     char    *gr_name;       /* Name der Gruppe */
     char    *gr_passwd;     /* verschlüsseltes Gruppenpasswort */
     gid_t    gr_gid;        /* numerische Gruppennummer */
     char   **gr_mem;        /* Zeiger auf Namen der Gruppenmitglieder */
};
```

`getgrent()` gibt beim ersten Aufruf einen Zeiger auf die erste Gruppen-Struktur in der Datei zurück; danach gibt es einen Zeiger auf die nächste Gruppen-Struktur in der Datei zurück. Auf diese Weise können aufeinander folgende Aufrufe zum Absuchen der gesamten Datei verwendet werden.

`setgrent()` bewirkt das Zurücksetzen des Schreib-/Lesezeigers auf den Anfang der Gruppendatei und ermöglicht damit ein wiederholtes Suchen.

`endgrent()` kann am Ende der Verarbeitung aufgerufen werden, um die Gruppendatei zu schließen.

`endgrent()`, `getgrent()` und `setgrent()` sind nicht threadsicher.

Returnwert `getgrent()`:

Zeiger auf die erste Gruppen-Struktur der Gruppendatei
: beim ersten Aufruf

Zeiger auf die nächste Gruppen-Struktur der Gruppendatei
: bei nachfolgenden Aufrufen

Nullzeiger
: bei `EOF` oder Fehler. `errno` wird gesetzt um den Fehler anzuzeigen.

Fehler `getgrent()` schlägt fehl, wenn gilt:

`EINTR`
: `getgrent()` wurde durch ein Signal unterbrochen.

EIO	Während des Lesens oder Schreibens ist ein Ein-/Ausgabefehler aufgetreten.
EMFILE	Im aufrufenden Prozess sind {OPEN_MAX} Dateideskriptoren geöffnet.
ENFILE	Im System ist die maximal zulässige Anzahl von Dateien geöffnet.
Erweiterung ENOMEM	Der Speicherplatz reicht nicht aus, um die globalen Daten von getgrent() anzulegen. ❑

Hinweis Der Returnwert von getgrent() kann in einen Bereich zeigen, der von einem nachfolgenden Aufruf von getgrgid(), getgrnam() oder getgrent() überschrieben wird.

Diese Funktionen werden weiterhin angeboten, weil sie in der Vergangenheit gebräuchlich waren. Jedoch ist der Aufbau der Struktur group implementierungsabhängig, weswegen Anwendungen, die diese Funktionen einsetzen, nicht portabel sind. Portable Anwendungen sollten daher getgrnam() und getgrgid() verwenden.

Siehe auch getgrgid(), getgrnam(), getlogin(), getpwent(), grp.h.

endpwent, getpwent, setpwent - Benutzerkatalog verwalten

Syntax

```
#include <pwd.h>

void endpwent (void);

struct passwd *getpwent (void);

void setpwent (void);
```

Beschreibung

`getpwent()` gibt einen Zeiger auf ein Objekt mit nachstehender Struktur zurück, die die einzelnen Felder einer Zeile der Datei `/etc/passwd` enthält. Jede Zeile enthält ein Objekt der Struktur `passwd` (Kennwort-Struktur), die in der Include-Datei `pwd.h` deklariert ist, mit folgenden Elementen:

```
struct passwd {
      char    *pw_name;
      char    *pw_passwd;
      uid_t    pw_uid;
      gid_t    pw_gid;
      char    *pw_age;
      char    *pw_comment;
      char    *pw_gecos;
      char    *pw_dir;
      char    *pw_shell;
};
```

Die Komponenten dieser Struktur werden aus dem Benutzerkatalog seriell gelesen. `getpwent()` gibt beim ersten Aufruf einen Zeiger auf die erste Kennwort-Struktur im Benutzerkatalog zurück; danach gibt es einen Zeiger auf die nächste Kennwort-Struktur in der Datei zurück. Auf diese Weise können aufeinander folgende Aufrufe zum Absuchen des gesamten Benutzerkatalogs verwendet werden.

`setpwent()` löscht den Zeiger, mit dem der Benutzerkatalog durch `getpwent()` seriell durchsucht werden soll. Ein Darauf folgender `getpwent`-Aufruf gibt einen Zeiger auf die erste Kennwort-Struktur zurück.

`endpwent()` kann am Ende der Verarbeitung aufgerufen werden, um den Benutzerkatalog zu schließen.

`endpwent()`, `getpwent()` und `setpwent()` sind nicht threadsicher.

Returnwert `getpwent()`:

Zeiger auf eine Struktur vom Typ `passwd`
bei Erfolg.

Nullzeiger bei `EOF` und bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `endpwent()` schlägt fehl, wenn gilt:

`EACCES` Die Benutzernummer (uid) des Aufrufers ist ungültig.

`getpwent()`, `setpwent()` und `endpwent()` schlagen fehl, wenn gilt:

`EFAULT` Fehler beim Anlegen der `passwd`-Struktur.

`ENOENT` Benutzer existiert nicht.

Hinweis Der Returnwert von `getpwent()` kann in einen Bereich zeigen, der von einem nachfolgenden Aufruf von `getpwuid()`, `getpwnam()` oder `getpwent()` überschrieben wird.

Eine Kennwortdatei `/etc/passwd` gibt es im POSIX-Subsystem nicht. Die Benutzerdaten werden intern im Benutzerkatalog hinterlegt (siehe Handbuch „POSIX Grundlagen (BS2000/OSD)“).

Diese Funktionen werden nur noch aus Kompatibilitätsgründen unterstützt.

Die Merkmale eines aktuellen Prozesses können wie folgt festgestellt werden:

– `getpwuid(geteuid())` gibt den Namen der effektiven Benutzernummer des Prozesses zurück.
– `getlogin()` gibt die Benutzerkennung des Prozesses zurück.
– `getpwuid(getuid())` gibt den Namen der realen Benutzernummer des Prozesses zurück.

Wenn Fehlersituationen untersucht werden sollen, muss `errno` vor dem Aufruf von `getpwent()` auf 0 gesetzt werden.

Siehe auch `endgrent()`, `getlogin()`, `getpwnam()`, `getpwuid()`, `putpwent()`, `pwd.h`, Handbuch „POSIX Grundlagen (BS2000/OSD)“.

endutxent, getutxent, getutxid, getutxline, pututxline, setutxent, utmpx-Einträge verwalten

Syntax

```
#include <utmpx.h>

void endutxent (void);

struct utmpx *getutxent (void);

struct utmpx *getutxid (const struct utmpx *id);

struct utmpx *getutxline (const struct utmpx *line);

struct utmpx *pututxline (const struct utmpx *utmpx);

void setutxent (void);
```

Beschreibung

Diese Funktionen ermöglichen den Zugriff auf die Benutzer-Abrechnungsdatei (user accounting database) `/var/adm/utmpx`.

`getutxent()`, `getutxid()` und `getutxline()` liefern einen Zeiger auf eine Struktur des folgenden Typs:

```
struct     utmpx {
     char     ut_user[32];  /* Anmeldenamen des Benutzers */
     char     ut_id[4];     /* /sbin/inittab id (normalerweise Zeilennr) */
     char     ut_line[32];  /* Gerätename (Konsole, lnxx) */
     pid_t    ut_pid;       /* Prozessnummer */
     short    ut_type;      /* Art des Eintrags */
     struct     exit_status {
         short     e_termination;  /* Ende-Status */
         short     e_exit;         /* Exit-Status */
     } ut_exit;   /* Exit-Status eines Prozesses markiert als DEAD_PROCESS */
     struct timeval     ut_tv;     /* Zeiteintrag gemacht */
     short ut_syslen;              /* signifikante Länge von ut_host */
                                   /* einschließlich abschließender Null */
     char     ut_host[257];        /* Host-Name, falls gegeben */
};
```

`getutxent()` liest den nächsten Eintrag aus einer `utmpx`-ähnlichen Datei. Wenn die Datei noch nicht geöffnet ist, wird sie geöffnet. Wird das Dateiende erreicht, scheitert die Operation.

`getutxid()` sucht von der aktuellen Position in der Datei `utmpx` vorwärts, bis ein Eintrag gefunden wird, dessen *ut_type* dem *id->ut_type* entspricht, wenn der angegebene Typ `RUN_LVL`, `BOOT_TIME`, `OLD_TIME` oder `NEW_TIME` ist. Ist der in *id* angegebene Typ `INIT_PROCESS`, `LOGIN_PROCESS`, `USER_PROCESS` oder `DEAD_PROCESS`, dann liefert `getutxid()` einen Zeiger auf den ersten Eintrag, dessen Typ einem dieser vier Typen ent-

spricht und dessen Komponente *ut_id* dem Wert des übergebenen *id->ut_id* entspricht. Wird das Dateiende erreicht, ohne dass ein entsprechender Eintrag gefunden wurde, scheitert die Operation.

Bei allen Einträgen, die mit `getutxid()` gefunden werden, bezeichnet die Komponente `ut_type` den Typ des Eintrags. Jeder Eintrag enthält, abhängig vom Wert von `ut_type`, weitere Daten, die für die Verarbeitung von Bedeutung sind:

Wert von ut_type	**weitere Komponenten**
`EMPTY`	keine weiteren Daten
`BOOT_TIME`	`ut_tv`
`OLD_TIME`	`ut_tv`
`NEW_TIME`	`ut_tv`
`USER_PROCESS`	`ut_id`, `ut_user` (Benutzerkennung), `ut_line`, `ut_pid`, `ut_tv`
`INIT_PROCESS`	`ut_id`, `ut_pid`, `ut_tv`
`LOGIN_PROCESS`	`ut_id`, `ut_user` (implementierungsspezifischer Name des `login`-Prozesses), `ut_pid`, `ut_tv`
`DEAD_PROCESS`	`ut_id`, `ut_pid`, `ut_tv`

`getutxline()` sucht vorwärts von der aktuellen Position in der Datei `utmpx`, bis ein Eintrag mit dem Typ `LOGIN_PROCESS` oder `USER_PROCESS` gefunden wird, dessen *ut_line* Zeichenkette *line->ut_line* entspricht. Wenn das Dateiende erreicht wird, ohne dass ein entsprechender Eintrag gefunden wird, scheitert die Operation.

`pututxline()` schreibt die angegebene `utmpx`-Struktur in die Datei `utmpx`. `getutxid()` wird verwendet, um nach der korrekten Position in der Datei zu suchen, falls diese noch nicht gegeben ist. Es wird erwartet, dass der Anwender von `pututxline()` den entsprechenden Eintrag mit einer der `getutx()`-Funktionen gesucht hat. Ist dies der Fall, führt `pututxline()` keine Suche durch. Wenn `pututxline()` keine entsprechende Stelle für den neuen Eintrag findet, wird der Eintrag am Dateiende angehängt. Ein Zeiger auf die Struktur `utmpx` wird zurückgegeben.
Um `pututxline()` anwenden zu können, muss der Prozess über die geeignete Privilegierung verfügen.

`setutxent()` setzt die Position des Eingabe-Streams auf den Dateianfang. Dies sollte gemacht werden, bevor in der gesamten Datei nach einem neuen Eintrag gesucht wird.

`endutxent()` schließt die geöffnete Datei.

`endutxent()`, `getutxent()`, `getutxid()`, `getutxline()`, `pututxline()` und `setutxent()` sind nicht threadsicher.

Returnwert `getutxent()`, `getutxid()` und `getutxline()`:

Zeiger auf eine `utmpx`-Struktur
: bei Erfolg. Die zurückgelieferte Struktur enthält eine Kopie des gewünschten Eintrags in der Benutzer-Abrechnungsdatei.

Nullzeiger
: bei Dateiende oder Fehler.

`pututxline()`:

Zeiger auf eine `utmpx`-Struktur
: bei Erfolg. Die zurückgelieferte Struktur enthält eine Kopie des Eintrags, der in die Benutzer-Abrechnungsdatei geschrieben wurde.

Fehler `pututxline()` schlägt fehl, wenn gilt:

`EPERM`
: Der Prozess verfügt nicht über eine ausreichend hohe Privilegierung.

Hinweis Der Returnwert zeigt in einen statischen Bereich, der von einem nachfolgenden Aufruf von `getutxid()` oder `getutxline()` überschrieben wird.

Der aktuellste Eintrag wird in einer statischen Struktur abgelegt. Bevor erneut auf die Datei zugegriffen wird, muss dieser Eintrag kopiert werden. Bei jedem Aufruf von `getutxid()` oder `getutxline()` überprüfen die Routinen die statische Struktur, bevor weitere E/A-Operationen ausgeführt werden. Wenn der Inhalt der statischen Struktur dem gesuchten Muster entspricht, wird nicht weitergesucht. Sollen mit `getutxline()` mehrere identische Einträge gesucht werden, so muss nach jeder erfolgreichen Suchoperation die statische Struktur gelöscht werden; andernfalls gibt `getutxline()` immer wieder dieselbe Struktur zurück.

Das implizite Lesen durch `pututxline()` (wenn die korrekte Position in der Datei noch nicht erreicht wurde) verändert nicht den Inhalt der statischen Struktur, die von `getutxent()`, `getutxid()` oder `getutxline()` zurückgeliefert wird, da `pututxline()` vor dem Lesen den Inhalt der Struktur sichert.

Die Größe der Vektoren in der Struktur kann mit dem Operator `sizeof` bestimmt werden.

Siehe auch `utmpx.h`.

environ - externe Variable für die Umgebung

Syntax extern char **environ;

Beschreibung

`environ` ist eine externe Variable, die auf einen Zeichenkettenvektor mit Umgebungsvariablen zeigt. Dieser wird auch kurz Umgebung genannt. Eine Zeichenkette dieses Vektors hat die Form "*name*=*value*", wobei *name* die Umgebungsvariable und *value* deren aktuellen Wert bezeichnet. Durch Umgebungsvariablen können einer Anwendung Informationen über die Programmumgebung zur Verfügung gestellt werden (siehe Abschnitt „Umgebungsvariablen“ auf Seite 71).

Hinweis Auf den `environ`-Vektor sollte nicht direkt von der Anwendung zugegriffen werden.

Siehe auch `exec`, `getenv()`, `putenv()`, Abschnitt „Umgebungsvariablen“ auf Seite 71.

erand48 - Pseudo-Zufallszahlen zwischen 0.0 und 1.0 mit Startwert generieren

Syntax #include <stdlib.h>

double erand48 (unsigned short int *xsubi*[3]);

Beschreibung

Siehe `drand48()`.

erf, erfc - Fehlerfunktion und komplementäre Fehlerfunktion anwenden

Syntax #include <math.h>

double erf(double *x*);
double erfc(double *x*);

Beschreibung

`erf()` berechnet für die Gleitpunktzahl *x* die Fehlerfunktion, die wie folgt definiert ist:

$$\frac{2}{\sqrt{\Pi}} \int_0^x e^{-t^2}\, dt$$

`erfc()` berechnet für die Gleitpunktzahl *x* die komplementäre Fehlerfunktion:

`1.0 - erf(`*x*`)`.

Returnwert Wert der Fehlerfunktion von *x*
wenn `erf()` erfolgreich beendet wurde.

Wert der komplementären Fehlerfunktion von *x*
wenn `erfc()` erfolgreich beendet wurde.

Hinweis `erfc()` wird zur Verfügung gestellt, da die Berechnung der Fehlerfunktion mit `erf()` bei großen Werten *x* zu Ungenauigkeiten führt.

Siehe auch `math.h`.

errno - Variable für Fehlernummer

Syntax #include <errno.h>

Beschreibung

`errno` wird von vielen Funktionen verwendet, um Fehlernummern zurückzugeben. Programme erhalten die Deklaration von `errno` durch das Inkludieren von `errno.h`. `errno` wird gleich einer Fehlernummer vom Typ `int` gesetzt (siehe `errno.h` und Abschnitt „Fehlerbehandlung“ auf Seite 130).

Beim Programmstart hat `errno` den Wert 0, aber keine der in diesem Handbuch beschriebenen Funktionen setzt `errno` gleich 0, um einen Fehler anzuzeigen. Der Wert von `errno` ist erst nach einem Funktionsaufruf definiert (siehe für jede Funktion den Abschnitt „Fehler“). Durch einen weiteren Funktionsaufruf wird der `errno`-Wert geändert.

Ein Programm, das `errno` zur Fehlerabfrage benutzt, sollte `errno` daher vor dem Funktionsaufruf auf 0 setzen und `errno` vor einem neuen Funktionsaufruf abfragen.

Hinweis `errno` sollte nicht im Quellcode deklariert werden. Bestehende Quellen müssen jedoch nicht geändert werden.

Eine Abbildung zwischen dem numerischen Wert und dem symbolischen Namen der Fehlernummer wird nicht garantiert. Korrektes Verhalten ist nur bei Verwendung der symbolischen Konstantennamen garantiert. Auch die Abbildung von Fehlersituationen auf `errno`-Werte ist nur für die von X/Open geforderten Fälle garantiert.

Siehe auch `perror()`, `strerror()`, `errno.h`, Abschnitt „Fehlerbehandlung“ auf Seite 130.

exec: execl, execv, execle, execve, execlp, execvp - Datei ausführen

Syntax

```
#include <unistd.h>
extern char **environ;

int execl (const char *path, const char *arg0, ... , (char *)0 );
int execv (const char *path, char *const argv[ ]);
int execle (const char *path, const char *arg0, ... , (char *)0, char *const envp[ ] );
int execve (const char *path, char *const argv[ ], const char *envp[ ]);
int execlp (const char *file, const char *arg0, ... , (char *)0 );
int execvp (const char *file, char *const argv[ ]);
```

Beschreibung

Die Funktionen der `exec`-Familie ersetzen das aktuelle Prozessabbild durch ein neues. Das neue Prozessabbild wird aus einer normalen, ausführbaren Datei *path* oder *file* erzeugt, die neue Prozessabbilddatei genannt wird. Ein erfolgreicher Aufruf von `exec` kehrt nicht zurück, da das aufrufende Prozessabbild durch das neue Prozessabbild überlagert wird.

Wenn durch den Aufruf einer `exec`-Funktion ein C-Programm ausgeführt wird, wird dieses wie folgt als C-Funktionsaufruf angesprungen:

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

Dabei ist *argc* der Argumentzähler und *argv* ein Vektor von `char`-Zeigern auf die Argumente selbst. *argc* ist mindestens 1, und das erste Element des Feldes weist auf eine Zeichenkette, die den Namen der ausführbaren Datei enthält.

Zusätzlich wird folgende Variable als Adresse eines Vektors von `char`-Zeigern auf die Umgebungsvariablen zeigen, initialisiert:

```
extern char **environ;
```

argv und `environ` werden durch den Nullzeiger abgeschlossen. Der Nullzeiger, der den Vektor *argv* abschließt, wird in *argc* nicht mitgezählt.

Die Argumente, die von einem Programm bei einer der `exec`-Funktionen angegeben wurden, werden an das neue Prozessabbild über die entsprechenden Argumente von `main()` übergeben.

path zeigt auf einen Pfadnamen, der die neue Prozessabbilddatei angibt.

file wird benutzt, um den Pfadnamen für die neue Prozessabbilddatei zu erzeugen. Wenn *file* einen Schrägstrich enthält, wird es als Pfadname der Prozessabbilddatei angesehen. Wenn *file* keinen Schrägstrich enthält, wird das Pfadpräfix für diese Datei dadurch gefunden, dass die Dateiverzeichnisse durchsucht werden, die durch die Umgebungsvariable `PATH` definiert sind (siehe Abschnitt „Umgebungsvariablen“ auf Seite 71). Die Umgebung

wird typischerweise von der POSIX-Shell bereitgestellt (siehe auch Handbuch „POSIX Grundlagen (BS2000/OSD)“). Andere X/Open-kompatible Systeme können für diesen Fall andere Vorgaben definieren.

Wenn die Prozessabbilddatei kein gültiges ausführbares Objekt ist, verwenden `execlp()` und `execvp()` den Inhalt dieser Datei als Standardeingabe eines Kommando-Interpreters, analog zu `system()`. In diesem Fall wird der Kommando-Interpreter das neue Prozessabbild.

arg0, ... sind Zeiger auf Zeichenketten, die mit dem Nullbyte abgeschlossen sind. Diese Zeichenketten bilden die Argumentliste, die dem neuen Prozessabbild zur Verfügung steht. Die Liste wird durch einen Nullzeiger abgeschlossen. Das Argument *arg*0 sollte auf einen Dateinamen zeigen, der dem Prozess zugeordnet ist, der von einer der `exec`-Funktionen erzeugt wird.

argv ist ein Vektor aus Zeigern auf Zeichenketten, die mit einem Nullbyte abgeschlossen sind. Das letzte Element dieses Vektors muss ein Nullzeiger sein. Diese Zeichenketten stellen die Argumentliste für das neue Prozessabbild dar. Der Wert *argv*[0] sollte auf einen Dateinamen zeigen, der mit dem Prozess verbunden ist, der von einer der `exec`-Funktionen erzeugt wird.

envp ist ein Vektor von Zeigern auf Zeichenketten, die mit dem Nullbyte abgeschlossen sind. Diese Zeichenketten bilden die Umgebung für das neue Prozessabbild. Der Vektor *envp* wird durch einen Nullzeiger abgeschlossen.

Bei den Funktionen, die *envp* nicht als Argument übergeben (`execl()`, `execv()`, `execlp()` und `execvp()`), wird die Umgebung für das neue Prozessabbild aus der externen Variablen `environ` des aufrufenden Prozesses gewonnen.

Die Anzahl von Bytes, die für die Argument- und Umgebungsliste des Prozesses zur Verfügung steht, ist `{ARG_MAX}`. Im POSIX-Subsystem schließt die Konstante `{ARG_MAX}` den Platz für abschließende Nullbytes, Zeiger und/oder Füllbytes mit ein. Andere X/Open-kompatible Systeme können hier andere Vereinbarungen treffen.

Dateideskriptoren des aufrufenden Prozessabbilds bleiben auch im neuen Prozessabbild offen, außer denen, für die das sbe-Bit `FD_CLOEXEC` gesetzt ist (siehe auch `fcntl()`). Für die Dateien, die offen bleiben, bleiben auch alle Attribute der Dateibeschreibung bestehen, einschließlich der Dateisperren.

Der Zustand von Umwandlungs- und Meldungskatalog-Deskriptoren im neuen Prozessabbild ist undefiniert. Für den neuen Prozess wird folgendes Äquivalent beim Systemstart ausgeführt:

```
setlocale(LC_ALL, "C")
```

Signale, die im aufrufenden Prozessabbild auf die Signalaktion SIG_DFL gesetzt sind, werden im neuen Prozessabbild auf die voreingestellte Signalaktion gesetzt. Signale, die im aufrufenden Prozessabbild ignoriert werden (SIG_IGN), werden auch im neuen Prozessabbild ignoriert. Signale, die im aufrufenden Prozessabbild abgefangen werden, werden im neuen Prozessabbild auf die voreingestellte Signalaktion gesetzt (siehe auch signal.h).

Nach einem erfolgreichen Aufruf einer exec-Funktion sind die vorher mit atexit() registrierten Funktionen nicht mehr registriert.

Wenn das s-Bit für den Eigentümer bei der neuen Prozessabbilddatei gesetzt ist (siehe auch chmod()), wird die effektive Benutzernummer des neuen Prozessabbilds auf die Benutzernummer des Eigentümers der neuen Prozessabbilddatei gesetzt. Analog dazu wird, wenn das s-Bit für die Gruppe der neuen Prozessabbilddatei gesetzt ist, die effektive Gruppennummer des neuen Prozessabbilds auf die Gruppennummer der neuen Prozessabbilddatei gesetzt. Die reale Benutzer- und Gruppennummer sowie die zusätzlichen Gruppennummern des neuen Prozessabbilds bleiben dieselben wie die des aufrufenden Prozessabbilds. Die effektive Benutzer- und die effektive Gruppennummer des neuen Prozessabbilds werden für eine Verwendung durch setuid() als die gesicherte Benutzer- und die gesicherte Gruppennummer gespeichert.

Gemeinsam nutzbare Speicherbereiche, die an das aufrufende Prozessabbild angehängt sind, werden nicht an das neue Prozessabbild angehängt (siehe auch shmat()).

Der neue Prozess erhält außerdem folgende Attribute aus dem aufrufenden Prozessabbild:

- Prioritätswert (siehe auch nice())
- semadj-Werte (siehe auch semop())
- Prozessnummer
- Vaterprozessnummer
- Prozessgruppennummer
- Sitzungsnummer
- reale Benutzernummer
- reale Gruppennummer
- zusätzliche Gruppennummern
- Restzeit bis zu einem Alarmuhr-Signal (siehe auch alarm())
- aktuelles Dateiverzeichnis
- Root-Dateiverzeichnis
- Schutzbitmaske (siehe auch umask())
- maximale Dateigröße (siehe auch ulimit())
- Signalmaske (siehe auch sigprocmask())
- wartende Signale (siehe auch sigpending())
- tms_utime, tms_stime, tms_cutime und tms_cstime (siehe auch times())

Alle anderen Prozessattribute der XPG4-konformen Bibliotheksfunktionen sind im alten und neuen Prozessabbild identisch.

Bei erfolgreicher Beendigung markieren die exec-Funktionen das Feld st_atime der Datei zum Ändern. Wenn eine exec-Funktion fehlschlägt, aber die Prozessabbilddatei gefunden hatte, ist nicht festgelegt, ob das Feld st_atime zum Ändern markiert ist. Sollte die exec-Funktion erfolgreich sein, nimmt man an, dass die Prozessabbilddatei geöffnet wurde. Das korrespondierende Schließen wird für einen Zeitpunkt nach dem Öffnen angesetzt, aber vor der Beendigung des Prozesses oder der erfolgreichen Beendigung eines nachfolgenden Aufrufs einer der exec-Funktionen.

POSIX-Dateien werden beim Aufruf einer exec-Funktion nur dann geschlossen, wenn das Flag CLOSE_ON_EXEC gesetzt ist.

Werden Threads verwendet, so wirkt sich die Funktion auf den Prozess oder auf einen Thread wie folgt aus:

– Beim Aufruf einer der exec()-Funktionen aus einem Prozess mit mehr als einem Thread werden alle Threads beendet und danach wird das neue ausführbare Programm geladen und ausgeführt. Es werden keine Destruktor-Funktionen aufgerufen.

BS2000

– BS2000-Dateien werden beim Aufruf einer exec()-Funktion immer geschlossen. ❑

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

Fehler Die exec-Funktionen schlagen fehl, wenn gilt:

E2BIG Die Anzahl der Bytes, die von der neuen Argument- und Umgebungsliste des Prozessabbilds verwendet werden, ist größer als die systemspezifische Grenze von {ARG_MAX} Bytes.

EACCES Das Durchsuchrecht für ein Dateiverzeichnis im Pfad-Präfix der neuen Prozessabbilddatei ist nicht gegeben,
oder die neue Prozessabbilddatei verweigert das Ausführrecht,
oder die neue Prozessabbilddatei ist keine normale Datei und die Implementierung unterstützt die Ausführung von Dateien dieses Typs nicht.

Erweiterung

EFAULT Programm konnte nicht geladen werden.

EINTR Ein Signal wurde abgefangen.

ELOOP Beim Übersetzen von *path* oder *file* wurden zuviele symbolische Verweise angetroffen. ❑

ENAMETOOLONG
Die Länge der Argumente *path* oder *file* oder ein Element der Umgebungsvariablen PATH, das einer Datei vorangestellt wird, überschreitet {PATH_MAX}, oder eine Pfadnamen-Komponente ist länger als {NAME_MAX}.

Erweiterung

ENOENT Eine oder mehrere Komponenten des Pfadnamens der neuen Prozessabbilddatei existieren nicht, oder *path* oder *file* zeigen auf eine leere Zeichenkette.

ENOMEM Ein neues Prozessabbild erfordert mehr Speicherplatz, als von der Hardware oder den systemspezifischen Speicherverwaltungseinschränkungen zugelassen ist.

ENOTDIR Eine Komponente des Pfad-Präfixes der neuen Prozessabbilddatei ist kein Dateiverzeichnis. ❑

Die exec-Funktionen - außer execlp() und execvp() - schlagen fehl, wenn gilt:

ENOEXEC Die neue Prozessabbilddatei besitzt zwar die nötigen Zugriffsrechte, aber nicht das richtige Format.

Hinweis Da der Zustand von Umwandlungs- und Meldungskatalog-Deskriptoren im neuen Prozessabbild undefiniert ist, sollten sich portable Anwendungen nicht auf deren Verwendung abstützen und diese vor der Verwendung einer der exec-Funktionen schließen.

Die Umgebungsvariablen BLSLIB*nn* (für 00 <= *nn* <= 98) werden vor dem Laden des auszuführenden Programms beginnend bei BLSLIB00 in aufsteigender Reihenfolge ausgewertet. Der Inhalt der Variablen wird als BS2000-Dateiname interpretiert und ein Link mit dem Variablennamen auf den jeweiligen Dateinamen abgesetzt. Bei der ersten nicht vorhandenen Variablen wird die weitere Suche abgebrochen. Auf jeden Fall aber wird ein Link mit dem Linknamen BLSLIB99 auf die Datei $.SYSLNK.CRTE abgesetzt. Dieses Verfahren ermöglicht es, auch nicht vollständig gebundene Programme, die noch dynamisch Module nachladen müssen, in einem Sohnprozess auszuführen, der die TFT (Terminal File Table) nicht von seinem Vaterprozess erbt.

Siehe auch alarm(), atexit(), exit(), fcntl(), fork(), getenv(), nice(), putenv(), semop(), setlocale(), shmat(), sigaction(), system(), times(), ulimit(), umask(), unistd.h, Abschnitt „Umgebungsvariablen“ auf Seite 71.

exit, _exit - Prozess beenden

Syntax

```
#include <stdlib.h>
void exit (int status);

#include <unistd.h>
void _exit (int status);
```

Beschreibung

`_exit()` und `exit()` beenden den aufrufenden Prozess.

Bei einem `exit`-Aufruf werden folgende Aktionen ausgelöst:

1. `exit()` ruft alle Funktionen auf, die durch `atexit()` registriert wurden, und zwar in der umgekehrten Reihenfolge ihrer Registrierung. Wenn eine von `atexit()` registrierte Funktion nicht zurückkehrt, werden keine weiteren registrierten Funktionen mehr aufgerufen, und die Ausführung von `exit()` wird abgebrochen. Wenn `exit()` mehr als einmal aufgerufen wird, ist das Verhalten nicht definiert.
2. `exit()` leert alle Ausgabeströme, schließt alle Datenströme und löscht alle Dateien, die von `tmpfile()` erzeugt wurden.

Mit `_exit()` werden im Unterschied zu `exit()` die mit `atexit()` registrierten Prozessendefunktionen nicht aufgerufen und geöffnete Dateien nicht geschlossen.

`_exit()` und `exit()` beenden den aufrufenden Prozess mit den folgenden Konsequenzen:

– Alle Dateideskriptoren, Dateiverzeichnisströme, Umwandlungsdeskriptoren und Meldungskatalog-Deskriptoren, die für den aufrufenden Prozess offen sind, werden geschlossen.

– Wenn der Vaterprozess des aufrufenden Prozesses `wait()` oder `waitpid()` ausführt, wird dieser von der Beendigung des aufrufenden Prozesses benachrichtigt und die niederwertigen 8 Bit von *status*, d.h. die Bits 0377, werden ihm verfügbar gemacht (siehe auch `wait()` und `waitpid()`).

– Wenn der Vaterprozess nicht wartet und anschließend `wait()` oder `waitpid()` ausführt, wird ihm der Status des Sohnprozesses verfügbar gemacht.

– Wenn der Vaterprozess des aufrufenden Prozesses kein `wait()` oder `waitpid()` ausführt, wird der aufrufende Prozess in einen so genannten Zombieprozess umgewandelt. Ein **Zombieprozess** ist ein inaktiver Prozess; er wird zu einem späteren Zeitpunkt gelöscht, nämlich wenn sein Vaterprozess `wait()` oder `waitpid()` ausführt.

- Die Beendigung eines Prozesses beendet nicht unmittelbar dessen Sohnprozesse. Das Senden des Signals SIGHUP beendet, wie unten beschrieben, Sohnprozesse indirekt unter bestimmten Umständen.
- Im POSIX-Subsystem wird zusätzlich das Signal SIGCHLD an den Vaterprozess des aufrufenden Prozesses gesendet. Andere X/Open-kompatible Systeme können für diesen Fall andere Vorgaben definieren.
- Die Vaterprozessnummer aller existierenden Sohn- oder Zombieprozesse des aufrufenden Prozesses wird gleich der Prozessnummer eines speziellen Systemprozesses gesetzt. Das heißt, diese Prozesse werden von dem Systemprozess init geerbt, dessen Prozessnummer gleich 1 ist.
- Jedes angehängte Segment des gemeinsam nutzbaren Speichers wird abgehängt, und der Wert von shm_nattch (siehe shmget()) in der Datenstruktur, die seiner Nummer für gemeinsam nutzbaren Speicher zugeordnet ist, wird um 1 dekrementiert.
- Für jedes Semaphor, für das der aufrufende Prozess einen semadj-Wert gesetzt hat (siehe auch semop()), wird dieser Wert zum semval des angegebenen Semaphors addiert.
- Wenn der Prozess ein steuernder Prozess ist, wird das Signal SIGHUP an jeden Prozess in der Vordergrundprozessgruppe des steuernden Terminals gesendet, das zu dem aufrufenden Prozess gehört.
- Wenn der Prozess ein steuernder Prozess ist, wird das steuernde Terminal, das dieser Sitzung zugeordnet ist, wieder freigegeben, wodurch es von einem neuen steuernden Prozess belegt werden kann.
- Wenn durch das Prozessende eine Prozessgruppe verwaist und ein Mitglied der frisch verwaisten Prozessgruppe angehalten wird, wird erst das Signal SIGHUP und dann das Signal SIGCONT an jeden Prozess der frisch verwaisten Prozessgruppe gesendet.

Die Symbole EXIT_SUCCESS und EXIT_FAILURE sind in stdlib.h definiert und können als Wert von *status* verwendet werden, um erfolgreiches oder nicht erfolgreiches Beenden anzuzeigen.

exit() und _exit() kehren nicht zurück.

Werden Threads verwendet, so wirkt sich die Funktion auf den Prozess oder auf einen Thread wie folgt aus:

– Beenden des Prozesses. Threads, die durch einen Aufruf von `_exit()` beendet werden, rufen nicht ihre cancellation cleanup handler oder die Daten-Destruktoren des Threads auf.

– *BS2000*
Die Monitor-Jobvariable `MONJV` wird nach folgenden Regeln versorgt:

– Je nach Wert des Parameters *status* wird die Zustandsanzeige der Monitor-Jobvariablen `MONJV` (1. - 3. Byte) auf den Wert "`$T` " oder "`$A` " gesetzt, und es werden die Variablen `SUBCODE1`, `SUBCODE2` und `MAINCODE`, die mit den gleichnamigen vordefinierten Funktionen von `SDF-P` abgefragt werden können, versorgt.

status kann die in der Include-Datei `stdlib.h` definierten symbolischen Konstanten `EXIT_SUCCESS` und `EXIT_FAILURE` oder einen beliebigen integer-Wert enthalten:

`EXIT_SUCCESS` (Wert 0)
verursacht eine normale Programmbeendigung.

Die Zustandsanzeige der `MONJV` bekommt den Wert "`$T` " zugewiesen. Außerdem werden `SUBCODE`=0, `MAINCODE` =`CCM0998` und `SUBCODE2`=*status* `modulo 256` gesetzt.

`EXIT_FAILURE` (Wert 9990888)
verursacht eine so genannte **Jobstep-Beendigung**:

– Das Programm wird anormal beendet.

– In einer `DO`- oder `CALL`-Prozedur verzweigt das System zum nächsten Kommando `ABEND`, `END-PROCEDURE`, `SET-JOB-STEP` oder `LOGOFF`.

– Es erfolgt die Systemmeldung `"ABNORMAL PROGRAM TERMINATION"`.

Die Zustandsanzeige der `MONJV` bekommt den Wert "`$A` " zugewiesen. Außerdem werden `SUBCODE`=1, `MAINCODE`=`CCM0999` und `SUBCODE2`=*status* `modulo 256` gesetzt.

`integer`-Wert ≠ 0 bzw. ≠ 9990888
eine Jobstep-Beendigung wird durchgeführt, und die Zustandsanzeige der `MONJV` bekommt den Wert "`$T`" zugewiesen. Außerdem werden `SUBCODE`=1, `MAINCODE`=`CCM0999` und `SUBCODE2`=*status* `modulo 256` gesetzt.

Enspricht dieser Wert den vordefinierten Werten `EXIT_SUCCESS` oder `EXIT_FAILURE`, werden die oben genannten Aktionen durchgeführt. ❑

Hinweis Normalerweise sollten Anwendungen `exit()` an Stelle von `_exit()` verwenden.

BS2000
Um Monitor-Jobvariablen versorgen und abfragen zu können, müssen Sie das C-Programm von der BS2000-Oberfläche aus mit folgendem Kommando starten:

```
/START-PROG programm,MONJV=monjvname
```

Der Inhalt der Jobvariablen lässt sich dann z.B. mit folgendem Kommando abfragen:

```
/SHOW-JV JV-NAME(monjvname)
```

Weitere Informationen zur Ablaufüberwachung mit Monitor-Jobvariablen finden Sie im Handbuch "JV (BS2000)". ❑

Siehe auch `abort()`, `atexit()`, `bs2exit()`, `close()`, `fclose()`, `semop()`, `shmget()`, `sigaction()`, `wait()`, `stdlib.h`, `unistd.h`.

exp - Exponentialfunktion anwenden

Syntax #include <math.h>

double exp(double *x*);

Beschreibung

`exp()` berechnet die Exponentialfunktion für die zulässige Gleitpunktzahl *x*.

Returnwert e^x bei Erfolg.

`HUGE_VAL` bei Überlauf. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `exp()` schlägt fehl, wenn gilt:

`ERANGE` Überlauf, der Returnwert ist zu groß.

Siehe auch `log()`, `log10()`, `pow()`, `math.h`.

expm1 - Exponentialfunktionen berechnen

Syntax #include <math.h>

double expm1(double *x*);

Beschreibung

Die Funktion `expm1()` berechnet e^x-1.0.

Returnwert e^x-1.0 bei Erfolg.

`HUGE_VAL` bei Überlauf. `errno` wird gesetzt, um den Fehler anzuzeigen.

Hinweis Für kleine *x*-Werte kann das Ergebnis von `expm1(x)` genauer sein als der Wert von `exp(x)`-1.0. Die Funktionen `expm1()` und `log1p()` sind hilfreich zur Berechnung des Ausdrucks $((1+x)^n-1)/x$, in der Form: `expm1(n * log1p(x))/x` bei sehr kleinen Werten von *x*.

Mit Hilfe dieser Funktion können auch inverse hyperbolische Funktionen genau dargestellt werden.

Siehe auch `exp()`, `ilogb()`, `log1p()`, `math.h`.

fabs - Absolutwert einer Gleitpunktzahl berechnen

Syntax #include <math.h>

double fabs(double *x*);

Beschreibung

`fabs()` berechnet den Absolutwert der Gleitpunktzahl *x*.

Returnwert Absolutwert für die Gleitpunktzahl *x*
bei Erfolg.

Siehe auch `abs()`, `cabs()`, `ceil()`, `floor()`, `math.h`.

fattach - einem Objekt im Namensraum des Dateisystems einen Dateideskriptor unter STREAMS zuordnen

Syntax #include <stropts.h>

int fattach (int *fildes*, const char **path*);

Beschreibung

Die Funktion `fattach()` ordnet einem Objekt (Datei oder Dateiverzeichnis) im Namensraum des Dateisystems einen Dateideskriptor unter STREAMS zu, wobei *fildes* ein Pfadname zugeordnet wird. *fildes* muss ein gültiger, offener Dateideskriptor sein, der eine STREAMS-Datei repräsentiert. *path* ist ein Pfadname eines existierenden Objekts, dessen Eigentümer mit Schreiberlaubnis der Benutzer sein muss. Alternativ dazu kann der Benutzer auch besondere Rechte besitzen. Alle nachfolgenden Operationen auf *path* arbeiten mit der STREAMS-Datei, solange, bis die Zuordnung der STREAMS-Datei zum Knoten aufgehoben wird. *fildes* kann mehr als einem Pfad zugeordnet sein, d. h. dem Dateideskriptor können mehrere Namen zugeordnet sein.

Die Attribute des benannten Streams werden folgendermaßen initialisiert (siehe auch `stat()`): Zugriffsrechte, Benutzer- und Gruppennummern sowie die Dateizeiten werden gleich denen von *path*, die Anzahl der Verweise wird gleich 1 und Größe und Geräte-Identifikation werden gleich den Werten gesetzt, die das STREAMS-Gerät zu *fildes* besitzt. Werden irgendwelche Attribute des benannten Streams anschließend geändert (z. B. mit `chmod()`), dann werden weder die Attribute des zu Grunde liegenden Objekts noch die Attribute der STREAMS-Datei, auf die sich *fildes* bezieht, davon beeinflusst.

Dateideskriptoren, die sich auf das zu Grunde liegende Objekt beziehen und noch vor einem Aufruf von `fattach()` geöffnet wurden, beziehen sich weiterhin auf das zu Grunde liegende Objekt.

Returnwert

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

Fehler `fattach()` schlägt fehl, wenn gilt:

`EACCES`	Eine Komponente des Pfades darf nicht durchsucht werden, oder der Benutzer ist der Eigentümer von *path*, besitzt jedoch keine Schreiberlaubnis für *path*.
`EBADF`	*fildes* ist kein gültiger, offener Dateideskriptor.
`ENOENT`	Eine Komponente des Pfadnamens existiert nicht, oder *path* zeigt auf eine leere Zeichenkette.
`ENOTDIR`	Eine Komponente des Pfadnamen-Präfix ist kein Dateiverzeichnis.
`EPERM`	Die effektive Benutzernummer des Prozesses ist nicht die des Eigentümers der mit *path* bezeichneten Datei und der Prozess hat nicht die entsprechenden Zugriffsrechte.
`EBUSY`	*path* ist derzeit ein Einhängpunkt oder diesem Pfad ist eine STREAMS-Datei zugeordnet.
`ENAMETOOLONG`	Die Länge von *path* überschreitet `{PATH_MAX}`, oder eine Komponente des Pfadnamens ist länger als `{NAME_MAX}` , während `{_POSIX_NO_TRUNC}` aktiv ist; oder Die Auflösung eines symbolischen Verweises des Pfadnamens erzeugt ein Zwischenergebnis, das länger ist als `{PATH_MAX}`.
`ELOOP`	Bei der Übersetzung von *path* traten zuviele symbolische Verweise auf.
`EINVAL`	*fildes* repräsentiert keine STREAMS-Datei.
`EREMOTE`	*path* ist eine Datei in einem von fern eingehängten Dateiverzeichnis.

Hinweis `fattach()` verhält sich ähnlich wie die ältere Funktion `mount()`, derart dass ein Objekt vorübergehend durch das Root-Verzeichnis des eingehängten Dateisystems ersetzt wird. Bei `fattach()` muss das ersetzte Objekt kein Verzeichnis sein und die ersetzende Datei ist eine STREAMS-Datei.

Siehe auch `fdetach()`, `isastream()`, `stropts.h`.

fchdir - aktuelles Dateiverzeichnis ändern

Syntax #include <unistd.h>

int fchdir(int *fildes*);

Beschreibung

`fchdir()` wechselt ebenso wie `chdir()` das aktuelle Dateiverzeichnis wobei jedoch das neue Verzeichnis nicht durch den Pfadnamen, sondern durch den Dateideskriptor *fildes* bezeichnet wird. Das aktuelle Dateiverzeichnis ist der Startpunkt für Suchen nach Pfadnamen, die nicht mit „/“ beginnen. Das *fildes*-Argument ist ein offener Dateideskriptor eines Verzeichnisses.
Um ein Verzeichnis zum aktuellen Verzeichnis zu machen, muss ein Prozess Zugriffsrechte zum Ausführen (Suchen) auf das Verzeichnis haben.

Returnwert

0	bei Erfolg.
-1	bei Fehler. Das aktuelle Arbeitsverzeichnis bleibt unverändert. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `fchdir()` schlägt fehl, wenn gilt:

`EACCES`	Für *fildes* gibt es keine Durchsucherlaubnis.
`EBADF`	*fildes* ist kein Dateideskriptor für eine offene Datei.
`ENOTDIR`	Der offene Dateideskriptor zeigt nicht auf ein Dateiverzeichnis.
`EINTR`	Ein Signal wurde während des Systemaufrufs `fchdir()` abgefangen.
`EIO`	Es trat während des Lesens oder Schreibens vom Dateisystem ein Ein- oder Ausgabefehler auf.
`ENOLINK`	*fildes* weist auf einen fernen Rechner, und die Verbindung zu diesem Rechner ist nicht mehr aktiv.

Hinweis Die Änderung des aktuellen Dateiverzeichnisses wirkt für die Dauer des aktuellen Programmes (bzw. der aktuellen Shell). Wird ein Programm oder eine Shell neu gestartet, dann ist wieder das Home-Verzeichnis als aktuelles Dateiverzeichnis eingestellt.
Um ein Verzeichnis zum aktuellen Dateiverzeichnis zu machen, muss ein Prozess Ausführrechte (Suchen) für das Verzeichnis haben.
`fchdir()` wirkt nur in dem jeweils aktiven Prozess und nur bis zur Beendigung des aktiven Programms.
`fchdir()` wird nur für POSIX-Dateien ausgeführt.

Siehe auch `chdir()`, `chroot()`, `unistd.h`.

fchmod - Dateizugriffsrechte ändern

Syntax

```
#include <sys/types.h>
#include <sys/stat.h>

int fchmod(int fildes, mode_t mode);
```

Beschreibung

`fchmod()` ändert ebenso wie `chmod()` `S_ISUID`, `S_ISGID` und die Schutzbits der angesprochenen Datei in die entsprechenden Bits von *mode* um, nur dass die Datei, deren Zugriffsrechte geändert werden sollen, nicht durch den Pfadnamen, sondern durch den Dateideskriptor *fildes* bezeichnet wird. Die Schutzbits werden wie folgt interpretiert (siehe auch `sys/stat.h`):

Symbolischer Name	Bitmuster	Bedeutung
`S_ISUID`	`04000`	Benutzernummer bei Ausführung setzen
`S_ISGID`	`020#0`	Gruppennummer bei Ausführung setzen, wenn # den Wert 7, 5, 3 oder 1 hat. Aufhebung der obligatorischen Sperre von Dateien und Dateisätzen, wenn # den Wert 6, 4, 2 oder 0 hat.
`S_ISVTX`	`01000`	Textsegment nach Ausführung sichern
`S_IRWXU`	`00700`	Lesen, Schreiben, Ausführen (Durchsuchen) durch Eigentümer
`S_IRUSR`	`00400`	Lesen durch Eigentümer
`S_IWUSR`	`00200`	Schreiben durch Eigentümer
`S_IXUSR`	`00100`	Ausführen durch Eigentümer (Durchsuchen, wenn es sich um ein Dateiverzeichnis handelt)
`S_IRWXG`	`00070`	Lesen, Schreiben, Ausführen (Durchsuchen) durch Gruppe
`S_IRGRP`	`00040`	Lesen durch Gruppe
`S_IWGRP`	`00020`	Schreiben durch Gruppe
`S_IXGRP`	`00010`	Ausführen durch Gruppe
`S_IRWXO`	`00007`	Lesen, Schreiben, Ausführen (Durchsuchen) durch Andere
`S_IROTH`	`00004`	Lesen durch Andere
`S_IWOTH`	`00002`	Schreiben durch Andere
`S_IXOTH`	`00001`	Ausführen durch Andere

Andere Modi werden durch bitweise ODER-Verknüpfung der Zugriffsmodi erzeugt.

Die effektive Benutzernummer des Prozesses muss mit der des Eigentümers der Datei übereinstimmen oder der Prozess muss das entsprechende Privileg haben, damit der Modus einer Datei geändert werden kann.

Wenn weder der Prozess noch ein Mitglied der anhängenden Gruppenliste privilegiert ist und die effektive Gruppennummer des Prozesses nicht mit der Gruppennummer der Datei übereinstimmt, wird das Schutzbit 02000 (Gruppennummer bei Ausführung setzen) gelöscht.

Wenn das Schutzbit 02000 (Gruppennummer bei Ausführung setzen) gesetzt und das Modusbit 00010 (Ausführen oder Suchen durch Gruppe) nicht gesetzt ist, liegt das obligatorische Sperren von Dateien und Dateisätzen bei einer normalen Datei vor. Dies kann zukünftige Aufrufe von `open()`, `creat()`, `read()` und `write()` auf diese Datei beeinflussen.

Wenn der Prozess kein privilegierter Prozess und die Datei kein Dateiverzeichnis ist, wird das Modusbit 01000 (Textsegment nach Ausführung sichern) gelöscht.

Wenn ein Verzeichnis beschrieben werden kann und das Sticky-Bit gesetzt ist, können Dateien in diesem Verzeichnis nur dann gelöscht oder umbenannt werden, wenn mindestens eine der folgenden Bedingungen zutrifft (siehe `unlink()` und `rename()`):

die Datei gehört dem Benutzer
das Verzeichnis gehört dem Benutzer
der Benutzer hat das Recht, auf die Datei zu schreiben
der Benutzer ist ein privilegierter Benutzer

Bei einer erfolgreichen Beendigung markiert `fchmod()` das Feld `st_ctime` der Datei zum Aktualisieren.

Returnwert

0	bei Erfolg.
-1	bei Fehler. Der Dateimodus wird nicht verändert. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `fchmod()` schlägt fehl, wenn gilt:

`EBADF`	*fildes* ist kein offener Dateideskriptor.
`EINVAL`	Es wurde versucht, auf eine BS2000-Datei zuzugreifen, oder der Wert von *mode* ist ungültig.
`EIO`	Während des Lesens oder Schreibens im Dateisystem trat ein Fehler auf.
`EINTR`	Ein Signal wurde während der Ausführung des `fchmod()`–Systemaufrufs abgefangen.
`EPERM`	Die Benutzernummer entspricht nicht der des Dateieigentümers, und der Prozess ist nicht entsprechend privilegiert.

EROFS	Die durch *fildes* angegebene Datei steht in einem schreibgeschützten Dateisystem.

Hinweis `fchmod()` wird nur für POSIX-Dateien ausgeführt.

Siehe auch `chmod()`, `chown()`, `creat()`, `fcntl()`, `fstatvfs()`, `mknod()`, `open()`, `read()`, `rename()`, `stat()`, `unlink()`, `write()`, `sys/stat.h`, `sys/types.h`.

fchown - Eigentümer oder Gruppe einer Datei ändern

Syntax

```
#include <unistd.h>

int fchown(int fildes, uid_t owner, gid_t group);
```

Beschreibung

`fchown()` ändert ebenso wie `chown()` die Benutzernummer und die Gruppennummer der angesprochenen Datei, außer dass die Datei nicht durch den Pfadnamen, sondern durch den Dateideskriptor *fildes* bezeichnet wird. Die Benutzernummer wird auf *owner*, die Gruppennummer auf *group* gesetzt. Wenn *owner* oder *group* mit -1 spezifiziert ist, wird die der Datei zugehörige ID nicht geändert.

Wenn `fchown()` von einem Prozess ohne Sonderrechte aufgerufen wird, dann wird das Bit zum Setzen der Benutzer- und Gruppennummer bei Ausführung, beziehungsweise `S_ISUID` und `S_ISGID`, gelöscht (siehe `chmod()`).
Die effektive Benutzernummer des Prozesses muss die des Eigentümers der Datei sein oder der Prozess muss Sonderrechte haben, um den Besitz der Datei zu ändern.

Bei erfolgreicher Beendigung markiert `fchown()` das Feld `st_ctime` der Datei zum Aktualisieren.

Returnwert

0	bei Erfolg. Benutzer- und Gruppennummer der angegebenen Datei sind entsprechend gesetzt.
-1	bei Fehler. Benutzer- und Gruppennnummer der Datei werden nicht geändert. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `fchown()` schlägt fehl, wenn gilt:

`EABDF`	*fildes* verweist nicht auf eine offene Datei.
`EINTR`	Ein Signal wurde während des Systemaufrufs abgefangen.
`EINVAL`	Es wurde versucht, auf eine BS2000-Datei zuzugreifen.
	group oder *owner* sind außerhalb des zulässigen Bereichs.
`EIO`	Es trat während des Lesens oder Schreibens im Dateisystem ein Ein- oder Ausgabefehler auf.
`EPERM`	Die Benutzernummer entspricht nicht dem Eigentümer der Datei, oder der Prozess ist nicht entsprechend privilegiert.
`EROFS`	Die Datei steht in einem schreibgeschützten Dateisystem.

Hinweis `fchown()` wird nur für POSIX-Dateien ausgeführt.

Siehe auch `chmod()`, `chown()`, `unistd.h`.

fclose - Datenstrom schließen

Syntax #include <stdio.h>

int fclose(FILE **stream*);

Beschreibung

`fclose()` leert den Puffer des Datenstroms, auf den *stream* zeigt, und schließt die zugehörige Datei. Alle gepufferten, aber noch nicht geschriebenen Daten für diesen Datenstrom werden in die Datei geschrieben; alle gepufferten, noch nicht gelesenen Daten werden entfernt. Die Zuordnung des Datenstroms zur Datei wird aufgehoben. Wurde der zugehörige Puffer automatisch reserviert, wird er wieder freigegeben. Die Funktion `fclose()` führt ein `close()` für den Dateideskriptor aus, auf den *stream* zeigt.

Nach dem Aufruf von `fclose()` ist das Verhalten von *stream* undefiniert.

Returnwert

0	bei Erfolg
`EOF`	bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `fclose()` schlägt fehl, wenn gilt:

`EAGAIN`	Das Kennzeichen `O_NONBLOCK` ist für den *stream* zu Grunde liegenden Dateideskriptor gesetzt, und der Prozess würde durch eine Schreiboperation verzögert.
`EBADF`	Der *stream* zu Grunde liegende Dateideskriptor ist kein gültiger Dateideskriptor.
Erweiterung	Die BS2000-Datei ist nicht in diesem Prozess zugreifbar. ❑
`EFBIG`	Es wurde versucht, in eine Datei zu schreiben, deren Größe die maximale Dateigröße oder die Grenze des Prozesses für die Dateigröße überschreitet (siehe auch `ulimit()`).
`EINTR`	`fclose()` wurde durch ein Signal unterbrochen.
`EIO`	Ein Ein-/Ausgabefehler ist aufgetreten. Der Prozess ist Mitglied einer Hintergrundprozessgruppe, die auf ihr steuerndes Terminal schreiben will, `TOSTOP` ist gesetzt, das Signal `SIGTTOU` wird vom Prozess weder ignoriert noch blockiert, und die Prozessgruppe des Prozesses ist verwaist.
`ENOSPC`	Auf dem Datenträger, auf dem sich die Datei befindet, ist kein Platz mehr frei.

`ENXIO` Es wurde ein nicht existierendes Gerät angefordert oder die Anforderung lag außerhalb der Leistungsgrenzen des Geräts.

`EPIPE` Es wurde der Versuch unternommen, auf eine Pipe oder FIFO zu schreiben, die durch keinen Prozess zum Lesen geöffnet war. An den Prozess wird auch das Signal `SIGPIPE` gesendet.

Werden Threads verwendet, so wirkt sich die Funktion auf den Prozess oder auf einen Thread wie folgt aus: Beim `EPIPE`-Fehler wird das Signal `SIGPIPE` nicht an den Prozess, sondern an den aufrufenden Thread gesendet.

Hinweis Jedes Mal, wenn ein Programm normal oder mit `exit()` beendet wird, wird für jede offene Datei automatisch ein `fclose()` ausgeführt. `fclose()` braucht also nur dann explizit aufgerufen zu werden, wenn vor Programmbeendigung eine Datei geschlossen werden soll, z.B. um das Limit für geöffnete Dateien (=2048) nicht zu überschreiten.

Ob `fclose()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

BS2000
Zeigt *stream* nicht auf eine `FILE`-Struktur, bricht das Programm ab.

Da bei Satz-Ein-/Ausgabe keine Daten gepuffert werden, entfällt der interne Aufruf der Funktion `fflush()`. ❑

Siehe auch `close()`, `exit()`, `fflush()`, `fopen()`, `setbuf()`, `stdio.h`.

fcntl - offene Datei steuern

Name **fcntl**

Syntax #include <fcntl.h>

Optional
#include <sys/types.h>
#include <unistd.h>

int fcntl(int *fildes*, int *cmd*, ... /* *arg* */);

Beschreibung

`fcntl()` ermöglicht die Steuerung von offenen Dateien.

fildes ist ein Dateideskriptor einer offenen Datei.

An `fcntl()` kann ein drittes Argument übergeben werden, dessen Datentyp und Wert vom übergebenen Kommando *cmd* abhängen. *cmd* spezifiziert die Operation, die von `fcntl()` ausgeführt wird, und kann einer der folgenden Werte sein:

`F_DUPFD` Ein neuer Dateideskriptor wird wie folgt zurückgegeben:

- Dateideskriptor mit der niedrigsten verfügbaren Nummer, die größer als oder gleich dem ganzzahligen Wert ist, der als drittes Argument (*arg*) übergeben wird
- dieselbe offene Datei (oder Pipe) wie die ursprüngliche Datei
- derselbe Schreib-/Lesezeiger wie der der ursprünglichen Datei (d.h. beide Dateideskriptoren teilen sich denselben Schreib-/Lesezeiger)
- derselbe Zugriffsmodus (Lesen, Schreiben oder Lesen/Schreiben) wie die ursprüngliche Datei
- dieselben Dateistatus-Bits wie die ursprüngliche Datei
- das Bit 'Schließen-bei-exec' (siehe `F_GETFD`), das zum neuen Dateideskriptor gehört, so setzen, dass die Datei bei `exec()`-Aufrufen geöffnet bleibt

`F_GETFD` ruft das Flag 'Schließen-bei-exec' auf, das zu dem Dateideskriptor *fildes* gehört. Wenn das niederwertige Bit `0` ist, bleibt die Datei bei `exec` offen, andernfalls wird die Datei bei Aufruf von `exec` geschlossen.

`F_SETFD` setzt das zu *fildes* gehörende Flag 'Schließen-bei-exec' auf das niederwertige Bit des ganzzahligen Wertes, der als drittes Argument übergeben wird (`0` oder `1` wie oben).

`F_GETFL` ruft das Dateistatus-Flag für *fildes* ab.

`F_SETFL` setzt das Dateistatus-Flag für *fildes* auf den ganzzahligen Wert, der als drittes Argument übergeben wird. Nur bestimmte Bits können gesetzt werden (siehe `fcntl()`).

Erweiterung

`F_FREESP` gibt Speicherplatz, der mit einem Abschnitt der normalen *fildes*-Datei verbunden ist, frei. Dieser Abschnitt wird von einer Variablen des Datentyps `struct flock`, auf die das dritte Argument *arg* zeigt, spezifiziert. Der Datentyp `struct flock` ist in der Include-Datei `fcntl.h` definiert (siehe `fcntl()`) und beinhaltet folgende Mitglieder:

- `l_whence` ist 0, 1 oder 2, um anzuzeigen, dass der relative Offset `l_start` vom Anfang der Datei, von der momentanen Position oder vom Ende der Datei gemessen wird.
- `l_start` ist der Offset von der Position aus, die in `l_whence` spezifiziert wird. `l_len` ist die Länge dieses Abschnitts. Eine Länge von 0 gibt bis zum Ende der Datei alles frei; in diesem Fall wird das Ende der Datei auf den Anfang des freigegebenen Teils gesetzt. Auf die Daten, die vorher in diesen Teil geschrieben wurden, kann nicht mehr zugegriffen werden.

Die folgenden Kommandos werden für Dateisperren und Datensatzsperren benutzt. Sperren können auf eine ganze Datei oder auf Segmente einer Datei gelegt werden.

`F_SETLK` Eine Sperre in einem der Dateisegmente ist entsprechend der Variablen des Typs `struct flock`, auf die *arg* zeigt, zu setzen oder zu löschen (siehe `fcntl()`). Die Aktion `F_SETLK` wird zum Einrichten der Lesesperre (`F_RDLCK`) und der Schreibsperre (`F_WRLCK`) sowie für die Aufhebung beider Sperrtypen (`F_UNLCK`) verwendet. Lässt sich eine Lese- oder Schreibsperre nicht setzen, gibt `fcntl()` sofort den Fehlerwert -1 zurück.

`F_SETLKW` Dieses *cmd* ist dasselbe wie `F_SETLK`, außer dass der Prozess schläft, bis das Segment frei zum Sperren ist, wenn die Sperranforderung durch andere Sperren blockiert wird.

`F_GETLK` Wenn die durch die `flock`-Struktur angegebene Sperranforderung, auf die *arg* zeigt, erzeugt werden könnte, wird diese Struktur unverändert zurückgegeben, außer dass der Sperrtyp auf `F_UNLCK` und das Feld `l_whence` auf `SEEK_SET` gesetzt wird. Wird eine Sperre gefunden, die eine Erzeugung dieser Sperre verhindern würde, dann wird die Struktur mit der Beschreibung der ersten Sperre überschrieben.

Dieses Kommando erzeugt niemals eine Sperre; es testet nur, ob einzelne Sperren eingerichtet werden könnten.

F_RSETLK, F_RSETLKW, F_RGETLK
: Diese Kommandos werden vom Netzwerkdämon lockd benutzt, um mit dem NFS-Server NFS-Dateien zu sperren.

Eine Lesesperre verhindert, dass ein Prozess den geschützten Bereich mit einer Schreibsperre belegen kann. Für ein bestimmtes Segment einer Datei kann zu einem Zeitpunkt mehr als eine Lesesperre vorhanden sein. Der Dateideskriptor, auf den die Lesesperre gesetzt wird, muss mit dem Leserecht geöffnet worden sein.

Eine Schreibsperre verhindert, dass ein Prozess den geschützten Bereich mit einer Schreib- oder einer Lesesperre belegt. Für ein bestimmtes Segment einer Datei kann zu einem gegebenen Zeitpunkt jeweils nur eine Schreib- oder Lesesperre vorliegen. Der Dateideskriptor, auf den eine Schreibsperre gesetzt wird, muss mit Schreibrecht geöffnet worden sein.

Die Struktur flock beschreibt Typ (l_type), Startpunkt-Offset (l_whence), relativen Offset (l_start), Größe (l_len), Prozessnummer (l_pid) und Systemnummer (l_sysid) des betroffenen Segments in der Datei.

Der Wert von l_whence ist entweder SEEK_SET, SEEK_CUR oder SEEK_END, je nachdem, ob der relative Offset l_start byte vom Anfang der Datei, der aktuellen Position oder dem Ende der Datei gerechnet wird. Der Wert von l_len entspricht der Anzahl der aufeinander folgenden Bytes, die gesperrt werden sollen. Der Wert von l_len kann negativ sein (wenn die Definition von off_t negative Werte für l_len zulässt). Das Feld l_pid wird nur für F_GETLK verwendet, um die Prozessnummer des Prozesses zurückzugeben, der eine blockierende Sperre enthält. Nach einer erfolgreichen F_GETLK-Anforderung, wenn also eine Sperre gefunden wurde, ist der Wert von l_whence SEEK_SET.

Wenn l_len positiv ist, beginnt der entsprechende Bereich bei l_start und endet bei l_start + l_len-1. Wenn l_len negativ ist, beginnt der entsprechende Bereich bei l_start + l_len und endet bei l_start-1. Sperren können jenseits des aktuellen Dateiendes beginnen und auch darüber hinausgehen, dürfen aber in Bezug auf den Dateianfang nicht negativ sein. Eine Sperre erstreckt sich auf den größtmöglichen Wert des Datei-Offsets für diese Datei, wenn l_len auf 0 gesetzt ist. Wenn für eine solche Sperre l_start auch auf 0 und l_whence auf SEEK_SET gesetzt ist, ist die gesamte Datei gesperrt.

Für jedes Byte der Datei wird maximal ein Sperren-Typ gesetzt. Wenn der aufrufende Prozess bereits Sperren für Bytes in dem Bereich hat, der durch die Anforderung angegeben ist, wird vor einer erfolgreichen Rückkehr von einer F_SETLK- oder einer F_SETLKW-Anforderung der vorherige Sperren-Typ für jedes Byte in dem angegebenen Bereich durch den neuen Sperren-Typ ersetzt. Wie weiter oben unter der Beschreibung von gemeinsamen Sperren und exklusiven Sperren angegeben ist, schlägt eine F_SETLK- bzw. eine F_SETLKW-Anforderung fehl oder blockiert, wenn für einen anderen Prozess Sperren für Bytes in dem angegebenen Bereich vorhanden sind und der Typ einer dieser Sperren nicht mit dem Typ in der Anforderung zusammenpasst.

Alle Sperren, die einer Datei für einen bestimmten Prozess zugeordnet sind, werden gelöscht, wenn der Dateideskriptor für diese Datei durch diesen Prozess geschlossen wird oder der Prozess, der den Dateideskriptor enthält, beendet wird. Sperren werden von einem Sohnprozess, der mit der Funktion `fork()` erzeugt wurde, nicht geerbt.

Es besteht die Gefahr eines Deadlocks, wenn ein Prozess, der einen gesperrten Bereich steuert, zeitweise stillgelegt wird, indem versucht wird, den gesperrten Bereich eines anderen Prozesses zu sperren. Wenn das System entdeckt, dass das Stillegen eines Prozesses bis zur Freigabe eines gesperrten Bereichs dazu führen würde, dass sich das Programm aufhängt, so schlägt die Funktion `fcntl()` fehl und gibt den Fehler `EDEADLK` zurück.

Wenn obligatorisches Sperren von Dateien und Dateisätzen in einer Datei aktiv ist (siehe `chmod()`), werden `open()`-, `read()`- und `write()`-Systemaufrufe auf die Datei durch die eingeschalteten Dateisatzsperren beeinflusst.

Folgender zusätzliche Wert kann beim Erstellen von `oflag` verwendet werden:

`O_LARGEFILE` Falls dieser Wert gesetzt ist, ist das in der internen Beschreibung der offenen Datei festgelegte Offset-Maximum der höchste Wert, der in einem Objekt des Typs `off64_t` korrekt dargestellt werden kann.

Das Flag `O_LARGEFILE` kann mit `F_SETFL` aktiviert oder deaktiviert werden.

Das Verhalten folgender Werte entspricht denen von `F_GETLK`, `F_SETLK`, `F_SETLKW` und `F_FREESP`, außer dass hier ein Argument vom Typ `struct flock64` an Stelle eines Arguments vom Typ struct flock übergeben werden muss:

`F_GETLK64, F_SETLK64, F_SETLKW64` und `F_FREESP64`

Die Struktur `flock64` ist wie die von `flock` (siehe <`fcntl()`) definiert, außer das gilt:

`off64_t l_start` und `off64_t l_len.`

Werden Threads verwendet, so wirkt sich die Funktion auf den Prozess oder auf einen Thread wie folgt aus: Beim Kommando `F_SETLKW` wartet der Thread, bis die Anforderung befriedigt werden kann.

Returnwert ein neuer Dateideskriptor
bei erfolgreicher Ausführung des Kommandos `F_DUPFD`.

Wert des Prozess-Statusbytes wie in `fcntl.h` definiert
bei erfolgreicher Ausführung des Kommandos `F_GETFD`.
Der Wert ist nicht negativ.

ein Wert ungleich -1
bei erfolgreicher Ausführung der Kommandos `F_SETFD`, `F_SETFL`, `F_GETLK`, `F_SETLK` und `F_SETLKW`

der Wert 0 bei erfolgreicher Ausführung des Kommandos F_FREESP

der Wert des Datei-Statusbytes und der Zugriffsarten
bei erfolgreicher Ausführung des Kommandos F_GETFL.
Der Wert ist nicht negativ.

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

Fehler fcntl() schlägt fehl, wenn gilt:

EACCES *cmd* ist F_SETLK, der Typ der Sperre (l_type) ist eine Lesesperre (F_RDLCK), und das Segment einer zu sperrenden Datei ist bereits von einem anderen Prozess schreibgeschützt.

Der Typ ist eine Schreibsperre (F_WRLCK), und das Segment einer zu sperrenden Datei wird bereits von einem anderen Prozess lese- oder schreibgeschützt.

EAGAIN *cmd* ist F_FREESP, die Datei existiert, obligatorisches Datei-/Datensatzsperren ist gesetzt, und es gibt noch ausstehende Datensatzsperren in der Datei.

Erweiterung
EAGAIN *cmd* ist F_SETLK oder F_SETLKW, und die Datei wird momentan mit mmap() in den virtuellen Speicher abgebildet.

EBADF *fildes* ist kein gültiger offener Dateideskriptor.

cmd ist F_SETLK oder SETLKW, die Sperre (l_type) ist eine Lesesperre (F_RDLCK), und *fildes* ist kein gültiger, zum Lesen geöffneter Dateideskriptor.

cmd ist F_SETLK oder SETLKW, die Sperre (l_type) ist eine Schreibsperre (F_WRLCK), und *fildes* ist kein gültiger, zum Schreiben geöffneter Dateideskriptor.

cmd ist F_FREESP, und *fildes* ist kein gültiger, zum Schreiben geöffneter Dateideskriptor.

Erweiterung
EDEADLK *cmd* ist F_FREESP, obligatorisches Datensatzsperren ist möglich, O_NDELAY und O_NONBLOCK sind gelöscht, und es wurde eine Situation entdeckt, in der es zu einem Deadlock kommen könnte.

EDEADLK *cmd* ist F_SETLKW, die Sperre ist durch eine Sperre von einem anderen Prozess blockiert, und ein Deadlock würde verursacht, wenn der Prozess angehalten wird, um auf die Aufhebung dieser Sperre zu warten.

Erweiterung

EFAULT *cmd* ist F_FREESP, und der Wert, auf den *arg* zeigt, befindet sich in einer Adresse außerhalb des Adressraums, der vom Prozess belegt wird.

cmd ist F_GETLK, F_SET_LK oder F_SETLKW, und der Wert, auf den *arg* zeigt, befindet sich in einer Adresse außerhalb des Adressraums, der vom Prozess belegt wird.

EINTR Ein Signal wurde während des Systemaufrufs fcntl() abgefangen.

EINVAL *cmd* ist F_DUPFD. *arg* ist entweder negativ, größer oder gleich dem Wert für die maximale Anzahl der jedem Benutzer zur Verfügung stehenden offenen Dateideskriptoren.

cmd besitzt keinen gültigen Wert.

cmd ist F_GETLK, F_SETLK oder SETLKW, und *arg* oder die Daten, auf die verwiesen wird, sind nicht gültig; oder *fildes* gibt eine Datei an, die Sperren nicht unterstützt.

Es wurde versucht, auf eine BS2000-Datei zuzugreifen.

Erweiterung

EIO Während des Lesens oder Schreibens im Dateisystem trat ein Ein-/Ausgabefehler auf.

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

ENOLCK *cmd* ist F_SETLK oder F_SETLKW, der Typ der Sperre ist eine Lese- oder Schreibsperre, und keine weiteren Dateisatzsperren stehen zur Verfügung (zuviele Dateisegmente gesperrt), weil das Maximum des Systems überschritten wurde.

ENOLINK *fildes* ist auf einem fernen Rechner und die Verbindung zu diesem Rechner ist nicht aktiv bzw. *cmd* ist F_FREESP, die Datei auf einem fernen Rechner und die Verbindung dahin nicht aktiv.

EOVERFLOW Einer der zurückgegebenen Werte kann nicht korrekt dargestellt werden.

Hinweis fcntl() wird nur für POSIX-Dateien ausgeführt.

Siehe auch close(), creat(), dup(), exec(), fork(), open(), sigaction(), pipe(), fcntl.h, sys/type.h, unistdt.h.

fcvt - Gleitpunktzahl in Zeichenkette umwandeln

Syntax #include <stdlib.h>

char *fcvt(double *value*, int *ndigit*, int **decpt*, int **sign*);

Beschreibung

Siehe `ecvt()`.

FD_CLR, FD_ISSET, FD_SET, FD_ZERO - Makros für synchrones I/O-Multiplexen

Syntax #include <sys/time.h>

FD_CLR (int *fd*, fd_set **fdset*);

FD_ISSET (int *fd*, fd_set **fdset*);

FD_SET (int *fd*, fd_set **fdset*);

FD_ZERO (fd_set **fdset*);

Beschreibung

Siehe `select()`.

fdelrec - Satz in ISAM-Datei löschen *(BS2000)*

Syntax

```
#include <stdio.h>

int fdelrec(FILE *stream, void *key);
```

Beschreibung

`fdelrec()` löscht aus einer ISAM-Datei mit Satz-Ein-/Ausgabe den Satz mit dem Schlüsselwert *key*.

`FILE` ***stream*** ist der Dateizeiger einer ISAM-Datei, die im Modus `type=record`, `forg=key` geöffnet wurde (siehe auch `fopen()`, `freopen()`).

`void` ***key*** ist der Zeiger auf einen Bereich, der den Schlüsselwert des zu löschenden Satzes in vollständiger Länge oder null enthält. Ist *key* gleich null, wird der zuletzt gelesene Satz gelöscht. Der Satz muss unmittelbar vor dem `fdelrec`-Aufruf gelesen werden.

Returnwert

0	bei Erfolg. Der Satz mit dem angegebenen Schlüssel wurde gelöscht.
> 0	der zu löschende Satz existiert nicht.
`EOF`	bei Fehler.

Hinweis

Wenn der Aufruf fehlerfrei war (Returnwerte 0 bzw. > 0), wird das `EOF`-Flag der Datei zurückgesetzt.

Ist der angegebene Schlüsselwert nicht in der Datei vorhanden (Returnwert > 0), bleibt die aktuelle Position des Lese-/Schreibzeigers unverändert. Einzige Ausnahme: Wenn die Datei zum Zeitpunkt des `fdelrec`-Aufrufs auf den zweiten oder höheren Schlüssel einer Gruppe von Sätzen mit gleichen Schlüsseln positioniert ist, positioniert `fdelrec()` die Datei auf den ersten Satz nach dieser Gruppe.

In ISAM-Dateien mit Schlüsselverdoppelung löscht `fdelrec()` den ersten Satz mit dem angegebenen Schlüssel. Anschließend ist die Datei auf den nächsten Satz (mit gleichem bzw. nächsthöherem) Schlüssel positioniert.

Siehe auch `flocate()`, `fopen()`, `freopen()`, `stdio.h` .

fdetach - Zuordnung zu einer STREAMS-Datei aufheben

Syntax

```
#include <stropts.h>

int fdetach(const char *path);
```

Beschreibung

Die Funktion `fdetach()` hebt die Zuordnung eines Dateideskriptors unter STREAMS zu einem Namen im Dateisystem auf. *path* ist der Pfadname des Objekts (Datei oder Dateiverzeichnis) im Namensraum des Dateisystems, dem vorher der Dateideskriptor mit `fattach()` zugeordnet wurde. Der Benutzer muss der Eigentümer der Datei oder ein Benutzer mit besonderen Rechten sein.

Ein erfolgreicher Aufruf von `fdetach()` hat folgende Auswirkungen: alle Pfadnamen, die die zugeordnete STREAMS-Datei bezeichnet haben, bezeichnen dann wieder das ursprüngliche Objekt, dem die STREAMS-Datei zugeordnet war. Alle nachfolgenden Operationen auf *path* arbeiten mit dem Knoten im Dateisystem und nicht mit der STREAMS-Datei.

Die Zugriffsrechte und der Zustand des Knotens werden so wiederhergestellt, wie sie vor der Zuordnung bestanden.

Alle offenen Datei-Deskriptoren, die eingerichtet wurden, während die STREAMS-Datei der mit *path* bezeichneten Datei zugeordnet war, beziehen sich auch nach derAusführung von `fdetach()` auf die STREAMS-Datei.

Wenn es keine offenen Datei-Deskriptoren oder andere Bezüge auf die STREAMS-Datei gibt, dann wirkt ein erfolgreicher `fdetach()` auf die zugeordnete Datei wie ein letzter `close()`-Aufruf auf diese Datei.

Returnwert

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

Fehler `fdetach()` schlägt fehl, wenn gilt:

`EACCES`	Eine Komponente des Pfades darf nicht durchsucht werden.
`EPERM`	Die effektive Benutzernummer des Prozesses ist nicht die des Eigentümers der von *path* bezeichneten Datei und der Prozess hat nicht die entsprechenden Zugriffsrechte.
ENOTDIR	Eine Komponente des Pfadnamen-Präfix ist kein Dateiverzeichnis.
`ENOENT`	Eine Komponente des Pfadnamens existiert nicht, oder *path* zeigt auf eine leere Zeichenkette.
EINVAL	*path* ist keiner STREAMS-Datei zugeordnet.

ENAMETOOLONG
: Die Länge von *path* überschreitet {PATH_MAX}, oder eine Komponente des Pfadnamens ist länger als{NAME_MAX}, während{_POSIX_NO_TRUNC} aktiv ist.

ELOOP
: Bei der Übersetzung von *path* traten zuviele symbolische Verweise auf.

Siehe auch close(), fattach(), stropts.h.

fdopen - Datenstrom mit Dateideskriptor verbinden

Syntax #include <stdio.h>

FILE *fdopen(int *fildes*, const char **mode*);

Beschreibung

`fdopen()` verbindet einen Datenstrom mit einem Dateideskriptor.

mode ist eine Zeichenkette, die einen der folgenden Werte annehmen kann:

`r` oder `rb`	Datei öffnen zum Lesen
`w` oder `wb`	Datei öffnen zum Schreiben
`a` oder `ab`	Datei öffnen zum Schreiben am Ende der Datei
`r+`, `r+b` oder `rb+`	Datei öffnen zum Aktualisieren (Lesen und Schreiben)
`w+`, `w+b` oder `wb+`	Datei öffnen zum Aktualisieren (Lesen und Schreiben)
`a+` , `a+b` oder `ab+`	Datei öffnen zum Aktualisieren (Lesen und Schreiben) am Ende der Datei

Die Bedeutung dieser Zeichenketten entspricht denen für `fopen()`, außer dass die *mode*-Argumente, die mit `w` beginnen, die Datei nicht auf die Länge 0 kürzen (siehe `fopen()`).

Das Argument *mode* für den Datenstrom darf nur diejenigen Zugriffsarten enthalten, die ursprünglich für die Datei festgelegt worden sind, d.h. eine Änderung der Zugriffsart mit `fdopen()` ist nicht möglich. Der zum Datenstrom gehörende Lese-/Schreibzeiger wird auf die Position des Lese-/Schreibzeigers gesetzt, der mit dem Dateideskriptor verbunden ist.

Die Kennzeichen für Fehler und Dateiende des Datenstroms werden gelöscht. `fdopen()` kann bewirken, dass das Feld `st_atime` der zu Grunde liegenden Datei zum Aktualisieren gekennzeichnet wird.

BS2000
Bei BS2000-Dateien wird das Feld `st_atime` ignoriert. Die Datei behält ihre ursprüngliche Zugriffsart. ❑

Für die automatische Konvertierung darf das `b` für binär in *modus* nicht angegeben werden. Die Umgebungsvariable `IO_CONVERSION` darf nicht vorhanden sein oder muss den Wert `YES` haben.

Returnwert Zeiger auf einen Datenstrom bei Erfolg.

Nullzeiger bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `fdopen()` schlägt fehl, wenn gilt:

`EBADF` *fildes* ist kein gültiger Dateideskriptor.

`EINVAL` bei POSIX-Dateien: *mode* ist kein gültiger Modus.

`EMFILE` `{FOPEN_MAX}`-Datenströme sind bereits für den aufrufenden Prozess geöffnet.

`{STREAM_MAX}`-Datenströme sind bereits für den aufrufenden Prozess geöffnet.

`ENOMEM` Es ist nicht genügend Platz vorhanden, um einen Puffer zuzuweisen.

BS2000
Treten Fehler auf, z.B ein ungültiger Dateideskriptor, liefert `fdopen()` weder ein definiertes Ergebnis noch eine Fehlermeldung. Das Programm bricht in diesem Fall nicht ab. ❑

Hinweis `{STREAM_MAX}` entspricht der Anzahl der Datenströme, die ein Prozess zur selben Zeit geöffnet haben darf und hat denselben Wert wie `{FOPEN_MAX}`, nämlich 2048.

Dateideskriptoren ergeben sich aus Aufrufen wie `open()`, `dup()`, `creat()` oder `pipe()`.

Ob `fdopen()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

Siehe auch `fclose()`, `fopen()`, `open()`, `stdio.h`, Abschnitt „Dateibearbeitung“ auf Seite 74.

feof - Datenstrom auf Dateiendekennzeichen prüfen

Syntax

```
#include <stdio.h>

int feof(FILE *stream);
```

Beschreibung

`feof()` prüft das Dateiendekennzeichen für den Datenstrom, auf den *stream* zeigt.

Returnwert

≠ 0	`EOF` für *stream* ist gesetzt, das Dateiende wurde erreicht.
0	`EOF` ist nicht gesetzt.

Hinweis

`feof()` wird üblicherweise nach Zugriffsfunktionen angewendet, die kein Dateiende anzeigen (`fread()`).

Wenn die Datei nach Erreichen des Dateiendes zurückpositioniert wird (z.B. mit `fseek()`, `fsetpos()`, `rewind()`) oder wenn die Funktion `clearerr()` aufgerufen wird, liefert `feof()` den Wert 0.

Ob `feof()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

BS2000
`feof()` ist sowohl als Makro als auch als Funktion realisiert.

`feof()` ist auch auf Dateien mit Satz-Ein-/Ausgabe unverändert anwendbar. ❑

Siehe auch `clearerr()`, `ferror()`, `fopen()`, `fseek()`, `fsetpos()`, `stdio.h`.

ferror - Datenstrom auf Fehlerkennzeichen prüfen

Syntax #include <stdio.h>

int ferror(FILE **stream*);

Beschreibung

`ferror()` prüft das Fehlerkennzeichen für den Datenstrom, auf den *stream* zeigt.

Returnwert

≠ 0	wenn das Fehlerkennzeichen für *stream* gesetzt ist.
0	wenn das Fehlerkennzeichen für *stream* nicht gesetzt ist.

Hinweis

Das Fehlerkennzeichen bleibt bestehen, bis der zugehörige Dateizeiger freigegeben wird (z.B. durch `rewind()`, `fclose()` oder Programmbeendigung) oder bis die Funktion `clearerr()` aufgerufen wird.

Ob `ferror()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

BS2000

`ferror()` ist sowohl als Makro als auch als Funktion realisiert.

`ferror()` sollte immer dann angewendet werden, wenn aus einer Datei gelesen oder in eine Datei geschrieben wird.

`ferror()` ist auch auf Dateien mit Satz-Ein-/Ausgabe unverändert anwendbar. ❑

Siehe auch `clearerr()`, `feof()`, `fopen()`, `stdio.h`.

fflush - Datenstrom leeren

Syntax #include <stdio.h>

int fflush(FILE **stream*);

Beschreibung

Zeigt *stream* auf einen Ausgabestrom oder einen Aktualisierungsstrom, dessen letzte Operation keine Eingabeoperation war, bewirkt `fflush()`, dass alle gepufferten Daten für diesen Datenstrom in die Datei geschrieben werden. Ist *stream* ein Nullzeiger, führt `fflush()` diese Tätigkeiten für alle geöffneten Dateien durch.

Returnwert

0	bei Erfolg. Der Puffer wurde geleert.
`EOF`	bei Fehler. Der Puffer wurde nicht geleert. `errno` wird gesetzt, um den Fehler anzuzeigen. *BS2000* Oder der Puffer brauchte nicht geleert zu werden, weil er noch nicht existiert (für die Datei ist noch keine Schreibfunktion ausgeführt) oder die Datei ist eine Eingabe- oder INCORE-Datei. ❑ *stream* ist keiner Datei zugeordnet (z.B. weil die Datei bereits geschlossen ist) oder die gepufferten Daten konnten nicht übertragen werden.

Fehler `fflush()` schlägt fehl, wenn gilt:

`EAGAIN`	Das Kennzeichen `O_NONBLOCK` ist für den *stream* zu Grunde liegenden Dateideskriptor gesetzt, und eine Schreiboperation würde den Prozess verzögern.
`EBADF`	Der *stream* Dateideskriptor ist nicht gültig.
`EFBIG`	Es wurde versucht, auf eine Datei zu schreiben, deren Größe die maximale Dateigröße oder die Grenze des Prozesses für die Dateigröße (siehe auch `ulimit()`) überschreitet.
`EINTR`	`fflush()` wurde durch ein Signal unterbrochen.
`EIO`	Ein Ein-/Ausgabefehler ist aufgetreten. Der Prozess ist Mitglied einer Hintergrund-Prozessgruppe und will auf das steuernde Terminal schreiben, `TOSTOP` ist gesetzt, das Signal `SIGTTOU` wird vom Prozess weder ignoriert noch blockiert und die Prozessgruppe des Prozesses ist verwaist.
`ENOSPC`	Auf dem Datenträger, auf dem sich die Datei befindet, ist kein freier Platz mehr vorhanden.

`EPIPE` Es wurde der Versuch unternommen, auf eine Pipe oder FIFO zu schreiben, die von keinem Prozess zum Lesen geöffnet war. Außerdem wird das Signal `SIGPIPE` an den Prozess gesendet.

Werden Threads verwendet, so wirkt sich die Funktion auf den Prozess oder auf einen Thread wie folgt aus: Beim `EPIPE`-Fehler wird das Signal `SIGPIPE` nicht an den Prozess, sondern an den aufrufenden Thread gesendet.

Hinweis Ob `fflush()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

BS2000
Bei allen Standard-Ausgabefunktionen, die Daten in eine BS2000-Datei schreiben (`printf()`, `putc()`, `fwrite()` etc.), werden die Daten in einem Puffer zwischengespeichert und erst in die Datei geschrieben, wenn eines der folgenden Ereignisse eintritt:

– Ein Zeilenendezeichen (`\n`) wird erkannt (nur bei Textdateien).
– Die maximale Satzlänge einer Plattendatei ist erreicht.
– Bei Terminals: Nach einer Ausgabe auf das Terminal folgt eine Eingabe vom Terminal.
– Die Funktionen `fseek()`, `fsetpos()`, `rewind()` oder `fflush()` werden aufgerufen.
– Die Datei wird geschlossen.

Zusätzlich nur bei ANSI-Funktionalität:
Wenn das Lesen aus einer beliebigen Textdatei eine Datenübertragung von der externen Datei in den Puffer notwendig macht, werden die noch in Puffern zwischengespeicherten Daten aller ISAM-Dateien automatisch in die Dateien geschrieben.

Die Pufferung entfällt bei Ausgaben in Zeichenketten (`sprintf()`) und in INCORE-Dateien.
Auch wenn die Daten im Puffer nicht mit einem Zeilenendezeichen enden, bewirkt `fflush()` in einer Textdatei einen Zeilenwechsel. Nachfolgende Daten werden in eine neue Zeile bzw. in einen neuen Satz geschrieben.

Ausnahme bei ANSI-Funktionalität:
Wenn die Daten einer ISAM-Datei im Puffer nicht mit einem Zeilenendezeichen enden, bewirkt `fflush()` keinen Zeilenwechsel bzw. Satzwechsel. Nachfolgende Daten verlängern den Satz in der Datei. Beim Lesen einer ISAM-Datei werden daher nur Zeilenendezeichen eingelesen, die vom Programm explizit geschrieben wurden.

`fflush()` wird intern automatisch ausgeführt, wenn eine Datei geschlossen wird (`fclose()`, `close()`) oder wenn ein Programm normal bzw. mit `exit()` beendet wird. `fflush()` kann dazu benutzt werden, die Ausgabe von Daten während des Programmablaufs zu steuern, z.B. um diverse Eingaben zu einer einzigen Ausgabe zu verketten und zu einem selbst definierten Zeitpunkt auf einmal auszugeben.

Bei Satz-Ein-/Ausgabe wird der Aufruf von `fflush()` zwar nicht mit Fehler abgewiesen, bleibt jedoch ohne Wirkung. Bei Dateien mit Satz-Ein-/Ausgabe werden keine Daten gepuffert. ❑

Siehe auch `exit()`, `close()`, `fclose()`, `stdio.h`.

ffs - erstes gesetztes Bit suchen

Syntax

```
#include <strings.h>

int ffs(int i);
```

Beschreibung

`ffs()` sucht das erste gesetzte Bit im übergebenen Argument, beginnend beim niedrigstwertigen Bit, und liefert die Position dieses Bits zurück. Die Nummerierung der Bits beginnt bei 1, angefangen mit dem niedrigstwertigen Bit.

Returnwert

Position des ersten gesetzten Bits	bei $i \neq 0$.
0	bei $i = 0$.

Siehe auch `strings.h`.

fgetc - Byte aus Datenstrom lesen

Syntax

```
#include <stdio.h>

int fgetc(FILE *stream);
```

Beschreibung

`fgetc()` liest das nächste vorhandene Byte vom Typ `unsigned char` aus dem Datenstrom, auf den *stream* zeigt, wandelt es in den Typ `int` um und schaltet den zum Datenstrom gehörigen Lese-/Schreibzeiger, sofern er definiert ist, entsprechend weiter.

`fgetc()` kann die Strukturkomponente `st_atime` für die Datei, der *stream* zugeordnet ist, zum Ändern markieren (siehe `sys/stat.h`). Die Strukturkomponente `st_atime` wird aktualisiert, sobald `fgetc()`, `fgets()`, `fgetwc()`, `fgetws()`, `fread()`, `fscanf()`, `getc()`, `getchar()`, `gets()` oder `scanf()` erfolgreich für *stream* aufgerufen werden und Daten zurückliefern, die nicht durch einen vorangegangenen Aufruf von `ungetc()` oder `ungetwc()` bereitgestellt wurden.

Returnwert

nächstes Zeichen aus dem Eingabestrom, auf den *stream* zeigt
bei erfolgreicher Beendigung.

`EOF` wenn der Datenstrom das Dateiende erreicht hat. Das Dateiendekennzeichen des Datenstroms wird gesetzt.

`EOF` wenn ein Lesefehler auftritt. Das Fehlerkennzeichen des Datenstroms wird gesetzt. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler

`fgetc()` schlägt fehl, wenn gilt:

`EAGAIN` Das Flag `O_NONBLOCK` wird für den Dateideskriptor gesetzt, der *stream* zu Grunde liegt, und der Prozess würde durch `fgetwc()` angehalten werden.

`EBADF` Der *stream* zu Grunde liegende Dateideskriptor ist kein gültiger, zum Lesen geöffneter Dateideskriptor.

`EINTR` Die Leseoperation wurde durch den Empfang eines Signals beendet. Es wurden keine Daten übertragen.

`EIO` Ein physikalischer Ein-/Ausgabefehler ist aufgetreten,
oder der Prozess ist Mitglied einer Hintergrund-Prozessgruppe und versucht von seinem steuernden Terminal zu lesen. Das Signal `SIGTTIN` wird vom Prozess entweder blockiert oder ignoriert, oder die Prozessgruppe ist verwaist.

Hinweis

Wenn der ganzzahlige Returnwert von `fgetc()` in einer Variablen vom Typ `char` abgelegt und dann mit der ganzzahligen Konstante `EOF` verglichen wird, kann es sein, dass dieser Vergleich nicht erfolgreich ist, da die Vorzeichen-Erweiterung eines Zeichens bei der Umwandlung in eine Ganzzahl rechnerabhängig ist. Daher sollte eine portable Anwendung immer eine `int`-Variable für das Ergebnis von `fgetc()` verwenden.

Wenn zwischen einer Fehlerbedingung und einer Dateiendebedingung unterschieden werden soll, müssen `ferror()` oder `feof()` verwendet werden.

Wenn in einem Programm der folgende Vergleich verwendet wird, muss die Variable *c* als `int`-Größe vereinbart werden:

```
while((c = fgetc(dz)) != EOF)
```

Wenn nämlich *c* als `char`-Größe definiert werden würde, würde die Bedingung `EOF` aus folgendem Grund nie erfüllt: -1 wird in den `char`-Wert 0xFF (also +255) konvertiert. `EOF` ist jedoch -1.

Wenn `fgetc()` in der POSIX-Umgebung von `stdin` liest und `EOF` das Einlese-Endekriterium ist, erreicht man die `EOF`-Bedingung durch folgende Maßnahmen:

- am blockorientierten Terminal durch Eingabe der Tastensequenz @ @ d
- am zeichenorientierten Terminal durch Eingabe von CTRL + D

BS2000
Wenn `fgetc()` in der BS2000-Umgebung von `stdin` liest und `EOF` das Einlese-Endekriterium ist, erreicht man die `EOF`-Bedingung durch folgende Maßnahmen am Terminal:

1. K2 drücken.
2. Die Systemkommandos `EOF` und `RESUME-PROGRAM` eingeben. ❑

Ob `fgetc()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

Siehe auch `feof()`, `ferror()`, `fopen()`, `getchar`, `getc()`, `stdio.h`, `sys/stat.h`.

fgetpos - aktuellen Wert des Lese-/Schreibzeigers im Datenstrom ermitteln

Name **fgetpos, fgetpos64**

Syntax

```
#include <stdio.h>

int fgetpos(FILE *stream, fpos_t *pos);
int fgetpos64(FILE *stream, fpos64_t *pos);
```

Beschreibung

`fgetpos()` speichert den aktuellen Wert des Lese-/Schreibzeigers von *stream* in dem Objekt, auf das *pos* zeigt. Der gespeicherte Wert enthält Informationen, mit denen `fsetpos()` den Datenstrom auf die Position einstellen kann, die zurzeit des Aufrufs von `fgetpos()` aktuell war.

Es besteht kein funktioneller Unterschied zwischen `fgetpos()` und `fgetpos64()`, außer dass `fgetpos64()` einen `fpos64_t`-Datentyp verwendet.

Returnwert

0	bei Erfolg.
≠ 0	bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen. *BS2000* `errno` wird auf `EBADF` gesetzt.

Fehler `fgetpos()` schlägt fehl, wenn gilt:

`EBADF`	Der *stream* zu Grunde liegende Dateideskriptor ist nicht gültig.
`ESPIPE`	Der *stream* zu Grunde liegende Dateideskriptor ist einer Pipe oder FIFO zugeordnet.

Hinweis Ob `fgetpos()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

BS2000
`fgetpos()` lässt sich auf Binärdateien (SAM im Binärmodus, PAM, INCORE) und Textdateien (SAM im Textmodus, ISAM) anwenden.
`fgetpos()` ist nicht anwendbar auf Systemdateien (SYSDTA, SYSLST, SYSOUT).

Für ISAM-Dateien ist das Funktionspaar `fgetpos()`/`fsetpos()` wesentlich performanter als das vergleichbare Funktionspaar `ftell()`/`fseek()`.

Bei Satz-Ein-/Ausgabe liefert `fgetpos()` die Position hinter dem zuletzt gelesenen, geschriebenen oder gelöschten Satz bzw. die Position, die durch ein unmittelbar vorangegangenes Positionieren erreicht wurde.

Bei ISAM-Dateien mit Schlüsselverdoppelung liefert `fgetpos()` immer die Position hinter dem letzten Satz einer Gruppe mit gleichen Schlüsseln, wenn einer dieser Sätze zuvor gelesen, geschrieben oder gelöscht wurde.

Siehe auch `fseek()`, `fseek64()`, `lseek()`, `lseek64()`, `fsetpos()`, `fsetpos64()`, `ftell()`, `ftell64()`, `ungetc()`, `stdio.h`.

fgets - Zeichenkette aus Datenstrom lesen

Syntax #include <stdio.h>

char *fgets(char **s*, int *n*, FILE **stream*);

Beschreibung

`fgets()` liest aus dem Datenstrom, auf den *stream* zeigt, höchstens *n*-1 Bytes bis zum Zeilenendezeichen oder bis zum Dateiende. Die eingelesene Zeichenkette wird in den Vektor eingetragen, auf den *s* zeigt, und mit dem Nullbyte beendet.

`fgets()` kann die Strukturkomponente `st_atime` für die Datei, der *stream* zugeordnet ist, zum Ändern markieren (siehe `sys/stat.h`). Die Strukturkomponente `st_atime` wird aktualisiert, sobald `fgetc()`, `fgets()`, `fgetwc()`, `fgetws()`, `fread()`, `fscanf()`, `getc()`, `getchar()`, `gets()` oder `scanf()` erfolgreich für *stream* aufgerufen werden und Daten zurückliefern, die nicht durch einen vorangegangenen Aufruf von `ungetc()` oder `ungetwc()` bereitgestellt wurden.

Returnwert

Zeiger auf die Ergebniszeichenkette	bei erfolgreicher Beendigung.
Nullzeiger	wenn der Datenstrom das Dateiende erreicht hat. Das Dateiendekennzeichen dieses Datenstroms wird gesetzt.
Nullzeiger	wenn ein Lesefehler auftritt. Das Fehlerkennzeichen des Datenstroms wird gesetzt. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler Siehe `fgetc()`.

Hinweis Der Bereich, in den `fgets()` die gelesene Zeichenkette abspeichern soll, muss explizit bereitgestellt werden.

Im Unterschied zu `gets()` trägt `fgets()` auch ein gelesenes Zeilenendezeichen in die Ergebniszeichenkette ein.

Ob `fgets()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

Beispiel Siehe `fputs()`.

Siehe auch `fgetc()`, `fopen()`, `fputs()`, `fread()`, `gets()`, `stdio.h`, `sys/stat.h`.

fgetwc - Langzeichen aus Datenstrom lesen

Syntax

```
#include <wchar.h>

Optional
#include <stdio.h> ❑

wint_t fgetwc(FILE *stream);
```

Beschreibung

`fgetwc()` liest das nächste Zeichen aus dem Eingabestrom, auf den *stream* zeigt, wandelt es in den entsprechenden Langzeichenwert um und bewegt den Lese-/Schreibzeiger für den Datenstrom, falls definiert, weiter.

Wenn ein Fehler auftritt, ist der Wert des Lese-/Schreibzeigers für den Datenstrom nicht definiert.

`fgetwc()` kann die Strukturkomponente `st_atime` für die Datei, der *stream* zugeordnet ist, zum Ändern markieren (siehe `sys/stat.h`). Die Strukturkomponente `st_atime` wird aktualisiert, sobald `fgetc(), fgets(), fgetwc(), fgetws(), fread(), fscanf(), getc(), getchar(), gets()` oder `scanf()` erfolgreich für *stream* aufgerufen werden und Daten zurückliefern, die nicht durch einen vorangegangenen Aufruf von `ungetc()` oder `ungetwc()` bereitgestellt wurden.

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`).

Returnwert

Langzeichenwert vom Typ `wint_t` bei erfolgreicher Beendigung.

`WEOF` wenn der Datenstrom am Dateiende angelangt ist. Das Dateiendekennzeichen für den Datenstrom wird gesetzt.

`WEOF` wenn ein Lesefehler auftritt. Die Fehleranzeige für den Datenstrom wird gesetzt. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler

`fgetwc()` schlägt fehl, wenn gilt:

`EAGAIN` Das Flag `O_NONBLOCK` wird für den Dateideskriptor gesetzt, der *stream* zu Grunde liegt, und der Prozess würde durch `fgetwc()` angehalten werden.

`EBADF` Der *stream* zu Grunde liegende Dateideskriptor ist kein gültiger, für das Lesen geöffneter Dateideskriptor.

`EINTR` Die Leseoperation wurde durch den Empfang eines Signals beendet. Es wurden keine Daten übertragen.

Erweiterung

`EINVAL` Es wurde versucht, auf eine BS2000-Datei zuzugreifen. ❑

`EIO` Der Prozess ist Mitglied in einer Hintergrund-Prozessgruppe und versucht, von seinem steuernden Terminal zu lesen. Das Signal `SIGTTIN` wird vom Prozess entweder blockiert oder ignoriert, oder die Prozessgruppe ist verwaist.

Hinweis In dieser Version des Laufzeitsystems werden die Langzeichen-Funktionen nur für POSIX-Dateien unterstützt.

`ferror()` bzw. `feof()` müssen verwendet werden, um zwischen einer Fehlerbedingung und einer Dateiendebedingung zu unterscheiden.

Siehe auch `feof()`, `ferror()`, `fgetc()`, `fopen()`, `stdio.h`, `wchar.h`.

fgetws - Langzeichenkette aus Datenstrom lesen

Syntax

#include <wchar.h>

Optional
#include <stdio.h> ❑

wchar_t *fgetws(wchar_t **ws*, int *n*, FILE **stream*);

Beschreibung

`fgetws()` liest Zeichen von *stream*, wandelt sie in die entsprechenden Langzeichenwerte um und legt sie im Vektor *ws* vom Typ `wchar_t` ab, bis *n*-1 Zeichen gelesen wurden, ein Zeilenendezeichen gelesen, konvertiert und an *ws* übertragen wird bzw. eine Dateiende-Bedingung angetroffen wird. Die Langzeichenkette *ws* wird mit einem Nullbyte-Langzeichen abgeschlossen.

Wenn ein Fehler auftritt, ist der Wert des Lese-/Schreibzeigers für den Datenstrom nicht definiert.

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

`fgetws()` kann die Strukturkomponente `st_atime` für die Datei, der *stream* zugeordnet ist, zum Ändern markieren (siehe `sys/stat.h`). Die Strukturkomponente `st_atime` wird aktualisiert, sobald `fgetc()`, `fgets()`, `fgetwc()`, `fgetws()`, `fread()`, `fscanf()`, `getc()`, `getchar()`, `gets()` oder `scanf()` erfolgreich für *stream* aufgerufen werden und Daten zurückliefern, die nicht durch einen vorangegangenen Aufruf von `ungetc()` oder `ungetwc()` bereitgestellt wurden.

Returnwert

ws	bei erfolgreicher Beendigung.
Nullzeiger	wenn der Datenstrom am Dateiende angelangt ist. Das Dateiendekennzeichen für den Datenstrom wird gesetzt.
Nullzeiger	wenn ein Lesefehler auftritt. Das Fehlerkennzeichen für den Datenstrom wird gesetzt. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler

Siehe `fgetwc()`.

Siehe auch

`fgetwc()`, `fopen()`, `fread()`, `stdio.h`, `wchar.h`.

_ _FILE_ _ - Makro für Quelldateinamen

Syntax _ _FILE_ _

Beschreibung

Dieses Makro generiert den Dateinamen des Quellprogramms als Zeichenkette in der Form:

"*name*\0"

Hinweis Dieses Makro muss in keiner Include-Datei definiert werden. Sein Name wird vom Compiler erkannt und ersetzt.

fileno - Dateideskriptor ermitteln

Syntax #include <stdio.h>

int fileno(FILE **stream*);

Beschreibung

`fileno()` gibt den ganzzahligen Dateideskriptor zurück, der zu *stream* gehört.

Returnwert

int-Wert	bei Erfolg. Wert des Dateideskriptors, der zu *stream* gehört.
-1	bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `fileno()` schlägt fehl, wenn gilt:

EABDF	*stream* ist kein gültiger Datenstrom.

Hinweis Ob `fileno()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

Siehe auch `fdopen()`, `fopen()`, `stdin()`, `stdio.h`, Abschnitt „Interaktion von Dateideskriptoren und Datenströmen“ auf Seite 79.

flocate - Lese-/Schreibzeiger in ISAM-Datei positionieren *(BS2000)*

Syntax

```
#include <stdio.h>

int flocate(FILE *stream, void *key, size_t keylen, int option);
```

Beschreibung

`flocate()` dient zum expliziten Positionieren einer ISAM-Datei mit Satz-Ein-/Ausgabe. `flocate()` ändert die aktuelle Position des Lese-/Schreibzeigers der Datei mit Dateizeiger *stream* entsprechend den Angaben:
Schlüsselwert *key*,
Schlüssellänge *keylen* und
Option *option* (`_KEY_FIRST, _KEY_LAST, _KEY_EQ, _KEY_GE`).

`FILE` ***stream*** ist der Dateizeiger einer ISAM-Datei, die im Modus `type=record,forg=key` geöffnet wurde (siehe `fopen()`, `freopen()`).

`void` ***key*** ist der Zeiger auf einen Bereich, der den Schlüsselwert enthält.

`size_t` *keylen* ist die Länge des Schlüsselwertes. Der Wert muss ungleich null sein.

Ist *keylen* kleiner als die Schlüssellänge der Datei, füllt `flocate()` den Schlüsselwert intern bis auf die Schlüssellänge der Datei mit binären Nullen auf und nimmt diesen generierten Schlüssel als Positioniergrundlage.
Ist *keylen* größer als die Schlüssellänge der Datei, schneidet `flocate()` den Schlüsselwert intern von rechts bis auf die Schlüssellänge der Datei ab und nimmt diesen verkürzten Schlüssel als Positioniergrundlage.

`int` *option* kann die folgenden in `stdio.h` definierten Werte enthalten:

`_KEY_FIRST`	positioniert auf den Dateianfang. Die Parameter *key* und *keylen* werden ignoriert.. Das Positionieren ist auch in leeren Dateien erfolgreich.
`_KEY_LAST`	positioniert auf das Dateiende. Die Parameter *key* und *keylen* werden ignoriert. Das Positionieren ist auch in leeren Dateien erfolgreich.
`_KEY_EQ`	positioniert auf den ersten Satz mit dem angegebenen Schlüssel *key*.
`_KEY_GE`	positioniert auf den ersten Satz mit dem Schlüsselwert größer oder gleich dem angegebenen Schlüssel *key*.

Returnwert

0	bei Erfolg. Der Satz mit dem angegebenen Schlüssel existiert.
> 0	der Satz existiert nicht.
`EOF`	bei Fehler.

Hinweis War der Aufruf fehlerfrei (Returnwerte 0 bzw. > 0), wird das Kennzeichen `EOF` der Datei zurückgesetzt.

Ist der angegebene Schlüsselwert nicht in der Datei vorhanden (Returnwert > 0), bleibt die aktuelle Position des Lese-/Schreibzeigers unverändert. Einzige Ausnahme: Wenn die Datei zum Zeitpunkt des `flocate`-Aufrufs auf den zweiten oder höheren Schlüssel einer Gruppe von Sätzen mit gleichen Schlüsseln positioniert ist, positioniert `flocate()` die Datei auf den ersten Satz nach dieser Gruppe.

In ISAM-Dateien mit Schlüsselverdoppelung kann mit `flocate()` nicht auf den zweiten oder höheren Satz einer Gruppe mit gleichen Schlüsseln positioniert werden. Dies lässt sich nur durch sequenzielles Lesen bzw. Löschen erreichen.
Mit `flocate()` kann nur auf den ersten Satz oder hinter den letzten Satz einer solchen Gruppe positioniert werden.

Siehe auch `fdelrec()`, `fgetpos()`, `fsetpos()`, `fopen()`, `freopen()`, `stdio.h`.

flockfile, ftrylockfile, funlockfile - Funktionen zum Sperren der Standardein/-ausgabe

Syntax

```
#include <stdio.h>

void flockfile(FILE *file);

int ftrylockfile(FILE *file);

void funlockfile(FILE *file);
```

Beschreibung

Die Funktionen `flockfile()` und `ftrylockfile()` ermöglichen ein explizites Sperren von (FILE*)-Objekten auf Anwendungsebene. Mit `funlockfile()` kann die Sperre wieder aufgehoben werden. Diese Funktionen können von einem Thread verwendet werden, um eine Folge von E/A-Anweisungen darzustellen, die als Einheit ausgeführt werden sollen.

Die Funktion `flockfile()` wird von einem Thread verwendet, um das Zugriffsrecht auf ein (FILE*)-Objekt zu erlangen.

Die Funktion `ftrylockfile()` wird von einem Thread verwendet, um das Zugriffsrecht auf ein (FILE*)-Objekt zu erlangen, wenn das Objekt verfügbar ist; `ftrylockfile()` ist eine Version von `flockfile()`, bei der keine Blockierung erfolgt.

Die Funktion `funlockfile()` wird verwendet, um das an den Thread vergebene Zugriffsrecht aufzuheben. `funlockfile()` wird ignoriert, wenn der aufrufende Thread nicht der Besitzer des (FILE*)-Objekts ist.

Logisch ist jedem (FILE*)-Objekt ein Sperrenzähler zugeordnet. Dieser Zähler wird implizit mit dem Wert 0 initialisiert, wenn das (FILE*)- Objekt erstellt wird. Die Sperre für das (FILE*)-Objekt wird aufgehoben, wenn der Zähler den Wert 0 hat.
Wenn der Zähler positiv ist, ist ein einzelner Thread Eigentümer des (FILE*)-Objekts. Wird die Funktion `flockfile()` aufgerufen, wenn der Zähler 0 ist oder einen positiven Wert hat und der Aufrufer Eigner des (FILE*)-Objekts ist, so wird der Zähler erhöht. Andernfalls wird der aufrufende Thread unterbrochen und wartet, dass der Zähler wieder den Wert 0 erhält. Jeder Aufruf von `funlockfile()` vermindert den Zähler. Dies ermöglicht das Verschachteln zusammengehöriger Aufrufe von `flockfile()` [oder erfolgreicher Aufrufe von `ftrylockfile()`] und `funlockfile()`.

Alle Funktionen, die auf (FILE*)-Objekte verweisen, verhalten sich, als ob sie intern `flockfile()` und `funlockfile()` verwendeten, um das Zugriffsrecht auf diese (FILE*)-Objekte zu erhalten.

Returnwert `flockfile()` und `funlockfile()`:

Kein Returnwert

`ftrylock()`:

0	bei Erfolg.
≠0	wenn keine Sperre aktiviert werden kann.

Hinweis Bei Echtzeitanwendungen kann es durch die Verwendung von FILE-Sperren zur Umkehrung von Prioritäten kommen. Das Problem tritt auf, wenn ein Thread hoher Priorität ein FILE-Objekt „sperrt“, das gerade von einem Thread niedriger Priorität „entsperrt“ wird, aber der Thread niedriger Priorität wird von einem Thread mittlerer Priorität vorzeitig angehalten. Diese Situation führt zur Umkehrung der Prioritäten; ein Thread hoher Priorität wird von Threads niedrigerer Priorität für unbegrenzte Zeit blockiert.

Entwickler von Echtzeitanwendungen müssen beim Systemdesign die Möglichkeit derartiger Umkehrungen von Prioritäten berücksichtigen. Sie können eine Reihe von Gegenmaßnahmen gegen derartige Situationen treffen, indem beispielsweise kritische Codeabschnitte, die durch FILE-Sperren geschützt werden, mit hoher Priorität ausgeführt werden, so dass ein Thread während der Ausführung kritischer Codeabschnitte nicht vorzeitig angehalten werden kann.

Siehe auch `getc_unlocked()`, `pthread_intro()`, `stdio()`.

floor, floorf, floorl- Gleitpunktzahl abrunden

Syntax
```
#include <math.h>
double floor(double x);
float floorf(float x)
long double floorl(long double)
```

Beschreibung
`floor()` rundet die Gleitpunktzahl *x* nach unten ganzzahlig ab.

Returnwert Größte ganze Zahl vom Typ `double`, die kleiner oder gleich *x* ist bei Erfolg.

`-HUGE_VAL`	bei Überlauf. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `floor()`, `floorf()`, und `floorl()` schlagen fehl, wenn gilt:

`ERANGE`	Überlauf, das Resultat ist zu groß.

Hinweis Der ganzzahlige, als `double` bzw. `float` bzw. `long` dargestellte Wert, den `floor()` bzw. `floorf()` bzw. `floorl()` liefert, kann möglicherweise nicht als `int` oder `long` dargestellt werden. Der Returnwert sollte vor einer Zuweisung an einen ganzzahligen Typ geprüft werden, um undefinierte Resultate eines ganzzahligen Überlaufs zu vermeiden.

Siehe auch `ceil()`, `ceilf()`, `ceill()`, `fabs()`, `math.h`.

fmod - Divisionsrest einer Gleitpunktzahl berechnen

Syntax
```
#include <math.h>
double fmod(double x, double y);
```

Beschreibung
`fmod()` berechnet den Rest der Division *x*/*y*. Der Rest hat das gleiche Vorzeichen wie der Dividend *x* und sein Absolutbetrag ist immer kleiner als der Divisor *y*.

Returnwert

Rest der Division *x*/*y*	bei Erfolg.
0	bei $y = 0$.

Hinweis Eine Anwendung sollte sicherstellen, dass *y* ungleich 0 ist, bevor sie `fmod()` aufruft.

Siehe auch `ceil()`, `ceilf()`, `ceill()`, `fabs()`, `floor()`, `math.h`.

fmtmsg - Meldung auf stderr und/oder die Systemkonsole ausgeben

Syntax #include <fmtmsg.h>

int fmtmsg(long *classification*, const char **label*, int *severity*, const char **text*,
const char **action*, const char **tag*);

Beschreibung

Aufbauend auf der Klassifikationskomponente einer Meldung, schreibt `fmtmsg()` eine formatierte Meldung auf `stderr`, auf die Systemkonsole oder auf beide.

`fmtmsg()` kann an Stelle der üblichen `printf()` Schnittstelle verwendet werden, um Meldungen über `stderr` auszugeben. `fmtmsg()` bietet in Verbindung mit `gettxt()` eine einfache Schnittstelle zum Erstellen von sprachunabhängigen Anwendungsprogrammen.

Eine formatierte Meldung besteht aus bis zu fünf Standardkomponenten, die weiter unten definiert werden. Die Komponente *classification* ist nicht Teil der Standardmeldung, die dem Benutzer angezeigt wird, sondern definiert die Quelle der Meldung und steuert die Anzeige der formatierten Meldung.

classification

enthält Bezeichner aus den folgenden Gruppen der Haupt- und Nebenklassifikationen. Jeder Bezeichner einer Nebenklassifikation kann durch ODER-Verknüpfung mit einem anderen Bezeichner einer anderen Nebenklassifikation verwendet werden. Zwei oder mehr Bezeichner aus derselben Nebenklassifikation sollten nicht zusammen verwendet werden, mit Ausnahme der Anzeigeklassifikation. Beide Bezeichner der Anzeigeklassifikation können so verwendet werden, dass die Meldungen sowohl auf `stderr` als auch auf der Systemkonsole erscheinen.

Hauptklassifikationen

bezeichnen den Ursprung eines Zustands. Die Bezeichner sind: `MM_HARD` (Hardware), `MM_SOFT` (Software) und `MM_FIRM` (Firmware).

Nebenklassifikationen des Meldungsursprungs

bezeichnen die Art der Software, in der das Problem auftrat. Die Bezeichner sind: `MM_APPL` (Anwendung), `MM_UTIL` (Hilfsprogramm) und `MM_OPSYS` (Betriebssystem).

Nebenklassifikationen für die Anzeige

bezeichnen, wo die Meldung angezeigt werden soll. Die Bezeichner sind `MM_PRINT`, um die Meldung auf der Standard-Fehlerausgabe auszugeben, und `MM_CONSOLE`, um die Meldung auf der Systemkonsole auszugeben. Sie können einen oder beide Bezeichner verwenden oder die Angabe weglassen (in diesem Falle wird nichts ausgegeben).

Nebenklassifikationen für den Status
: geben an, ob sich das Anwendungsprogramm nach dem Zustand stabilisieren kann. Bezeichner sind: `MM_RECOVER` (stabilisierbar) und `MM_NRECOV` (nicht stabilisierbar).

Zusätzlicher Bezeichner `MM_NULLMC`
: gibt an, dass keine Klassifikationskomponente für die Meldung angegeben wird.

label gibt den Ursprung der Meldung an. Das Format dieser Komponente besteht aus zwei Feldern, die durch einen Doppelpunkt getrennt werden. Das erste Feld ist bis zu 10 Zeichen lang; das zweite ist bis zu 14 Zeichen lang.

Es wird dazu geraten, mit *label* das Paket und das Programm oder den Anwendungsnamen zu bezeichnen. So zeigt beispielsweise der Inhalt `UX:cat` für *label* an, dass das Paket UNIX-System V und die Anwendung `cat` gemeint ist.

severity zeigt die Warnstufe des Zustands an. Bezeichner für die Warnstufen für *severity* sind:

`MM_HALT` zeigt an, dass die Anwendung auf einen schwer wiegenden Fehler gestoßen ist und die Bearbeitung anhält. Die Zeichenkette „`HALT`“ wird ausgegeben.

`MM_ERROR` zeigt an, dass die Anwendung einen Fehler erkannt hat. Die Zeichenkette „`ERROR`“ wird ausgegeben.

`MM_WARNING`
: zeigt an, dass ein ungewöhnlicher Zustand eingetreten ist, bei dem es sich um ein Problem handeln könnte und der beobachtet werden sollte. Die Zeichenkette „`WARNING`“ wird ausgegeben.

`MM_INFO` liefert Informationen über einen Zustand, der keinen Fehler darstellt. Die Zeichenkette „`INFO`“ wird ausgegeben.

`MM_NOSEV` zeigt an, dass für die Meldung keine Warnstufe existiert.

text beschreibt die Ursache der Meldung. Die Zeichenkette *text* ist nicht auf eine bestimmte Länge beschränkt. Wenn die Zeichenkette leer ist, ist der ausgegebene Text undefiniert.

action beschreibt die erste Aktion, die im Fehlerbehebungsprozess ausgeführt werden soll. `fmtmsg()` schreibt vor dieser Zeichenkette das Präfix „`TO FIX:`“. Die Zeichenkette *action* ist nicht auf eine bestimmte Länge beschränkt.

tag Ein Bezeichner, der auf die Online-Dokumentation für die Meldung verweist. Empfohlen wird, dass *tag* den über *label* angesprochenen Ursprung der Meldung und eine eindeutige Zahl enthält. Ein Beispiel für *tag* ist `UX:cat:146`.

Umgebungsvariablen

Es gibt zwei Umgebungsvariablen, die das Verhalten von `fmtmsg()` beeinflussen: `MSGVERB` und `SEV_LEVEL`.

`MSGVERB` teilt `fmtmsg()` mit, welche Meldungskomponenten beim Schreiben der Meldungen auf `stderr` ausgewählt werden sollen. Der Wert von `MSGVERB` besteht aus einer Liste optionaler Schlüsselwörter, die durch Doppelpunkte getrennt werden. `MSGVERB` kann wie folgt gesetzt werden:

`MSGVERB=[`*Schlüsselwort*`[:`*Schlüsselwort*`[:...]]]`
`export MSGVERB`

Gültige *Schlüsselwörter* sind: `label`, `severity`, `text`, `action` und `tag`.
Wenn `MSGVERB` ein Schlüsselwort für eine Komponente enthält und diese Komponente nicht den ihr zugeordneten Nullwert hat (siehe unten), gibt `fmtmsg()` diese Komponente bei der Meldungsausgabe auf `stderr` aus. Wenn `MSGVERB` das Schlüsselwort für eine Meldungskomponente nicht enthält, wird diese Komponente nicht ausgegeben. Die Schlüsselwörter können in einer beliebigen Reihenfolge angegeben werden. Ist `MSGVERB` nicht definiert, enthält dieser Bezeichner eine Nullzeichenkette, ist der Wert nicht im korrekten Format angegeben, oder sind ungültige Schlüsselwörter angegeben, so wählt `fmtmsg()` alle Komponenten aus.

Beim ersten Aufruf von `fmtmsg()` wird die `MSGVERB`-Umgebungsvariable abgeprüft, um die Meldungskomponenten selektieren zu können, wenn eine Meldung über die Standard-Fehlerausgabe `stderr` generiert wird. Die Werte, die beim ersten Aufruf akzeptiert werden, werden für die nachfolgenden Aufrufe gesichert.

`MSGVERB` beeinflusst nur die Selektion der Komponenten, die über die Standard-Fehlerausgabe angezeigt werden sollen. Bei Ausgabe auf die Konsole werden alle Meldungen selektiert.

`SEV_LEVEL` definiert die Warnstufen und weist die auszugebenden Zeichenketten zu, die von `fmtmsg()` benutzt werden sollen. Die unten angegebenen Standardwarnstufen können nicht verändert werden. Weitere Warnstufen können definiert, verändert und entfernt werden. Dies geschieht über die Funktion `addseverity ()`(siehe `addseverity`(3C)). Wenn dieselbe Warnstufe durch `SEV_LEVEL` und `addseverity()` definiert wird, so setzt sich die Definition von `addseverity()` durch.

```
0   (keine Warnstufe verwendet)
1   HALT
2   ERROR
3   WARNING
4   INFO
```

`SEV_LEVEL` kann wie folgt eingestellt werden:

```
SEV_LEVEL=[Beschreibung[:Beschreibung[:...]]]
export SEV_LEVEL
```

Beschreibung enthält eine Liste mit drei Feldern, die durch Kommata getrennt werden:

Beschreibung=severity_keyword , level , printstring

severity_keyword ist eine Zeichenkette, die als Schlüsselwort für die Option `-s` *severity* vom Kommando `fmtmsg` verwendet wird. Dieses Feld wird nicht von der Funktion `fmtmsg()` verwendet.

level ist eine Zeichenkette, die eine positive ganze Zahl enthält (nicht 0, 1, 2, 3 oder 4, denn diese Werte sind für die Standardwarnstufen reserviert). Wenn das Schlüsselwort *severity_keyword* verwendet wird, stellt *level* die Warnstufe des Wertes dar, der an die Funktion `fmtmsg()` übergeben wurde.

printstring ist eine Zeichenkette, die von `fmtmsg()` für das Standardmeldungsformat verwendet wird, wenn die Warnstufe *level* angegeben wird.

Stellt *Beschreibung* in der Liste keine durch Kommata getrennte Liste mit drei Feldern dar, oder ist das zweite Feld einer Liste keine ganze Zahl, so wird *Beschreibung* in der Liste ignoriert.

Wird `fmtmsg()` erstmals aufgerufen, dann wird die Umgebungsvariable `SEV_LEVEL` überprüft, um festzustellen, ob neben den fünf Standardwarnstufen und den durch `addseverity()` festgelegten zusätzliche Warnstufen definiert wurden. Die Werte, die beim erstmaligen Aufruf festgestellt wurden, werden für spätere Aufrufe gespeichert.

Returnwert

`MM_OK` bei Erfolg.

`MM_NOTOK` Die Funktion ist völlig fehlgeschlagen.

`MM_NOMSG` Die Funktion konnte eine Meldung über die Standard-Fehlerausgabe nicht generieren, wurde aber ansonsten erfolgreich ausgeführt.

`MM_NOCON` Die Funktion konnte eine Meldung über die Systemkonsole nicht generieren, wurde aber ansonsten erfolgreich ausgeführt.

Eine oder mehrere Meldungskomponenten können systematisch aus der Meldung weggelassen werden, wenn der Nullwert der jeweiligen Komponente angegeben wird.
Die folgende Tabelle zeigt die Nullwerte und Bezeichner für die Argumente von `fmtmsg()`.

Argument	Typ	Nullwert	Bezeichner
label	char*	(char*) NULL	MM_NULLLBL
severity	int	0	MM_NULLSEV
class	long	0L	MM_NULLMC
text	char*	(char*) NULL	MM_NULLTXT
action	char*	(char*) NULL	MM_NULLACT
tag	char*	(char*) NULL	MM_NULLTAG

Ein weiteres Mittel zum systematischen Weglassen einer Komponenten besteht im Auslassen der Schlüsselwörter der Komponenten bei der Definition der `MSGVERB`-Umgebungsvariablen.

Beispiel 1
```
fmtmsg(MM_PRINT, "UX:cat", MM_ERROR, "Falsche Syntax",
       "Siehe Handbuch", "UX:cat:001")
```

liefert eine komplette Meldung mit dem Standardmeldungsformat:

```
UX:cat: ERROR: Falsche Syntax TO FIX: Siehe Handbuch UX:cat:001
```

Beispiel 2 Wird die Umgebungsvariable `MSGVERB` wie folgt gesetzt:

```
MSGVERB=severity:text:action
```

und dann Beispiel 1 verwendet, so generiert `fmtmsg()`:

```
ERROR: Falsche Syntax TO FIX: Siehe Handbuch
```

Beispiel 3 Wird die Umgebungsvariable `SEV_LEVEL` wie folgt gesetzt:

```
SEV_LEVEL=note,5,NOTE
```

so liefert der folgende Aufruf von `fmtmsg()`:

```
fmtmsg(MM_UTIL | MM_PRINT, "UX:cat", 5, "Falsche Syntax",
       "Siehe Handbuch", "UX:cat:001")
```

die folgende Ausgabe:

```
UX:cat: NOTE: Falsche Syntax TO FIX: Siehe Handbuch UX:cat:001
```

Siehe auch `printf(). fmtmsg.h`.

fopen - Datenstrom öffnen

Name **fopen, fopen64**

Syntax #include <stdio.h>

FILE *fopen(const char **filename*, const char **mode*);
FILE *fopen64(const char **filename*, const char **mode*);

Beschreibung

`fopen()` öffnet die Datei, deren Pfadname die Zeichenkette ist, die auf *filename* zeigt, und ordnet ihr einen Datenstrom zu.

filename kann sein:

- ein gültiger POSIX-Dateiname
- ein gültiger BS2000-Dateiname:
  - `link=`*linkname*
    linkname bezeichnet einen BS2000-Linknamen.
  - (SYSDTA), (SYSOUT), (SYSLST), die entsprechende Systemdatei
  - (SYSTERM), Terminal-Ein-/Ausgabe
  - (INCORE), temporäre Binärdatei, die nur im virtuellen Speicher angelegt wird.

mode ist eine Zeichenkette, die die gewünschte Zugriffsart angibt und dafür einen der folgenden Werte annehmen:

`r`	Öffnen Textdatei zum Lesen. Die Datei muss bereits vorhanden sein.
`w`	Öffnen Textdatei zum Neuschreiben. Ist die Datei vorhanden, wird der alte Inhalt gelöscht. Ist die Datei nicht vorhanden, wird sie neu erstellt.
`a`	Öffnen Textdatei zum Anfügen ans Ende der Datei. Ist die Datei vorhanden, wird auf das Dateiende positioniert, d.h. der alte Inhalt bleibt erhalten und die neuen Daten werden ans Ende der Datei angefügt. Ist die Datei nicht vorhanden, wird sie neu erstellt.
`rb`	Öffnen Binärdatei zum Lesen. Die Datei muss bereits vorhanden sein.
`wb`	Öffnen Binärdatei zum Neuschreiben. Ist die Datei vorhanden, wird der alte Inhalt gelöscht. Ist die Datei nicht vorhanden, wird sie neu erstellt.
`ab`	Öffnen Binärdatei zum Anfügen ans Ende der Datei. Ist die Datei vorhanden, wird auf das Dateiende positioniert, d.h. der alte Inhalt bleibt erhalten und die neuen Daten werden ans Ende der Datei angefügt. Ist die Datei nicht vorhanden, wird sie neu erstellt.

`r+w`, `r+` Öffnen Textdatei zum Lesen und Schreiben. Die Datei muss bereits vorhanden sein. Der alte Inhalt bleibt erhalten.

`w+r`, `w+` Öffnen Textdatei zum Neuschreiben und Lesen. Ist die Datei vorhanden, wird der alte Inhalt gelöscht. Ist die Datei nicht vorhanden, wird sie neu erstellt.

`a+r`, `a+` Öffnen Textdatei zum Anfügen ans Ende der Datei und zum Lesen. Ist die Datei vorhanden, bleibt der alte Inhalt erhalten und die neuen Daten werden ans Ende der Datei angefügt. Eine bereits vorhandene Datei ist nach dem Öffnen bei KR-Funktionalität (nur bei C/C++ Versionen kleiner V3 vorhanden) auf das Dateiende, bei ANSI-Funktionalität auf den Dateianfang positioniert.
Ist die Datei nicht vorhanden, wird sie neu erstellt.

`r+b`, `rb+` Öffnen Binärdatei zum Lesen und Schreiben. Die Datei muss bereits vorhanden sein. Der alte Inhalt bleibt erhalten.

`w+b`, `wb+` Öffnen Binärdatei zum Neuschreiben und Lesen. Ist die Datei vorhanden, wird der alte Inhalt gelöscht. Ist die Datei nicht vorhanden, wird sie neu erstellt.

`a+b`, `ab+` Öffnen Binärdatei zum Anfügen ans Ende der Datei und zum Lesen.
Ist die Datei vorhanden, bleibt der alte Inhalt erhalten und die neuen Daten werden ans Ende der Datei angefügt. Eine bereits vorhandene Datei ist nach dem Öffnen bei KR-Funktionalität (nur bei C/C++ Versionen kleiner V3 vorhanden) auf das Dateiende, bei ANSI-Funktionalität auf den Dateianfang positioniert.
Ist die Datei nicht vorhanden, wird sie neu erstellt.

Das Zeichen `b` wird in den obigen Zugriffsarten ignoriert. Das Öffnen einer Datei zum Lesen, d.h. mit `r` als erstem Zeichen im Argument *mode*, schlägt fehl, wenn die Datei nicht existiert oder nicht gelesen werden kann.

Wird eine Datei zum Anfügen geöffnet, d.h. mit `a` als erstem Zeichen im Argument *mode*, erfolgen alle nachfolgenden Schreiboperationen in die Datei am aktuellen Dateiende, ohne dass dabei eventuell dazwischen erfolgte Aufrufe von `fseek()` eine Rolle spielen.

Wird eine Datei zum Aktualisieren geöffnet, d.h. mit `+` als zweitem Zeichen im Argument *mode*, dann können auf dem zugeordneten Datenstrom sowohl Ein- als auch Ausgabeoperationen durchgeführt werden. Dennoch darf auf eine Ausgabeoperation nicht unmittelbar eine Eingabeoperation folgen, ohne dass dazwischen ein Aufruf der Funktion `fflush()` oder einer der Funktionen zur Positionierung, `fseek()`, `fsetpos()` oder `rewind()`, erfolgt ist.

Auf eine Eingabeoperation darf nicht unmittelbar eine Ausgabeoperation folgen, ohne dass dazwischen eine der Funktionen zur Positionierung aufgerufen wurde, außer die Eingabeoperation hat das Dateiende erreicht.

Ist ein Datenstrom geöffnet, so ist er genau dann vollständig gepuffert, wenn sichergestellt ist, dass er keine Verweise auf ein interaktives Gerät hat, z.B. Terminal. Die Kennzeichen für Dateiende und Fehler dieses Datenstroms werden gelöscht.

Für die automatische Konvertierung darf das `b` für binär in *modus* nicht angegeben werden. Die Umgebungsvariable `IO_CONVERSION` darf nicht vorhanden sein oder muss den Wert `YES` haben.

BS2000
Wenn BS2000-Dateien ausgeführt werden, ist Folgendes zu beachten:
In *mode* kann zusätzlich zur Zugriffsart eine Angabe zur Behandlung des Tabulatorzeichens (`\t`) gemacht werden. Diese Angabe ist nur für Textdateien mit den Zugriffsmethoden SAM und ISAM relevant.

`"...,tabexp=yes"`
Das Tabulatorzeichen wird in die entsprechende Anzahl Leerzeichen expandiert; Voreinstellung bei KR-Funktionalität (nur bei C/C++ Versionen kleiner V3 vorhanden).

`"...,tabexp=no"`
Das Tabulatorzeichen wird nicht expandiert; Voreinstellung bei ANSI-Funktionalität.

Es besteht kein funktionaler Unterschied zwischen `fopen` und `fopen64`, außer dass `fopen64` einen Dateizeiger zurückgibt, der über die 2GB-Grenze hinausgehen kann. `fopen64()` setzt das `O_LARGEFILE` Bit im File Status Flag.

Returnwert

Dateizeiger	bei Erfolg.
Nullzeiger	wenn auf *filename* nicht zugegriffen werden kann, *mode* ungültig ist oder die Datei nicht geöffnet werden kann. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `fopen()` schlägt fehl, wenn gilt:

`EACCES`	Für eine Komponete des Pfades existiert kein Durchsuchrecht. Die Datei existiert, und die für *mode* geltenden Zugriffrechte werden verweigert. Die Datei existiert nicht, und das übergeordnete Verzeichnis der zu erstellenden Datei hat kein Schreibrecht.
`EINTR`	Während des Systemaufrufs `fopen()` wurde ein Signal abgefangen.
`EINVAL`	Der Wert des Arguments *mode* ist ungültig.
`EISDIR`	Die angegebene Datei ist ein Dateiverzeichnis und *mode* verlangt Schreibrecht.
`EMFILE`	`{OPEN_MAX}`-Dateideskriptoren sind bereits für den aufrufenden Prozess geöffnet.

{FOPEN_MAX}-Datenströme sind bereits für den aufrufenden Prozess geöffnet

{STREAM_MAX}-Datenströme sind bereits für den aufrufenden Prozess geöffnet.

ENAMETOOLONG
: Die Länge von *filename* überschreitet {PATH_MAX}, oder eine Komponente des Pfades ist länger als {NAME_MAX}.

ENFILE
: Die maximale Anzahl von Dateien im System ist bereits geöffnet.

ENOENT
: Die angegebene Datei existiert nicht, oder *filename* zeigt auf die leere Zeichenkette.

ENOMEM
: Es ist nicht ausreichend Speicherplatz vorhanden.

ENOSPC
: Die Datei existiert nicht, und das Dateiverzeichnis, in dem eine neue Datei erstellt werden soll, kann nicht erweitert werden.

ENOTDIR
: Eine Komponente des Pfades ist kein Dateiverzeichnis.

ENXIO
: Die angegebene Datei ist eine Gerätedatei für ein zeichen- oder blockorientiertes Gerät und das dieser Datei zugeordnete Gerät existiert nicht.

EROFS
: Die angegebene Datei befindet sich in einem Dateisystem, das nur Leserecht hat, und *mode* verlangt Schreibrecht.

ETXTBSY
: Bei der Datei handelt es sich um eine reine Prozedurdatei (gemeinsam verwendete Textdatei), die nur ausgeführt wird und für mode ist Schreibzugriff erforderlich.

EOVERFLOW
: Die genannte Datei ist eine reguläre Datei und die Größe der Datei kann in einem Objekt des Typs off_t nicht korrekt dargestellt werden.

Hinweis {STREAM_MAX} entspricht der Anzahl der Datenströme, die ein Prozess zu selben Zeit geöffnet haben darf und hat denselben Wert wie {FOPEN_MAX}, nämlich 2048.

Ob fopen() für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

BS2000
Der BS2000-Dateiname bzw. -Linkname kann in Klein- und Großbuchstaben geschrieben werden; er wird automatisch in Großbuchstaben umgesetzt. Durch die Angabe eines b an zweiter bzw. dritter Stelle im Parameter *mode* wird die Datei als Binärdatei geöffnet. Die Angabe ist nur für SAM-Dateien relevant, da nur SAM-Dateien sowohl im Binär- als auch im Textmodus verarbeitet werden können.

Systemdateien und ISAM-Dateien werden immer als Textdateien verarbeitet. Die Angabe des Binärmodus führt bei diesen Dateien zu einem Fehler beim Öffnen.

(INCORE)- und PAM-Dateien werden immer als Binärdateien verarbeitet. Aus Kompatibilitätsgründen funktioniert das Öffnen als Binärdatei auch ohne explizite Angabe des Binärmodus.

Wird eine nicht vorhandene Datei neu angelegt, dann wird standardmäßig eine Datei mit folgenden Attributen erzeugt:

	Binärdatei	**Textdatei**
Zugriffsmethode	SAM	SAM (KR-Funktionalität, nur bei C/C++ Versionen kleiner V3 vorhanden) ISAM (ANSI-Funktionalität)
Satzformat	F	V

Bei Verwendung eines Linknamens lassen sich mit dem `ADD-FILE-LINK`-Kommando folgende Dateiattribute ändern: Zugriffsmethode, Satzlänge, Satzformat, Blocklänge und Blockformat.

In allen Fällen, in denen der alte Inhalt einer bereits existierenden Datei gelöscht wird (geöffnet zum Neuschreiben bzw. zum Neuschreiben und Lesen), bleiben die Katalogeigenschaften dieser Datei erhalten.

Wenn eine Datei zum Aktualisieren geöffnet wird, kann das Lesen und Schreiben über denselben Dateizeiger erfolgen. Dennoch sollte auf eine Ausgabe nicht unmittelbar eine Eingabe folgen ohne ein vorhergehendes Positionieren (`fseek()`, `fsetpos()`, `rewind()`, oder einen `fflush`-Aufruf. Dasselbe gilt für eine Ausgabe, die einer Eingabe folgt.

Position des Lese-/Schreibzeigers im Anfügemodus

(INCORE)-Dateien können nur zum Neuschreiben (`w`), zum Neuschreiben und Lesen (`w+r`) oder zum Lesen (`r`) geöffnet werden. Es müssen zuerst Daten geschrieben werden. Um die geschriebenen Daten wieder einlesen zu können, gibt es folgende Möglichkeiten: Wurde die Datei nur zum Neuschreiben geöffnet, kann man sie mit der Funktion `freopen()` zum Lesen öffnen. Wurde die Datei zum Neuschreiben und zum Lesen geöffnet, kann man den Lese-/Schreibzeiger mit der Funktion `rewind()` auf den Dateianfang positionieren.

Eine Datei kann gleichzeitig für verschiedene Zugriffsmodi geöffnet werden, sofern diese Modi im BS2000-Datenverwaltungssystem miteinander verträglich sind.

Startet ein Programm, werden ihm automatisch drei Dateizeiger wie folgt zugeordnet:

`stdin` Dateizeiger für Standard-Eingabe (Terminal)

`stdout` Dateizeiger für Standard-Ausgabe (Terminal)

`stderr` Dateizeiger für Standard-Fehlerausgabe (Terminal)

Es können maximal _NFILE-Dateien gleichzeitig geöffnet sein. _NFILE ist in `stdio.h` mit 2048 definiert.

Für das Öffnen von Dateien mit Satz-Ein-/Ausgabe wird *mode* um zwei Angaben erweitert. Diese Angaben folgen in der Zeichenkette hinter der Zugriffsart (s.o.), jeweils durch ein Komma getrennt:

```
"...,type=record [,forg={seq/key}]"
```

`type=record`	Die Datei wird für Satz-Ein-/Ausgabe geröffnet. Fehlt diese Angabe, wird die Datei für Strom-Ein-/Ausgabe geöffnet.
`forg=seq`	Die Dateiorganisation ist sequenziell. Sequenzielle Dateien können SAM- oder PAM-Dateien sein.
`forg=key`	Die Dateiorganisation ist indexsequenziell. Indexsequenzielle Dateien sind ISAM-Dateien.

Fehlt die Angabe von `forg()`, ist die Dateiorganisation vom FCBTYP der Datei abhängig: Der FCBTYP ist durch den Katalogeintrag einer bereits existierenden Datei festgelegt bzw. durch ein `ADD-FILE-LINK`-Kommando. Für SAM- und PAM-Dateien wird sequenzielle Dateiorganisation angenommen, für ISAM-Dateien indexsequenzielle Dateiorganisation.

Fehlt die Angabe von `forg()` und der FCBTYP ist nicht festgelegt (Datei nicht vorhanden, kein `ADD-FILE-LINK`-Kommando), wird sequenzielle Dateiorganisation angenommen und eine SAM-Datei erstellt.

Für die Satz-Ein-/Ausgabe gelten nachstehende Einschränkungen; werden diese Einschränkungen nicht eingehalten, wird die Datei nicht geöffnet und ein Fehler-Returnwert geliefert:

Die Datei muss im Binärmodus geöffnet werden (Angabe `b` bei der Zugriffsart).

`type=record` ist zulässig für SAM-, PAM- oder ISAM-Dateien.

`forg=seq` ist zulässig für SAM- oder PAM-Dateien, `forg=key` für ISAM-Dateien.

Bei ISAM-Dateien ist der Anfügemodus `a` unzulässig. Die Position wird aus dem Schlüssel im Satz bestimmt.

Siehe auch `creat()`, `fclose()`, `fdopen()`, `ferror()`, `freopen()`, `open()`, `stdio.h`.

fork - neuen Prozess erzeugen

Syntax

```
#include <unistd.h>

Optional
#include <sys/types.h> ❑

pid_t fork(void);
```

Beschreibung

`fork()` erzeugt einen neuen Prozess. Der neue Prozess (Sohnprozess) ist in Bezug auf die folgenden Punkte eine exakte Kopie des aufrufenden Prozesses (Vaterprozess):

- reale und effektive Benutzer- und Gruppennummern
- Umgebung
- sbe-Bit (siehe `exec()`)
- Signalaktionen (`SIG_DFL`, `SIG_IGN`, Adresse der Signalbehandlungsfunktion)
- zusätzliche Gruppennummern
- s-Bit für Eigentümer
- s-Bit für Gruppe
- Prioritätswert (siehe `nice()`)
- alle zugeteilten gemeinsam nutzbaren Speichersegmente (siehe `shmop()`)
- Prozessgruppennummer
- Terminalnummer (siehe `exit()`)
- aktuelles Dateiverzeichnis
- Root-Dateiverzeichnis
- Schutzbitmaske (siehe `umask()`)
- Betriebsmittel-Grenzwerte (siehe `getrlimit()`)
- steuerndes Terminal

Der Sohnprozess unterscheidet sich vom Vaterprozess in folgenden Punkten:

- Der Sohnprozess besitzt eine eigene eindeutige Prozessnummer. Die Sohnprozessnummer entspricht keiner aktiven Prozessgruppennummer.
- Der Sohnprozess besitzt zusätzlich eine von der Sohnprozessnummer verschiedene Vaterprozessnummer (d.h. die Prozessnummer des Vaterprozesses).
- Der Sohnprozess besitzt eine eigene Kopie des Vater-Dateideskriptors. Alle Dateideskriptoren des Sohnprozesses teilen die Dateibeschreibung des Vaterprozess-Dateideskriptors.
- Der Sohnprozess besitzt eine eigene Kopie der Vater-Dateiverzeichnisströme. Alle Dateiverzeichnisströme des Sohnprozesses können den Lese-/Schreibzeiger mit dem entsprechenden Dateiverzeichnisstrom des Vaterprozesses teilen.
- Der Sohnprozess kann eine eigene Kopie des Vater-Meldungskatalog-Deskriptors besitzen.

– Die Werte des Sohnprozesses für die `tms`-Strukturkomponenten `tms_utime`, `tms_stime`, `tms_cutime` und `tms_cstime` sind gleich 0 gesetzt (siehe `times()`).
– Die Restzeit bis zu einem Alarmuhr-Signal ist gleich 0 gesetzt (siehe `alarm()`).
– Alle `semadj`-Werte sind gelöscht (siehe `semop()`).
– Dateisperren des Vaterprozesses werden vom Sohnprozess nicht geerbt (siehe auch `fcntl()`).
– Die für den Sohnprozess anstehende Signalmenge wird mit der leeren Menge initialisiert.

Werden Threads verwendet, so wirkt sich die Funktion auf den Prozess oder auf einen Thread wie folgt aus:

– Ein Prozess wird mit einem einzigen Thread erzeugt. Wenn ein ”multi-threaded” Prozess `fork()` aufruft, enthält der neue Prozess eine Kopie des aufrufenden Threads und seines gesamten Adressraums einschließlich des Zustands von Mutex-Objekten und anderen Ressourcen. Mit der Funktion `pthread_atfork()`können Fork-Handler eingerichtet werden.
– *BS2000*
BS2000-Dateien, mit Ausnahme von Memory-Pools, werden mit `fork()` nicht vererbt. Außerdem werden die folgenden BS2000-Ressourcen nicht vererbt:
  – offene BS2000-Dateien sind nicht mehr offen
  – AID-Haltepunkte
  – Task File Table (TFT)
  – SYSFILE-Zuweisungen
  – angemeldete STXIT- und Contingency-Routinen ❑

Returnwert
- 0 — bei erfolgreicher Beendigung. Dieser Returnwert wird an den Sohnprozess und die Prozessnummer des Sohnprozesses an den Vaterprozess zurückgeliefert.
- -1 — bei Fehler. Dieser Returnwert wird an den Vaterprozess zurückgeliefert; es wird kein Sohnprozess erzeugt. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `fork()` schlägt fehl, wenn gilt:

- `EAGAIN` — Das System hat nicht genügend Ressourcen, um einen weiteren Prozess zu erzeugen. Die systembedingte Grenze für die Anzahl der systemweit oder für eine einzelne Benutzernummer gleichzeitig ablaufenden Prozesse `{CHILD_MAX}` würde überschritten werden,
oder wenn im Vaterprozess DIV- oder FASTRAM-Bereiche abgelegt sind.

Erweiterung
- `ENOMEM` — Der Swap-Bereich ist zu klein. ❑

Hinweis `fork()` kann ab dieser Version auch in Signalbehandlungs- und Contingency-Routinen verwendet werden.

Siehe auch `alarm()`, `exec`, `fcntl()`, `semop()`, `signal()`, `times()`, `sys/types.h`, `unistd.h`.

fpathconf - Wert einer Pfadnamen-Variablen ermitteln

Syntax #include <unistd.h>

long int fpathconf(int *fildes*, int *name*);

Beschreibung

Siehe `pathconf()`.

fprintf, printf, sprintf - formatiert in Ausgabestrom schreiben

Syntax #include <stdio.h>

int fprintf(FILE **stream*, const char **format* [, *arglist*]);
int printf (const char **format* [, *arglist*]);
int sprintf (char **s*, const char **format* [, *arglist*]);

Beschreibung

`fprintf()` schreibt Ausgaben formatiert in den Ausgabestrom, auf den *stream* zeigt.

`printf()` schreibt Ausgaben formatiert in den Standard-Ausgabestrom `stdout`.

`sprintf()` schreibt Ausgaben formatiert in aufeinander folgende Bytes, beginnend an der Adresse *s*, gefolgt vom einem Nullbyte. Der Benutzer ist dafür verantwortlich, dass genügend Platz für die Ausgabe verfügbar ist.

Jede dieser Funktionen konvertiert die Argumente in *arglist* und gibt sie unter der Steuerung von *format* aus.

format ist eine Zeichenkette, die in ihrem anfänglichen Umschaltmodus beginnt und endet, sofern einer definiert ist, und keine, eine oder mehrere Umwandlungsanweisungen enthält. *format* kann drei Arten von Zeichen enthalten:

– Zeichen vom Typ `char`, die 1 : 1 in den Ausgabedatenstrom kopiert werden.

– Zwischenraumzeichen, beginnend mit einem Gegenschrägstrich (`\`) (siehe `isspace()`).

– Umwandlungsanweisungen, beginnend mit dem Prozentzeichen (`%`), von denen jede keinem, einem oder mehreren Argumenten in *arglist* zugeordnet wird. Wenn in *arglist* weniger Argumente übergeben werden, als in *format* festgelegt sind, ist das Ergebnis undefiniert. Wenn in *format* weniger Argumente festgelegt sind, als in *arglist* übergeben werden, werden die überflüssigen Argumente ignoriert.

Zeichen

Für die derzeitige Version des C-Laufzeitsystems gilt:
Es sind nur Zeichen aus dem EBCDIC-Zeichensatz zugelassen.

Zwischenraumzeichen

Zeichen	Bedeutung	Steuerverhalten
`\b`	Rücksetzzeichen	Die Ausgabe wird 1 Zeichen vor die aktuelle Position verschoben, es sei denn, die aktuelle Position ist der Zeilenanfang. In dieser Version des C-Laufzeitsystems wird `\b` nur für BS2000-Ausgabe, aber nicht für Ausgabe ins POSIX-Subsystem ausgewertet.
`\f`	Seitenvorschub	Die Ausgabe wird an den Anfang der nächsten logischen Seite verschoben. In dieser Version des C-Laufzeitsystems wird `\f` nur für BS2000-Ausgabe, aber nicht für Ausgabe ins POSIX-Subsystem ausgewertet.
`\n`	Zeilenendezeichen	Die Ausgabe wird an den Anfang der nächsten Zeile verschoben.
`\r`	Wagenrücklauf	Die Ausgabe wird an den Anfang der aktuellen Zeile verschoben. Alles, was vorher in den Datenstrom dieser Zeile geschrieben wurde, wird nicht ausgegeben.
`\t`	horizontaler Tabulator	Die Ausgabe wird 8 Zeichen nach der aktuellen Position verschoben.
`\v`	vertikaler Tabulator	Die Ausgabe wird auf die Position des nächsten vertikalen Tabulators verschoben. In dieser Version des C-Laufzeitsystems wird `\v` nur für BS2000-Ausgabe, aber nicht für Ausgabe ins POSIX-Subsystem ausgewertet.

Umwandlungsanweisungen

Die Umwandlung kann auf das *a*-te Argument der Argumentliste an Stelle des nächsten, unbenutzten Arguments angewendet werden. In diesem Fall wird das Umwandlungszeichen `%` durch die Zeichenfolge `%`*a*`$` ersetzt, wobei *a* eine Dezimalzahl aus dem Bereich `[1, {NL_ARGMAX}]` ist, welche die Position des Arguments in der Argumentliste angibt. Diese Eigenschaft erlaubt die Definition von Formatzeichenketten, die Argumente entsprechend einer speziellen Sprache auswählen.

In Formatzeichenketten, die die Umwandlungsanweisung `%`*a*`$` enthalten, können Elemente aus *arglist* durch die Formatzeichenkette *format* so oft wie gewünscht referenziert werden (*a*-mal). In Formatzeichenketten, die die Umwandlungsanweisung `%` enthalten, wird jedes Argument genau einmal ausgewertet.

Alle Formen von `printf()` erlauben das Einfügen eines landessprach-spezifischen Dezimalzeichens in die Ausgabezeichenkette. Das Dezimalzeichen wird durch die Lokalität des Progamms definiert (Kategorie `LC_NUMERIC`). In der Lokalität `POSIX` oder einer Lokalität, bei der das Dezimalzeichen nicht definiert ist, ist es auf . (Punkt) voreingestellt.

Jede Umwandlungsanweisung wird entweder vom Zeichen % oder von der Zeichenfolge %*a*$ eingeleitet; darauf folgen die nachfolgend angegebenen Daten:

– Keines oder mehrere **Formatierungszeichen**, die die Bedeutung der Umwandlungsanweisung verändern.

– Eine optionale Dezimalzahl, die eine minimale **Feldbreite** für die Ausgabe eines Arguments angibt. Wenn der umgewandelte Wert aus weniger Zeichen als der Feldbreite besteht, wird links bis zur Feldbreite aufgefüllt (bzw. rechts, wenn das Formatierungszeichen "–" für linksbündige Ausrichtung angegeben wurde).

– Eine **Genauigkeit**, die angibt, wie viele Ziffern mindestens für die Umwandlungen d, i, o, u, x oder X erscheinen sollen, wie viele Ziffern nach dem Dezimalzeichen für die Umwandlungen e, E und f erscheinen sollen, wie viele signifikante Stellen bei den Umwandlungen g und G vorhanden sind oder wie viele Zeichen maximal aus der Zeichenkette für die Umwandlung *s* ausgegeben werden sollen. Die Genauigkeit hat die Form ".", gefolgt von einer Zeichenkette aus dezimalen Ziffern, wobei keine Ziffer als 0 gewertet wird.

– Ein optionales h, das angibt, dass ein folgendes d, i, o, u, x oder X auf ein Argument vom Typ short int oder unsigned short int angewendet werden soll (das Argument ist entsprechend der ganzzahligen Erweiterung erweitert worden, und sein Wert wird vor der Ausgabe in ein short int oder unsigned short int umgewandelt);
ein optionales h, das angibt, dass ein nachfolgendes *n* auf einen Zeiger auf ein Argument vom Typ short int angewendet werden soll;
ein optionales l , welches angibt, dass ein folgendes d, i, o, u, x oder X auf ein Argument vom Typ long int oder unsigned long int angewendet werden soll;
ein optionales l, welches angibt, dass ein nachfolgendes *n* auf einen Zeiger auf ein Argument vom Typ long int angewendet werden soll;
ein optionales L, welches angibt, dass ein folgendes e, E, f, g oder G auf ein Argument vom Typ long double angewendet werden soll.

  Wenn h , l oder L vor einem anderen Umwandlungszeichen auftreten, ist das Verhalten undefiniert.

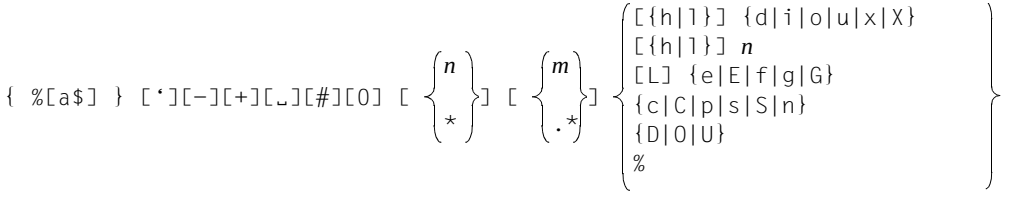
– Ein **Umwandlungszeichen**, das den Typ der durchzuführenden Umwandlung angibt.

Feldbreite, Genauigkeit oder beides können durch das Zeichen * (Asterisk) angegeben werden. In diesem Fall versorgt ein Argument vom Typ int Feldbreite oder Genauigkeit. Argumente, die Feldbreite, Genauigkeit oder beides spezifizieren, müssen in dieser Reihenfolge vor dem Argument stehen, das umgewandelt werden soll. Eine negative Feldbreite wird als "–" Formatierungszeichen interpretiert, dem eine positive Feldbreite folgt. Eine negative Genauigkeit wird interpretiert, als ob die Genauigkeit weggelassen wird. In Formatzeichenketten, die eine Umwandlungsanweisung der Form %*a*$ enthalten, kann eine Feldbreite oder Genauigkeit durch die Zeichenfolge **a*$ angegeben werden, wobei *a* eine Dezimalzahl aus dem Intervall [1, {NL_ARGMAX}] ist, welche die Position einer Ganzzahl in der Argumentliste angibt, die ihrerseits die Feldbreite oder Genauigkeit angibt, z.B.:

```
printf ("%1$d:%2$.*3$d:%4$.*3$d\n", hour, min, precision, sec);
```

format kann entweder nummerierte oder nichtnummerierte Argument-Angaben enthalten, d.h. entweder %*a*$ und *a*$ oder % und *, aber nicht beides. Die Ergebnisse einer Vermischung von nummerierten und unnummerierten Argumentangaben in einer *format*-Zeichenkette sind undefiniert. Die Verwendung nummerierter Argumentangaben für das Argument *A* erfordert, dass alle führenden Argumente, vom ersten bis zum (*A*-1), in der Formatzeichenkette angegeben werden.

Umwandlungsanweisungen können in XPG4 Version 2-konformen Umgebungen wie folgt aussehen:



1. Anfang einer Umwandlungsanweisung
2. Formatierungszeichen
3. Feldbreite
4. Genauigkeit
5. Zeichen, die die eigentliche Umwandlung festlegen

Formatierungszeichen

‘ Der ganzzahlige Teil des Ergebnisses einer Dezimalumwandlung (%i, %d, %u, %f, %g oder %G) wird mit Gruppierungszeichen in Tausenderbereiche unterteilt. Für andere Umwandlungen ist das Verhalten undefiniert. Als Gruppierungszeichen wird das nichtmonetäre verwendet.

− Das Ergebnis der Umwandlung wird linksbündig innerhalb des Felds ausgerichtet.

+ Das Ergebnis einer vorzeichenbehafteten Umwandlung beginnt immer mit einem Vorzeichen (+ oder −).

␣ Wenn das erste Zeichen einer vorzeichenbehafteten Umwandlung kein Vorzeichen ist, wird dem Resultat ein Leerzeichen vorangestellt. Dies bedeutet, dass das Leerzeichen ignoriert wird, wenn sowohl das Leerzeichen als auch das Zeichen + angegeben werden.

Dieses Formatierungszeichen gibt an, dass der umzuwandelnde Wert in einer "anderen Form" darzustellen ist. Für `c`, `d`, `i`, `s` und `u` hat dieses Formatierungszeichen keine Wirkung. Für die Umwandlung `o` wird die Genauigkeit in der Form erhöht, dass die erste Ziffer des Ergebnisses die Ziffer 0 ist. Für `x` (oder `X`) wird einem Resultat ungleich 0 die Zeichenfolge `"0x"` (oder `"0X"`) vorangestellt. Für `e`, `E`, `f`, `g` oder `G` enthält das Ergebnis immer ein Dezimalzeichen, auch wenn keine weiteren Ziffern folgen (normalerweise erscheint ein Dezimalzeichen nur dann im Ergebnis, wenn es von einer Ziffer gefolgt wird). Für `g` und `G` werden abschließende Nullen nicht aus dem Ergebnis entfernt, wie sonst üblich.

0 Für `d`, `i`, `o`, `u`, `x`, `X`, `e`, `E`, `f`, `g` und `G` werden führende Nullen, nach einer vorhandenen Anzeige von Vorzeichen oder Basis, verwendet, um bis zur Feldbreite aufzufüllen; es wird nicht mit Leerzeichen aufgefüllt. Wenn sowohl das Formatierungszeichen `0` als auch − angegeben werden, wird das Formatierungszeichen `0` ignoriert. Für `d`, `i`, `o`, `u`, `x` und `X` wird, wenn eine Genauigkeit angegeben ist, das Formatierungszeichen `0` ignoriert. Für andere Umwandlungen ist das Verhalten undefiniert.

Umwandlungszeichen

`d, i` Das `int`-Argument wird in eine vorzeichenbehaftete Dezimalzahl der Form [−]*dddd* umgewandelt. Die Genauigkeit legt die minimale Anzahl von Ziffern fest, die erscheinen sollen; wenn der umzuwandelnde Wert in weniger Ziffern repräsentiert werden kann, wird er um führende Nullen erweitert. Die voreingestellte Genauigkeit ist 1. Das Ergebnis einer Umwandlung des Werts 0 mit einer ausdrücklich genannten Genauigkeit von 0 liefert kein Zeichen.

`o` Das `unsigned int`-Argument wird in eine vorzeichenlose Oktalzahl der Form *dddd* umgewandelt. Die Genauigkeit legt die minimale Anzahl von Ziffern fest, die erscheinen sollen; wenn der umzuwandelnde Wert in weniger Ziffern repräsentiert werden kann, wird er um führende Nullen erweitert. Die voreingestellte Genauigkeit ist 1. Das Ergebnis einer Umwandlung des Werts 0 mit einer ausdrücklich genannten Genauigkeit von 0 liefert kein Zeichen.

`u` Das `unsigned int`-Argument wird in eine vorzeichenlose Dezimalzahl der Form *dddd* umgewandelt. Die Genauigkeit legt die minimale Anzahl von Ziffern fest, die erscheinen sollen; wenn der umzuwandelnde Wert in weniger Ziffern repräsentiert werden kann, wird er um führende Nullen erweitert. Die

voreingestellte Genauigkeit ist 1. Das Ergebnis einer Umwandlung des Werts 0 mit einer ausdrücklich genannten Genauigkeit von 0 liefert kein Zeichen.

`x` Das `unsigned int`-Argument wird in eine vorzeichenlose Hexadezimalzahl der Form *dddd* umgewandelt; außer den Zahlen werden die Buchstaben `abcdef` als numerische Zeichen verwendet. Die Genauigkeit legt die minimale Anzahl von Ziffern fest, die erscheinen sollen; wenn der umzuwandelnde Wert in weniger Ziffern repräsentiert werden kann, wird er um führende Nullen erweitert. Die voreingestellte Genauigkeit ist 1. Das Ergebnis einer Umwandlung des Werts 0 mit einer ausdrücklich genannten Genauigkeit von 0 liefert kein Zeichen.

`X` Verhält sich wie das Umwandlungszeichen `x`, nur werden die Buchstaben `ABCDEF` verwendet.

`f` Das `double`-Argument wird in die dezimale Schreibweise der Form `[-]`*ddd*`.`*ddd* umgewandelt, wobei die Anzahl der Ziffern nach dem Dezimalzeichen gleich der angegebenen Genauigkeit ist. Ist keine Genauigkeit angegeben, so wird diese mit dem Wert 6 angenommen; wenn die Genauigkeit gleich 0 ist und kein #-Formatierungszeichen gesetzt ist, wird kein Dezimalzeichen ausgegeben. Wenn das Dezimalzeichen erscheint, wird davor mindestens eine Ziffer ausgegeben. Der Wert wird auf die entsprechende Zahl von Ziffern gerundet.

`e, E` Das `double`-Argument wird in die Form `[-]`*d.ddde+-dd* umgewandelt, wobei genau eine Ziffer vor dem Dezimalzeichen ausgegeben wird (die ungleich 0 ist, wenn das Argument ungleich 0 ist) und wobei die Anzahl der Nachkommastellen gleich der Genauigkeit ist; wenn die Genauigkeit fehlt, wird der Wert 6 angenommen; wenn die Genauigkeit gleich 0 und kein #-Formatierungszeichen gesetzt ist, wird kein Dezimalzeichen ausgegeben. Der Wert wird auf die entsprechende Zahl von Ziffern gerundet. Das Umwandlungszeichen `E` erzeugt eine Zahl mit `E` an Stelle von `e` für die Anzeige des Exponenten. Der Exponent enthält immer mindestens zwei Ziffern. Wenn der Wert gleich 0 ist, ist der Exponent gleich 0.

`g, G` Das `double`-Argument wird in die Form von `f` oder `e` umgewandelt (bzw. in die Form `E` für das Umwandlungszeichen `G`), wobei die Genauigkeit die Anzahl der signifikanten Stellen angibt. Wenn die angegebene Genauigkeit gleich 0 ist, wird der Wert 1 angenommen. Die Form hängt vom umgewandelten Wert ab; die Form `e` wird nur dann verwendet, wenn der Exponent einer solchen Umwandlung kleiner als -4 oder größer gleich der Genauigkeit ist. Abschließende Nullen werden vom gebrochenen Teil des Ergebnisses entfernt; ein Dezimalzeichen erscheint nur dann, wenn es von einer Ziffer gefolgt wird.

`c` Das Argument vom Typ `int` wird in den Typ `unsigned char` umgewandelt, das resultierende Zeichen wird geschrieben.

`s` Das Argument ist vom Typ Zeiger auf `char`. Zeichen aus dem Vektor werden bis zum abschließenden Nullbyte geschrieben (ausschließlich); wenn die Genauigkeit angegeben ist, werden nicht mehr als diese Anzahl von Zeichen geschrieben. Wird die Genauigkeit nicht angegeben oder ist diese größer als die Länge des Vektors, enthält der Vektor das Nullbyte.

`p` Das Argument muss ein Zeiger auf `void` sein. Der Wert des Zeigers wird in eine Folge von abdruckbaren Zeichen umgewandelt; im POSIX-Subsystem ist dies die hexadezimale Darstellung der Adresse.

`n` Das Argument muss ein Zeiger auf `int` sein, in welches die Anzahl der bisher von einer der `printf`-Funktionen geschriebenen Bytes eingetragen wird. Es wird kein Argument umgewandelt.

`C` `wchar_t` wird in einen Byte-Vektor umgewandelt, der ein Zeichen darstellt. Dieses Zeichen wird geschrieben. Wenn die Genauigkeit angegeben ist, ist die Wirkung nicht definiert. Die Umwandlung entspricht einer Umwandlung durch `wctomb()`.
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). Daher ist dieses Umwandlungszeichen wirkungslos.

`S` Das Argument muss ein Zeiger auf einen Vektor vom Typ `wchar_t` sein. Langzeichenwerte dieses Vektors werden bis zum abschließenden Nullbyte (ohne das Nullbyte) in eine Bytefolge umgewandelt. Die Ergebnis-Bytes werden geschrieben. Wenn die Genauigkeit angegeben ist, werden nur die angegebenen Bytes geschrieben. Es werden nur vollständige Zeichen geschrieben. Wenn die Genauigkeit nicht angegeben oder größer als die Vektorgröße der umgewandelten Bytes ist, muss der Langzeichen-Vektor mit einem Nullbyte abgeschlossen werden. Die Umwandlung entspricht der Umwandlung durch `wcstombs()`.
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). Daher ist dieses Umwandlungszeichen wirkungslos.

`%` Es wird das Zeichen `%` ausgegeben; kein Argument wird umgewandelt.

Wenn das Zeichen nach `%` oder nach der Zeichenfolge *%a*`$` kein gültiges Umwandlungszeichen ist, ist das Ergebnis der Umwandlung undefiniert.

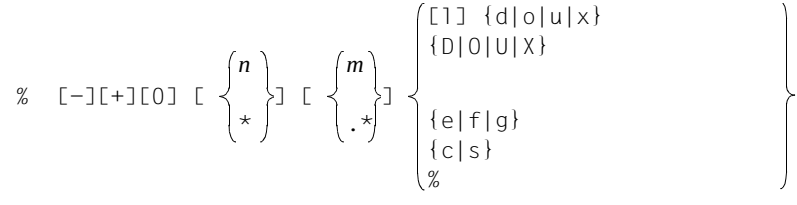
In keinem Fall verursacht eine nicht existierende oder zu kleine Feldbreite das Abschneiden eines Feldes; wenn das Ergebnis einer Umwandlung breiter als die Feldbreite ist, wird das Feld einfach erweitert, um die Ausgabe aufzunehmen. Zeichen, die von `printf()` und `fprintf()` erzeugt werden, werden ausgegeben, als ob `putc()` aufgerufen werden würde.

Die Strukturkomponenten `st_ctime` und `st_mtime` der Datei werden zwischen der erfolgreichen Ausführung von `fprintf()` oder `printf()` und der nächsten erfolgreichen Beendigung eines Aufrufs von `fflush()` oder `fclose()` für denselben Datenstrom oder einem Aufruf von `exit()` oder `abort()` für die Änderung markiert (siehe `sys/stat.h`).

BS2000
Für die Ausgabe in `STDOUT` unterscheiden sich die Umwandlungsanweisungen, je nachdem, ob KR-Funktionalität (nur bei C/C++ Versionen kleiner V3 vorhanden) oder ANSI-Funktionalität unterstützt werden soll.

Umwandlungsanweisung (KR-Funktionalität)
nur bei C/C++ Versionen kleiner V3 vorhanden

Umwandlungsanweisungen können folgendermaßen aufgebaut sein:



1. Jede Umwandlungsanweisung muss mit einem Prozentzeichen (%) beginnen.
2. Formatierungszeichen, z.B. zur Steuerung der Vorzeichenausgabe, links- oder rechtsbündigen Ausrichtung, Breite des Ausgabefeldes etc.
3. Zeichen, die die eigentliche Umwandlung festlegen.

Bedeutung der Formatierungszeichen in der KR-Funktionalität (nur bei C/C++ Versionen kleiner V3 vorhanden):

− Linksbündige Ausrichtung des Ausgabefeldes. Voreingestellt: Rechtsbündige Ausrichtung.

\+ Das Ergebnis einer Umwandlung mit Vorzeichen wird immer mit Vorzeichen ausgegeben. Voreingestellt: Nur ein ggf. negatives Vorzeichen wird ausgegeben.

`0` Mit Nullen auffüllen. Bei allen Umwandlungen wird das Ausgabefeld mit Nullen aufgefüllt. Voreingestellt: Das Ausgabefeld wird mit Leerzeichen aufgefüllt. Die Auffüllung mit Nullen wird auch bei linksbündiger Ausrichtung (Formatierungszeichen `-`) durchgeführt.

n Minimale Gesamtfeldbreite (inklusive Dezimalzeichen). Falls für die Umwandlung einer Zahl mehr Stellen benötigt werden, hat diese Angabe keine Bedeutung. Ist die Ausgabe kürzer als die angegebene Feldbreite, wird sie bis zur Feldbreite mit Leerzeichen bzw. Nullen aufgefüllt (vgl. Formatierungszeichen `-` und 0).

`*` Die Gesamtfeldbreite (siehe *n*) wird statt in der Umwandlungsanweisung mit einem Argument festgelegt. Der aktuelle Wert (ganzzahlig) muss unmittelbar vor dem umzuwandelnden Argument oder unmittelbar vor dem Wert der Genauigkeitsangabe (Formatierungszeichen *.m*) in der Argumentliste stehen (durch ein Komma getrennt).

.m Genauigkeitsangabe.
`e`-, `f`-, `g`-Umwandlung: Genaue Anzahl der Stellen nach dem Dezimalzeichen (maximal 20). Voreingestellt: 6 Stellen.
`s`-Umwandlung: Maximale Anzahl der auszugebenden Zeichen. Voreingestellt: Alle Zeichen bis zum abschließenden Nullbyte.
Bei allen anderen Umwandlungen wird die Genauigkeitsangabe ignoriert.

`.*` Die Genauigkeit (siehe *.m*) wird statt in der Umwandlungsanweisung mit einem Argument festgelegt. Der aktuelle Wert (ganzzahlig) muss durch ein Komma getrennt unmittelbar vor dem umzuwandelnden Argument in der Argumentliste stehen.

Bedeutung der Umwandlungszeichen in der KR-Funktionalität (nur bei C/C++ Versionen kleiner V3 vorhanden):

`l` `l` vor `d`, `o`, `u`, `x`:
Umwandlung eines Arguments vom Typ `long`.
Die Angabe ist identisch mit den Großbuchstaben `D`, `O`, `U`, `X`.

`d, o, u, x` Darstellung einer ganzen Zahl (`int`) als
Dezimalzahl mit Vorzeichen (`d`),
Oktalzahl ohne Vorzeichen (`o`),
Dezimalzahl ohne Vorzeichen (`u`),
Hexadezimalzahl ohne Vorzeichen (`x`).

`f` Darstellung einer Gleitpunktzahl (`float` oder `double`) im Format
`[-]`*ddd.ddd*
Das Dezimalzeichen wird durch die Lokalität (Kategorie `LC_NUMERIC`) be-

einflusst. Voreingestellt ist der Punkt. Die Anzahl der Stellen nach dem Dezimalzeichen hängt von der Genauigkeitsangabe *.m* ab; voreingestellt: 6 Stellen. Bei Genauigkeit 0 erfolgt die Ausgabe ohne Dezimalzeichen.

`e` Darstellung einer Gleitpunktzahl (`float` oder `double`) im Format `[−]`*d*.*ddd*`e{+|−}`*dd*.
Das Dezimalzeichen wird durch die Lokalität (Kategorie `LC_NUMERIC`) beeinflusst. Voreingestellt ist der Punkt. Die Anzahl der Stellen nach dem Dezimalzeichen hängt von der Genauigkeitsangabe *.m* ab; Voreingestellt: 6 Stellen. Bei Genauigkeit 0 wird ein Dezimalzeichen ohne nachfolgende Ziffern ausgegeben.

`g` Darstellung einer Gleitpunktzahl (`float` oder `double`) im `f`- oder `e`-Format. Die Anzahl der Stellen nach dem Dezimalzeichen hängt von der Genauigkeitsangabe *.m* ab. Es wird jeweils die Darstellung gewählt, die unter Wahrung der Genauigkeit am wenigsten Platz beansprucht.

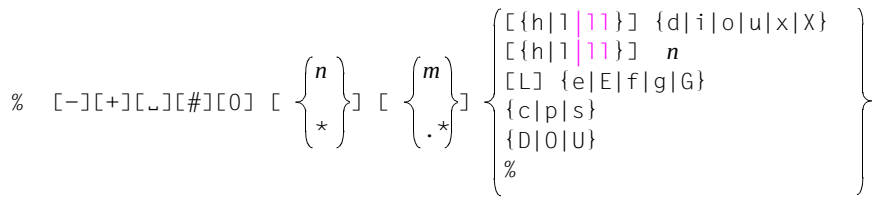
`c` Format für die Ausgabe eines einzelnen Zeichens (`char`). Das Nullbyte wird ignoriert.

`s` Format für die Ausgabe von Zeichenketten. Die `printf`-Funktionen schreiben so viele Zeichen der Zeichenkette, wie mit der Genauigkeit *.m* angegeben ist. Voreingestellt: Die `printf()`-Funktionen schreiben alle Zeichen ausschließlich des abschließenden Nullbytes.

`%` Ausgabe des Zeichens `%`, keine Umwandlung.

Umwandlungsanweisung (ANSI-Funktionalität)

Umwandlungsanweisungen können folgendermaßen aufgebaut sein: :



1. Jede Umwandlungsanweisung muss mit einem Prozentzeichen (%) beginnen.
2. Formatierungszeichen, z.B. zur Steuerung der Vorzeichenausgabe, links- oder rechtsbündigen Ausrichtung, Breite des Ausgabefeldes etc.
3. Zeichen, die die eigentliche Umwandlung festlegen.

Bedeutung der Formatierungszeichen in der ANSI-Funktionalität:

− Linksbündige Ausrichtung des Ausgabefeldes. Voreingestellt: rechtsbündig.

+ Das Ergebnis einer Umwandlung mit Vorzeichen wird immer mit Vorzeichen ausgegeben. Voreingestellt: Nur ein ggf. negatives Vorzeichen wird ausgegeben.

␣ Wenn das erste Zeichen einer mit Vorzeichen umzuwandelnden Zeichenfolge kein Vorzeichen ist, wird dem Ergebnis ein Leerzeichen vorangestellt. Das Formatierungszeichen ␣ wird ignoriert, wenn gleichzeitig + angegeben wird.

Umwandlung des Ergebnisses in ein alternatives Format.
o-Umwandlung: Die Genauigkeit wird so erhöht, dass die erste Ziffer des Ergebnisses die Ziffer 0 ist.
x- bzw. X-Umwandlung: Einem Ergebnis ungleich 0 wird die Zeichenfolge 0x bzw. 0 vorangestellt.
e-, E-, f-, g- bzw. G-Umwandlung: Das Ergebnis enthält immer ein Dezimalzeichen, auch wenn danach keine weiteren Ziffern folgen (normalerweise enthält das Ergebnis nur ein Dezimalzeichen, wenn danach mindestens eine Ziffer folgt). Außerdem werden bei g- bzw. G-Umwandlung abschlie-

ßende Nullen nicht weggelassen.
Das Formatierungszeichen # hat keine Wirkung bei c-, s-, d-, i-, u-Umwandlung.

0 Mit Nullen auffüllen. Bei der Umwandlung von ganzen Zahlen (d, i, o, u, x, X) und Gleitpunktzahlen (e, E, f, g, G) wird das Ausgabefeld mit Nullen aufgefüllt. Voreingestellt: Das Ausgabefeld wird mit Leerzeichen aufgefüllt.
0 wird ignoriert, wenn das Formatierungszeichen oder bei der Umwandlung von ganzen Zahlen eine Genauigkeit *.m* angegeben wird. Bei c-, p- und s-Umwandlung hat das Formatierungszeichen 0 keine Wirkung.

n Minimale Gesamtfeldbreite (inklusive Dezimalzeichen). Falls für die Umwandlung einer Zahl mehr Stellen benötigt werden, hat diese Angabe keine Bedeutung. Ist die Ausgabe kürzer als die angegebene Feldbreite, wird sie bis zur Feldbreite mit Leerzeichen bzw. Nullen aufgefüllt (vgl. Formatierungszeichen − und 0).

* Die Gesamtfeldbreite (siehe *n*) wird statt in der Umwandlungsanweisung mit einem Argument festgelegt. Der aktuelle Wert (ganzzahlig) muss unmittelbar vor dem umzuwandelnden Argument oder unmittelbar vor dem Wert der Genauigkeitsangabe (Formatierungszeichen *.m*) in der Argumentliste stehen (durch ein Komma getrennt).

.m Genauigkeitsangabe.
d-, i-, o-, u-, x- bzw. X-Umwandlung: Minimale Anzahl der auszugebenden Ziffern. Voreingestellt: 1.
e-, E-, f−Umwandlung: Genaue Anzahl der Stellen nach dem Dezimalzeichen (maximal 20). Voreingestellt: 6 Stellen.
g- bzw. G-Umwandlung: Maximale Anzahl der signifikanten Stellen.
s-Umwandlung: Maximale Anzahl der auszugebenden Zeichen. Voreingestellt: Alle Zeichen bis zum abschließenden Nullbyte (\0).

.* Die Genauigkeitsangabe (siehe *.m*) wird statt in der Umwandlungsanweisung mit einem Argument festgelegt. Der aktuelle Wert (ganzzahlig) muss unmittelbar vor dem umzuwandelnden Argument in der Argumentliste stehen (durch ein Komma getrennt).

Bedeutung der Umwandlungszeichen in der ANSI-Funktionalität:

h h vor d, i, o, u, x, X:
Umwandlung eines Arguments vom Typ short.

h vor n:
Das Argument ist vom Typ Zeiger auf short int (keine Umwandlung).

`l`	`l` vor `d`, `i`, `o`, `u`, `x`, `X`: Umwandlung eines Arguments vom Typ `long`. `l` vor `d`, `o`, `u` ist gleichbedeutend mit den Großbuchstaben `D`, `O`, `U`. `l` vor `n`: Das Argument ist vom Typ Zeiger auf `long int` (keine Umwandlung).
`ll`	`ll` vor d, i, o, u, x, X : Umwandlung eines Arguments vom Typ `long long int` bzw. `unsigned long long int`. `ll` vor n: Das Argument ist vom Typ Zeiger auf `long long int`.
`L`	`L` vor `e`, `E`, `f`, `g`, `G`: Umwandlung eines Arguments vom Typ `long double`.
`d`, `i`, `o`, `u`, `x`, `X`	Darstellung einer ganzen Zahl (`int`) als Dezimalzahl mit Vorzeichen (`d`, `i`), Oktalzahl ohne Vorzeichen (`o`), Dezimalzahl ohne Vorzeichen (`u`), Hexadezimalzahl ohne Vorzeichen (`x`, `X`). Bei `x` werden die Kleinbuchstaben `abcdef` benutzt, bei `X` die Großbuchstaben `ABCDEF`. Die Genauigkeitsangabe *.m* gibt die minimale Anzahl der auszugebenden Ziffern an. Ist der Wert mit weniger Ziffern darstellbar, wird das Ergebnis mit führenden Nullen aufgefüllt. Voreingestellt ist Genauigkeit 1. Bei Genauigkeit 0 und Wert 0 erfolgt keine Ausgabe.
`f`	Darstellung einer Gleitpunktzahl (`float` oder `double`) im Format `[-]`*ddd.ddd*. Das Dezimalzeichen wird durch die Lokalität (Kategorie `LC_NUMERIC`) beeinflusst. Voreingestellt ist der Punkt. Die Anzahl der Stellen nach dem Dezimalzeichen hängt von der Genauigkeitsangabe *.m* ab; voreingestellt: 6 Stellen. Bei Genauigkeit 0 erfolgt die Ausgabe ohne Dezimalzeichen.
`e, E`	Darstellung einer Gleitpunktzahl (`float` oder `double`) im Format `[-]`*d.ddd*`e{+|-}`*dd*. Das Dezimalzeichen wird durch die Lokalität (Kategorie `LC_NUMERIC`) beeinflusst. Voreingestellt ist der Punkt. Bei `E`-Umwandlung wird dem Exponenten der Großbuchstabe `E` vorangestellt. Die Anzahl der Stellen nach dem Dezimalzeichen hängt von der Genauigkeitsangabe *.m* ab; voreingestellt: 6 Stellen. Bei Genauigkeit 0 erfolgt die Ausgabe ohne Dezimalzeichen.
`g, G`	Darstellung einer Gleitpunktzahl (`float` oder `double`) im `f`- oder `e`-Format (bzw. bei `G`-Umwandlung im `E`-Format). Die Anzahl der signifikanten Stellen hängt von der Genauigkeitsangabe *.m* ab. Das `e`- bzw. `E`-Format wird nur dann verwendet, wenn der Exponent des Umwandlungsergebnisses kleiner -4 oder größer als die angegebene Genauigkeit ist.

`c` Format für die Ausgabe eines einzelnen Zeichens (`char`). Das Nullbyte wird ignoriert.

`p` Umwandlung eines Arguments vom Typ Zeiger auf `void`. Die Ausgabe erfolgt als 8stellige Hexadezimalzahl (analog zu der Angabe `%08.8x`).

`s` Format für die Ausgabe von Zeichenketten. Die `printf`-Funktionen schreiben so viele Zeichen der Zeichenkette, wie mit der Genauigkeit *.m* angegeben ist. Voreingestellt: Die `printf`-Funktionen schreiben alle Zeichen ausschließlich des abschließenden Nullbytes.

`n` Es findet keine Umwandlung und Ausgabe des Arguments statt. Das Argument ist vom Typ Zeiger auf `int`. Dieser ganzzahligen Variablen wird die Anzahl der Zeichen zugewiesen, die die `printf`-Funktionen bis zu diesem Zeitpunkt für die Ausgabe erzeugt haben.

`%` Ausgabe des Zeichens `%`, keine Umwandlung. ❑

Returnwert Anzahl der übertragenen Zeichen (ohne Nullbyte bei `sprintf()`)
bei erfolgreicher Beendigung.

negativer Wert
bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `fprintf()` und `printf()` schlagen fehl, wenn gilt:

`EAGAIN` Für den Dateideskriptor, der *stream* zu Grunde liegt, ist das `O_NONBLOCK`-Flag gesetzt, und der Prozess wird beim Schreiben verzögert.

`EBADF` Der Dateideskriptor, der sich auf *stream* bezieht, ist kein für das Schreiben gültiger Dateideskriptor.

`EFBIG` Es wurde versucht, in eine Datei zu schreiben, wobei die maximale Dateigröße oder die Grenze für die Prozessdateigröße überschritten wurde (siehe `ulimit()`).

`EINTR` Das Schreiben wurde durch den Empfang eines Signals unterbrochen, und es wurden keine Daten übertragen.

`EIO` Der Prozess ist Mitglied einer Hintergrund-Prozessgruppe und versucht, auf sein steuerndes Terminal zu schreiben. `TOSTOP` ist gesetzt. Das Signal `SIGTTOU` wird vom Prozess weder blockiert noch ignoriert, und die Prozessgruppe des Prozesses ist verwaist.

`ENOSPC` Es ist kein freier Platz auf dem Datenträger mehr vorhanden.

`EPIPE` Es wurde versucht, in eine Pipe oder FIFO zu schreiben, die für keinen Prozess zum Lesen geöffnet war. Außerdem wird das Signal `SIGPIPE` an den Prozess gesendet.

Hinweis

Bei der Umwandlung von Gleitpunktzahlen runden die `printf`-Funktionen auf die angegebene Genauigkeit.

Die `printf`-Funktionen nehmen keine Konvertierung von einem Datentyp in einen anderen vor. Soll ein Wert nicht entsprechend seinem Typ ausgegeben werden, muss er explizit konvertiert werden (z.B. mit dem `cast`-Operator).

Die Zeichen werden nicht sofort in die externe Datei geschrieben, sondern in einem C-internen Puffer zwischengespeichert (siehe Abschnitt „Pufferung von Datenströmen“ auf Seite 77).

Ob `fprintf()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

BS2000
Maximale Anzahl der auszugebenden Zeichen: Bei KR-Funktionalität (nur bei C/C++ Versionen kleiner V3 vorhanden) können pro `fprintf`-Aufruf maximal 1400 Zeichen ausgegeben werden, bei ANSI-Funktionalität maximal 1400 Zeichen pro Konversionselement (z.B. `%s`). ❑

Versuche, nicht initialisierte Variablen oder Variablen nicht entsprechend ihrem Datentyp auszugeben, können zu undefinierten Ergebnissen führen.

Wenn in einer Umwandlungsanweisung dem Prozentzeichen (`%`) ein nicht definiertes Formatierungs- bzw. Umwandlungszeichen folgt, ist das Verhalten undefiniert.

Siehe auch `fputc()`, `fscanf()`, `setlocale()`, `stdio.h`, Abschnitt „Lokalität“ auf Seite 52.

fputc - Byte in Datenstrom schreiben

Syntax #include <stdio.h>

int fputc(int *c*, FILE **stream*);

Beschreibung

`fputc()` wandelt das durch *c* angegebene Byte in `unsigned char` um und schreibt es in den Ausgabestrom, auf den *stream* zeigt. Die Ausgabe erfolgt an der Position, die durch den zugehörigen Lese-/Schreibzeiger für die Datei angegeben wird, sofern er definiert ist. Der Lese-/Schreibzeiger wird anschließend entsprechend erhöht. Wenn die Datei Positionierungsanforderungen nicht unterstützen kann oder der Datenstrom zum Anfügen geöffnet wurde, wird das Zeichen an den Ausgabestrom angefügt.

Die Strukturkomponenten `st_ctime` und `st_mtime` der Datei werden zwischen der erfolgreichen Ausführung von `fputc()` und der nächsten erfolgreichen Beendigung eines Aufrufs von `fflush()` oder `fclose()` für denselben Datenstrom oder einem Aufruf von `exit()` oder `abort()` für die Änderung markiert (siehe `sys/stat.h`).

Returnwert geschriebener Wert
bei Erfolg.

`EOF` bei Fehler, z.B. wenn *stream* nicht zum Schreiben geöffnet ist oder die Ausgabedatei nicht mehr vergrößert werden kann. Das Fehlerkennzeichen wird gesetzt. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `fputc()` schlägt fehl, wenn gilt:

`EAGAIN` Für den Dateideskriptor, der *stream* zu Grunde liegt, ist das `O_NONBLOCK`-Flag gesetzt, und der Prozess wird beim Schreiben verzögert.

`EBADF` Der Dateideskriptor, der sich auf *stream* bezieht, ist kein für das Schreiben gültiger Dateideskriptor.

`EFBIG` Es wurde versucht, in eine Datei zu schreiben, wobei die maximale Dateigröße oder die Grenze für die Prozessdateigröße überschritten wurde (siehe `ulimit()`).

`EINTR` Das Schreiben wurde durch den Empfang eines Signals unterbrochen, und es wurden keine Daten übertragen.

`EIO` Der Prozess ist Mitglied einer Hintergrund-Prozessgruppe und versucht, auf sein steuerndes Terminal zu schreiben. `TOSTOP` ist gesetzt. Das Signal `SIGTTOU` wird vom Prozess weder blockiert noch ignoriert, und die Prozessgruppe des Prozesses ist verwaist.

`ENOSPC` Es ist kein freier Platz auf dem Datenträger mehr vorhanden.

`EPIPE` Es wurde versucht, in eine Pipe oder FIFO zu schreiben, die für keinen Prozess zum Lesen geöffnet war. Außerdem wird das Signal `SIGPIPE` an den Prozess gesendet.

Werden Threads verwendet, so wirkt sich die Funktion auf den Prozess oder auf einen Thread wie folgt aus: Beim `EPIPE`-Fehler wird das Signal `SIGPIPE` nicht an den Prozess, sondern an den aufrufenden Thread gesendet.

Hinweis Die Zeichen werden nicht sofort in die externe Datei geschrieben, sondern in einem C-internen Puffer zwischengespeichert (siehe Abschnitt „Pufferung von Datenströmen“ auf Seite 77).

Bei der Ausgabe in Textdateien werden die Steuerzeichen für Zwischenraum (`\n`, `\t` etc.) je nach Art der Textdatei in ihre entsprechende Wirkung umgesetzt (siehe Abschnitt „Zwischenraumzeichen“ auf Seite 85).

`fputc()` läuft langsamer als `putc()`, beansprucht jedoch weniger Speicherplatz pro Aufruf.

Ob `fputc()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

Siehe auch `ferror()`, `fopen()`, `putc()`, `puts()`, `setbuf()`, `stdio.h`, `sys/stat.h`.

fputs - Zeichenkette in Datenstrom schreiben

Syntax

```
#include <stdio.h>

int fputs(const char *s, FILE *stream);
```

Beschreibung

`fputs()` schreibt die mit dem Nullbyte abgeschlossene Zeichenkette, auf die *s* zeigt, auf den Datenstrom, auf den *stream* zeigt. Das abschließende Nullbyte wird nicht geschrieben.

Die Strukturkomponenten `st_ctime` und `st_mtime` der Datei werden zwischen der erfolgreichen Ausführung von `fputs()` und der nächsten erfolgreichen Beendigung eines Aufrufs von `fflush()` oder `fclose()` für denselben Datenstrom oder einem Aufruf von `exit()` oder `abort()` für die Änderung markiert (siehe `sys/stat.h`).

Returnwert

nicht negative Zahl — bei Erfolg.

BS2000

0 — bei Erfolg. ❑

`EOF` — bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler

Siehe `fputc()`.

Hinweis

`puts()` fügt im Gegensatz zu `fputs()` ein Zeilenendezeichen an.

Bei der Ausgabe in Textdateien werden die Steuerzeichen für Zwischenraum (`\n`, `\t` etc.) je nach Art der Textdatei in ihre entsprechende Wirkung umgesetzt (siehe Abschnitt „Zwischenraumzeichen“ auf Seite 85).

Ob `fputs()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

Siehe auch `fopen()`, `fputc()`, `putc()`, `puts()`, `stdio.h`, `sys/stat.h`.

fputwc - Langzeichen in Datenstrom schreiben

Syntax

```
#include <wchar.h>

Optional
#include <stdio.h> ❑

wint_t fputwc(wint_t wc, FILE *stream);
```

Beschreibung

`fputwc()` schreibt das Zeichen, das dem Langzeichenwert *wc* entspricht, in den Ausgabestrom, auf den *stream* zeigt. Das Zeichen wird an die Position geschrieben, die durch den zugehörigen Lese-/Schreibzeiger für den Datenstrom angegeben ist (falls dieser definiert wurde). Der Zeiger wird entsprechend weiterbewegt. Wenn die Datei Positionierungsanforderungen nicht unterstützt oder der Datenstrom im Anfügemodus geöffnet wurde, wird das Zeichen an den Ausgabestrom angehängt. Wenn während der Schreiboperation ein Fehler auftritt, ist der Einfügemodus der Ausgabedatei in einem undefinierten Zustand.

Die Strukturkomponenten `st_ctime` und `st_mtime` der Datei werden zwischen der erfolgreichen Ausführung von `fputwc()` und der nächsten erfolgreichen Beendigung eines Aufrufs von `fflush()` oder `fclose()` für denselben Datenstrom oder einem Aufruf von `exit()` oder `abort()` für die Änderung markiert (siehe `sys/stat.h`).

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Returnwert

wc	bei erfolgreicher Beendigung.
`WEOF`	bei Fehler. Das Fehlerkennzeichen für den Datenstrom wird gesetzt. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler

`fputwc()` schlägt fehl, wenn der Datenstrom nicht gepuffert ist oder Daten im Puffer von *stream* geschrieben werden und wenn gilt:

`EAGAIN`	Das Flag `O_NONBLOCK` wird für den *stream* zu Grunde liegenden Dateideskriptor gesetzt. Es tritt für den Prozess eine Verzögerung bei der Schreiboperation ein.
`EBADF`	Der dem Datenstrom zu Grunde liegende Dateideskriptor ist für eine Schreiboperation ungültig.
`EFBIG`	Es wurde versucht, in eine Datei zu schreiben, die die maximale Dateigröße oder die Dateigrößenbegrenzung des Prozesses überschreitet (siehe `ulimit()`).
`EINTR`	Die Schreiboperation wurde auf Grund des Empfangs eines Signals beendet. Es wurden keine Daten übertragen.

Erweiterung

EINVAL	Es wurde versucht, auf eine BS2000-Datei zuzugreifen. ❑
EIO	Der Prozess ist Teil einer Hintergrund-Prozessgruppe, die versucht, auf das steuernde Terminal zu schreiben. TOSTOP wird gesetzt. Der Prozess ignoriert oder blockiert dass Signal SIGTOU nicht, und die Prozessgruppe des Prozesses ist verwaist.
ENOSPC	Nicht genügend Speicherplatz auf dem Gerät, das die Datei enthält.
EPIPE	Es wird versucht, in eine Pipe oder FIFO zu schreiben, die für keinen Prozess zum Lesen geöffnet ist. Ein Signal SIGPIPE wird auch an den Prozess gesendet. Werden Threads verwendet, so wirkt sich die Funktion auf den Prozess oder auf einen Thread wie folgt aus: Beim EPIPE-Fehler wird das Signal SIGPIPE nicht an den Prozess, sondern an den aufrufenden Thread gesendet.

Siehe auch ferror(), fopen(), setbuf(), stdio.h, sys/stat.h, wchar.h.

fputws - Langzeichenkette in Datenstrom schreiben,

Syntax

```
#include <wchar.h>

Optional
#include <stdio.h> ❑

int fputws(const wchar_t *ws, FILE *stream);
```

Beschreibung

`fputws()` schreibt eine Zeichenkette mit einem Nullbyte entsprechend der Zeichenkette aus Langzeichenwerten, auf die *ws* zeigt, auf den Datenstrom, auf den *stream* zeigt. Es wird kein Zeichen geschrieben, das dem abschließenden Nullbyte entspricht.

Die Strukturkomponenten `st_ctime` und `st_mtime` der Datei werden zwischen der erfolgreichen Ausführung von `fputws()` und der nächsten erfolgreichen Beendigung eines Aufrufs von `fflush()` oder `fclose()` für denselben Datenstrom oder einem Aufruf von `exit()` oder `abort()` für die Änderung markiert (siehe `sys/stat.h`).

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Returnwert

nicht negative Zahl	bei erfolgreicher Beendigung.
-1	bei Fehler, z.B. wenn der Datenstrom nicht gepuffert ist oder Daten im Puffer von *stream* geschrieben werden müssen. Das Fehlerkennzeichen für den Datenstrom wird gesetzt. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler Siehe `fputwc()`.

Hinweis `fputws()` hängt kein Zeilenendezeichen an.

Siehe auch `fopen()`, `fputwc()`, `stdio.h`, `sys/stat.h`, `wchar.h`.

fread - Daten binär einlesen

Syntax #include <stdio.h>

size_t fread(void *_ptr_, size_t _size_, size_t _nitems_, FILE *_stream_);

Beschreibung

`fread()` liest *nitems* Elemente der Größe *size* aus dem Datenstrom, auf den *stream* zeigt, in den Vektor, auf den *ptr* zeigt. Der Lese-/Schreibzeiger des Datenstroms wird, wenn er definiert ist, um die Anzahl von Bytes weitergeschaltet, die erfolgreich gelesen wurden. Wenn ein Fehler auftritt, ist der sich daraus ergebende Wert des Lese-/Schreibzeigers unbestimmt. Wenn ein Element teilweise gelesen wird, ist sein Wert unbestimmt.

`fread()` kann die Strukturkomponente `st_atime` für die Datei, der *stream* zugeordnet ist, zum Ändern markieren (siehe `sys/stat.h`). Die Strukturkomponente `st_atime` wird aktualisiert, sobald `fgetc()`, `fgets()`, `fgetwc()`, `fgetws()`, `fread()`, `fscanf()`, `getc()`, `getchar()`, `gets()` oder `scanf()` erfolgreich für *stream* aufgerufen werden und Daten zurückliefern, die nicht durch einen vorangegangenen Aufruf von `ungetc()` oder `ungetwc()` bereitgestellt wurden.

BS2000
Satz-Ein-/Ausgabe

`fread()` liest einen Satz (bzw. Block) von der aktuellen Dateiposition.

Anzahl der einzulesenden Zeichen: Im folgenden sei *n* die Gesamtanzahl der einzulesenden Zeichen, d.h.

$n = size * nitems$

Ist *n* größer als die aktuelle Satzlänge, wird trotzdem nur dieser Satz gelesen.

Ist *n* kleiner als die aktuelle Satzlänge, werden nur die ersten *n* Zeichen des Satzes gelesen. Beim nächsten Lesezugriff werden die Daten des nächsten Satzes gelesen.

`fread()` liefert den gleichen Returnwert wie bei Datenstrom-Ein-/Ausgabe, nämlich die Anzahl der vollständig eingelesenen Elemente. Bei Satz-Ein-/Ausgabe ist es sinnvoll, ausschließlich die Elementgröße 1 zu verwenden, da in diesem Fall der Returnwert der Länge des gelesenen Satzes entspricht (ohne ein ggf. vorhandenes Satzlängenfeld). ❑

Returnwert

Anzahl der erfolgreich gelesenen Elemente	bei erfolgreicher Beendigung. Der Returnwert ist nur dann kleiner als *nitems*, wenn das Dateiende erreicht wird oder ein Lesefehler auftritt.
0	wenn *size* oder *nitems* gleich 0 sind. Der Inhalt des Vektors, auf den *ptr* zeigt, und der Zustand des Datenstroms bleiben unverändert. `errno` wird nicht gesetzt.

-1 wenn ein Lesefehler auftritt. Das Fehlerkennzeichen für den Datenstrom wird gesetzt. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler Siehe `fgetc()`.

Hinweis `ferror()` oder `feof()` müssen verwendet werden, um zwischen dem Auftreten des Dateiendes und eines Lesefehlers zu unterscheiden.

Der Vektor, auf den *ptr* zeigt, muss zum Abspeichern der eingelesenen Datenelemente ausreichen.

Um sicherzugehen, dass *size* die richtige Anzahl Bytes für ein Datenelement angibt, sollte die Funktion `sizeof()` für die Größe der Dateneinheit verwenden, auf die *ptr* zeigt.

`fread()` liest über Zeilenende (`\n`) hinweg und eignet sich daher dazu, Binärdateien einzulesen.

Ob `fread()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

Siehe auch `feof()`, `ferror()`, `fgetc()`, `fopen()`, `getc()`, `gets()`, `scanf()`, `stdio.h`, `sys/stat.h`.

free - reservierten Speicherbereich freigeben

Syntax #include <stdlib.h>

void free(void **ptr*);

Beschreibung

`free()` gibt den Speicherplatz frei, der zuvor durch `malloc()`, `calloc()` oder `realloc()` reserviert wurde.

`free()` ist Teil eines C-spezifischen Speicherverwaltungspaketes mit einer eigenen Freispeicherverwaltung. Der mit `free()` freigegebene Speicher wird nicht an das Betriebssystem zurückgegeben, sondern durch die Freispeicherverwaltung erfasst.

ptr ist der Zeiger auf den freizugebenden Speicherbereich. *ptr* muss das Ergebnis eines vorangegangenen Aufrufs von `malloc()`, `calloc()` oder `realloc()` sein, andernfalls ist das Ergebnis undefiniert.

Siehe auch `calloc()`, `malloc()`, `realloc()`, `sdtlib.h`.

freopen - Datenstrom leeren und neu öffnen

Name **freopen, freopen64**

Syntax #include <stdio.h>

FILE *freopen(const char **filename*, const char **mode*, FILE **stream*);
FILE *freopen64(const char **filename*, const char **mode*, FILE **stream*);

Beschreibung

`freopen()` versucht zuerst, für den *stream* zugeordneten Datenstrom den Puffer zu leeren und den zugehörigen Dateideskriptor zu schließen. Fehler beim Leeren des Puffers oder beim Schließen der Datei werden ignoriert. Die Kennzeichen für Fehler oder Dateiende des Datenstroms werden gelöscht.

`freopen()` öffnet dann die Datei, auf deren Pfadname *filename* zeigt, und weist ihr den mit *stream* angegebenen Datenstrom zu. Das Argument *mode* wird genauso verwendet wie bei der Funktion `fopen()` (siehe `fopen()`).

Der ursprüngliche Datenstrom wird geschlossen, unabhängig davon, ob das nachfolgende Öffnen erfolgreich ist.

Für die automatische Konvertierung darf das `b` für binär in *modus* nicht angegeben werden. Die Umgebungsvariable `IO_CONVERSION` muss den Wert `YES` haben.

Es besteht kein funktionaler Unterschied zwischen `freopen` und `freopen64`, außer dass `freopen64` einen Dateizeiger zurückgibt, der über die 2GB-Grenze hinausgehen kann. `freopen64()` setzt das `O_LARGEFILE` Bit im File Status Flag.

BS2000
Siehe `fopen()`, `fopen64()`.

Einschränkung
Bezieht sich *stream* auf eine BS2000-Datei und *filename* auf eine POSIX-Datei, ist das Öffnen der POSIX-Datei mit `freopen()` nur möglich, wenn *stream* sich auf `stdin`, `stdout` oder `stderr` bezieht. Ist das nicht der Fall, wird nur die BS2000-Datei geschlossen und 0 zurückgegeben.

Bezieht sich *stream* auf eine POSIX-Datei und *filename* auf eine BS2000-Datei, ist das Öffnen der BS2000-Datei mit `freopen()` nur möglich, wenn *stream* sich auf `stdin`, `stdout` oder `stderr` bezieht. Ist das nicht der Fall, wird nur die POSIX-Datei geschlossen und 0 zurückgegeben. Dies gilt unabhängig davon, womit die Standardströme verknüpft sind.

Returnwert Wert von *stream*
bei Erfolg.

Nullzeiger bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `freopen()` schlägt fehl, wenn gilt:

`EACCES` Für eine Komponente des Pfades existiert kein Durchsuchrecht.
Die Datei existiert, und die für *mode* geltenden Zugriffsrechte werden verweigert.
Die Datei existiert nicht, und das übergeordnete Dateiverzeichnis der zu erstellenden Datei hat kein Schreibrecht.

`EINTR` Während des Systemaufrufs `freopen()` wurde ein Signal abgefangen.

`EISDIR` Die angegebene Datei ist ein Dateiverzeichnis und *mode* verlangt Schreibrecht.

`ELOOP` Es wurden zu viele symbolische Links bei der Auflösung von `path` festgestellt.

`EMFILE` `{OPEN_MAX}`-Dateideskriptoren sind bereits für den aufrufenden Prozess geöffnet.

`ENAMETOOLONG`
Die Länge von *filename* überschreitet `{PATH_MAX}` ,oder eine Komponente des Pfades ist länger als `{NAME_MAX}`.

`ENFILE` Die maximal erlaubte Anzahl von Dateien im System ist bereits geöffnet.

`ENOENT` Die angegebene Datei existiert nicht, oder *filename* zeigt auf eine leere Zeichenkette.

`ENOSPC` Die Datei existiert nicht, und das Dateiverzeichnis, in dem eine neue Datei erstellt werden soll, kann nicht erweitert werden.

`ENOTDIR` Eine Komponente des Pfades ist kein Dateiverzeichnis.

`ENXIO` Die angegebene Datei ist eine zeichen- oder blockorientierte Gerätedatei und das dieser Datei zugeordnete Gerät existiert nicht.

`EOVERFLOW` Die genannte Datei ist eine reguläre Datei, aber ihre Größe kann in einem Objekt des Typs `off_t` nicht korrekt dargestellt werden.

`EROFS` Die angegebene Datei befindet sich in einem Dateisystem, das nur Leserecht hat, und *mode* verlangt Schreibrecht.

`ETXTBSY` Die Datei ist eine reine Prozedurdatei (gemeinsam verwendete Textdatei), die gerade ausgeführt wird und `mode` setzt Schreibzugriff voraus.

Hinweis `freopen()` wird normalerweise dazu verwendet, die Dateizeiger `stdin`, `stdout` und `stderr` anderen Dateien zuzuordnen als den voreingestellten, geöffneten Standarddateien. `stderr` ist standardmäßig nicht gepuffert. Durch die Verwendung von `freopen()` wird `stderr` jedoch gepuffert oder zeilengepuffert.

Ob `freopen()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

BS2000
Siehe `fopen()`.

Siehe auch `creat()`, `fclose()`, `fopen()`, `fdopen()`, `stdio.h`.

frexp - Gleitpunktzahl (double) in Mantisse und Exponent zerlegen

Syntax #include <math.h>

double frexp(double *num*, int **exp*);

Beschreibung

`frexp()` zerlegt einen Gleitpunktwert *num* in die Mantisse *x* und den Exponenten *exp*, nach der Formel:

$num = x * 2^{exp}$

$|x|$ liegt im Intervall [0.5, 1.0]

exp ist ein Zeiger auf eine ganze Zahl, die den Exponenten zur Basis 2 angibt.

`frexp()` ist die Umkehrfunktion von `ldexp()`.

Returnwert

Mantisse *x*	eine Gleitpunktzahl vom Typ `double` im Intervall [0.5, 1.0], die die Gleichung erfüllt: $num = x * 2^{exp}$. Der Exponent wird in *exp* gespeichert.
0	falls *num* gleich 0 ist (in diesem Fall ist auch der Exponent gleich 0).

Hinweis Eine Anwendung, die die Fehlersituation überprüfen möchte, sollte `errno` gleich 0 setzen, bevor `frexp()` aufgerufen wird. Wenn `errno` nach der Rückkehr gesetzt ist, ist ein Fehler aufgetreten.

Siehe auch `ldexp()`, `modf()`, `math.h`.

fscanf, scanf, sscanf - formatiert lesen

Syntax #include <stdio.h>

int fscanf(FILE **stream*, const char **format* [, *arglist*]);
int scanf(const char **format* [, *arglist*]);
int sscanf(const char **s*, const char **format* [, *arglist*]);

Beschreibung

`scanf()` liest Bytes formatiert aus dem Standard-Eingabestrom `stdin`.

`fscanf()` liest Bytes formatiert aus dem Datenstrom, auf den *stream* zeigt.

`sscanf()` liest Bytes formatiert aus der Zeichenkette *s*.

Jede dieser Funktionen liest Bytes, wandelt sie gemäß den Angaben in der Formatzeichenkette *format* um und speichert die Ergebnisse in den Bereichen ab, die mit den eventuell vorhandenen Argumenten von *arglist* angegeben wurden.

format ist eine Zeichenkette, die in ihrem anfänglichen Umschaltmodus beginnt und endet, sofern einer definiert ist, und keine, eine oder mehrere Umwandlungsanweisungen enthält. *format* kann drei Arten von Zeichen enthalten:

– Zeichen vom Typ `char`, die 1 : 1 in den Ausgabedatenstrom kopiert werden.

– Zwischenraumzeichen, beginnend mit einem Gegenschrägstrich (`\`) (siehe `isspace()`).

– Umwandlungsanweisungen, beginnend mit dem Prozentzeichen (`%`), von denen jede keinem, einem oder mehreren Argumenten in *arglist* zugeordnet wird. Wenn in *arglist* weniger Argumente übergeben werden, als in *format* festgelegt sind, ist das Ergebnis undefiniert. Wenn in *format* weniger Argumente festgelegt sind, als in *arglist* übergeben werden, werden die überflüssigen Argumente ignoriert.

Zeichen

Für die derzeitige Version des C-Laufzeitsystems gilt:
Es sind nur Zeichen aus dem EBCDIC-Zeichensatz zugelassen.

Die `scanf`-Funktionen lesen das Eingabezeichen, jedoch ohne es umzuwandeln und in einer Variablen abzuspeichern. Stimmt das Eingabezeichen nicht mit dem in *format* angegebenen Zeichen überein, wird die Eingabebearbeitung abgebrochen.

Zwischenraumzeichen

Die Formatzeichenkette *format* kann beliebig viele oder keine Zwischenraumzeichen enthalten. Diese Zeichen haben keine Steuerfunktion.

Zwischenraumzeichen in der Eingabe werden als Trennzeichen zwischen Eingabefeldern behandelt und nicht mit umgewandelt (Ausnahme siehe %c und %[]). Führende Zwischenraumzeichen werden bei der Eingabe ignoriert.

Je nachdem, welche Funktionalität von scanf-Funktionen unterstützt werden soll, wird eine unterschiedliche Anzahl von Zwischenraumzeichen erkannt.

Zeichen	**Bedeutung**	**Gültig für folgende Funktionalität**		
		XPG4	**ANSI** *(BS2000)*	**KR** *(BS2000)*
␣	Leerzeichen	x	x	x
\n	Zeilenendezeichen	x	x	x
\t	horizontaler Tabulator	x	x	x
\f	Seitenwechsel	x	x	-
\v	vertikaler Tabulator	-	x	-
\r	Wagenrücklauf	-	x	-

Umwandlungsanweisungen

Die Umwandlung kann auf das *a*-te Argument der Argumentliste *arglist* an Stelle des nächsten, unbenutzten Arguments angewendet werden. In diesem Fall wird das Umwandlungszeichen % durch die Zeichenfolge %*a*$ ersetzt, wobei *a* eine Dezimalzahl aus dem Wertebereich [1, {NL_ARGMAX}] ist, die angibt, welche Position das Argument in der Argumentliste besitzt. Diese Eigenschaft erlaubt die Definition von Formatzeichenketten, die Argumente entsprechend einer speziellen Landessprache auswählen. In Formatzeichenketten, die die Umwandlungsanweisung %*a*$ enthalten, ist nicht festgelegt, ob aufgezählte Elemente der Argumentliste *arglist* von der Formatzeichenkette *format* mehr als einmal referenziert werden können.

format kann sowohl % als auch %*a*$ als Umwandlungsanweisung enthalten. Innerhalb einer Formatzeichenkette dürfen aber nicht beide Formen gleichzeitig verwendet werden. Nur %% oder %* können mit der Form %*a*$ gemischt werden.

Alle Formen von fscanf() erlauben das Erkennen eines landessprach-spezifischen Dezimalzeichens in der Eingabezeichenkette. Das Dezimalzeichen wird durch die Lokalität des Progamms definiert (Kategorie LC_NUMERIC). In der Lokalität POSIX oder einer Lokalität, bei der das Dezimalzeichen nicht definiert ist, ist das Dezimalzeichen auf . (Punkt) voreingestellt.

Jede Umwandlungsanweisung wird entweder vom Zeichen % oder von der Zeichenfolge %*a*$ eingeleitet; darauf folgen die nachfolgend angegebenen Daten:

– Ein optionales Zeichen * zur Unterdrückung der Zuweisung.

– Eine optionale Ganzzahl ungleich 0, welche die maximale **Feldbreite** angibt.

– Ein optionales `h`,`l`,`ll` oder `L`, die die Größe des aufnehmenden Objekts angeben.
`d`, `i` und `n` wird ein `h` vorangestellt, wenn das entsprechende Argument ein Zeiger auf `short int` und kein Zeiger auf `int` ist, oder ein `l`, wenn es ein Zeiger auf `long int` ist.
`o`, `u` und `x` wird ein `h` vorangestellt, wenn das entsprechende Argument ein Zeiger auf `unsigned short int` statt auf `unsigned int` ist, oder ein `l`, wenn es ein Zeiger auf `unsigned long int` ist oder ein ll, wenn es ein Zeiger auf long long int ist.
`e`, `f` und `g` wird ein `l` vorangestellt, wenn das entsprechende Argument ein Zeiger auf `double` an Stelle eines Zeigers auf `float` ist.
Wenn `h`,`l` oder `L` mit einem anderen Umwandlungszeichen auftreten, ist das Verhalten undefiniert.

– Ein **Umwandlungszeichen**, das den Typ der durchzuführenden Umwandlung angibt.

`fscanf()` führt jede Anweisung einzeln aus. Wenn eine Anweisung fehlschlägt, wie unten genauer erläutert, kehrt die Funktion zurück. Fehler werden als Eingabefehler bezeichnet, wenn Eingabezeichen fehlen, oder als Formatfehler, wenn unpassende Eingabezeichen auftreten.

Eine Anweisung, die aus einem Zwischenraumzeichen besteht, wird so ausgeführt, dass die Eingabe bis zum ersten Byte gelesen wird, das kein Zwischenraumzeichen ist und selbst nicht gelesen wird, oder bis keine Bytes mehr gelesen werden können.

Eine Anweisung, die aus einem normalen Zeichen besteht, wird so ausgeführt, dass die nächsten Bytes aus der Eingabe gelesen werden. Wenn eines dieser Bytes sich von dem Zeichen der Anweisung unterscheidet, so schlägt die Anweisung fehl und das unpassende und alle nachfolgenden Bytes werden nicht gelesen.

Eine Anweisung, die eine Umwandlungsanweisung ist, definiert eine Menge von passenden Eingabefolgen, wie dies unten für jede einzelne Umwandlungsanweisung beschrieben wird. Eine Umwandlungsanweisung wird in den folgenden Schritten ausgeführt:

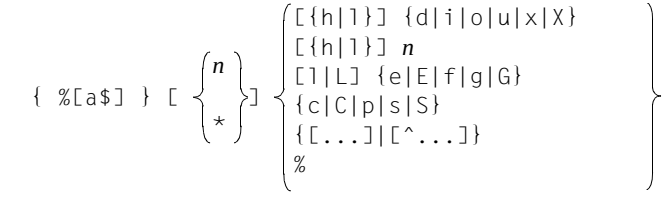
Die Eingabe von Zwischenraumzeichen wird überlesen, solange die Anweisung weder ein `[` noch eines der Umwandlungszeichen `c` oder *n* enthält.

Ein Eingabeelement wird aus der Eingabe gelesen, solange die Anweisung nicht das Umwandlungszeichen *n* enthält. Ein Eingabeelement ist definiert als die längste Folge von Eingabezeichen (bis zu einer eventuell angegebenen maximalen Feldbreite), die ein Anfang einer passenden Folge ist. Das erste Byte nach einem Eingabeelement bleibt, sofern es vorhanden ist, ungelesen. Wenn die Länge des Eingabeelements gleich 0 ist, schlägt die Ausführung der Anweisung fehl; diese Bedingung bedeutet einen Formatfehler, solange nicht ein Fehler weitere Eingaben verhindert, was dann einen Eingabefehler bedeutet.

Außer im Fall des Umwandlungszeichens `%` wird das Eingabeelement, bzw. im Fall einer Umwandlungsanweisung *%a* die Anzahl der Eingabezeichen, umgewandelt in einen Datentyp, der dem Umwandlungszeichen entspricht. Wenn das Eingabeelement keine passende Folge ist, schlägt die Ausführung dieser Anweisung fehl: Diese Bedingung ist ein Formatfehler. Wenn nicht die Unterdrückung der Zuweisung durch das Zeichen * angegeben wurde, wird das Ergebnis der Umwandlung in dem Objekt abgelegt, welches das erste auf

format folgende Argument ist, in dem bisher noch kein Umwandlungsergebnis abgelegt wurde. Wenn dieses Objekt nicht den passenden Datentyp hat oder wenn das Ergebnis der Umwandlung nicht in dem zur Verfügung stehenden Platz dargestellt werden kann, ist das Verhalten undefiniert.

Umwandlungsanweisungen können in XPG4-konformen Umgebungen wie folgt aussehen:



Umwandlungszeichen

`d` Liest eine dezimale Ganzzahl ein, die auch mit einem Vorzeichen versehen sein kann. Deren Format ist dasselbe, das auch `strtod()` mit dem Wert 10 für *base* erwartet. Das entsprechende Argument sollte ein Zeiger auf `int` sein.

`i` Liest eine dezimale Ganzzahl ein, die auch mit einem Vorzeichen versehen sein kann. Deren Format ist dasselbe, das auch `strtol()` mit dem Wert 0 für *base* erwartet. Das entsprechende Argument sollte vom Typ Zeiger auf `int` sein.

`o` Liest eine oktale Ganzzahl ein, die auch mit einem Vorzeichen versehen sein kann. Deren Format ist dasselbe, das auch `strtol()` mit dem Wert 8 für *base* erwartet. Das entsprechende Argument sollte vom Typ Zeiger auf `unsigned` sein.

`u` Liest eine dezimale Ganzzahl ein, die auch mit einem Vorzeichen versehen sein kann. Deren Format ist dasselbe, das auch `strtol()` mit dem Wert 10 für *base* erwartet. Das entsprechende Argument sollte vom Typ Zeiger auf `unsigned` sein.

`x, X` Liest eine hexadezimale Ganzzahl ein, die auch mit einem Vorzeichen versehen sein kann. Deren Format ist dasselbe, das auch `strtol()` mit dem Wert 16 für *base* erwartet. Das entsprechende Argument sollte vom Typ Zeiger auf `unsigned` sein.

`e, E, f, g, G` Diese Umwandlungszeichen lesen eine Gleitpunktzahl ein, die auch mit einem Vorzeichen versehen sein kann. Deren Format ist dasselbe, das auch `strtod()` erwartet. Das entsprechende Argument sollte vom Typ Zeiger auf `float` sein.

`s` Liest eine Folge von Bytes ein, die keine Zwischenraumzeichen sind. Das entsprechende Argument sollte ein Zeiger auf das erste Byte eines `char`-Vektors sein, der groß genug ist, um die Folge und ein abschließendes Nullbyte aufzunehmen, das automatisch angefügt wird.

`S` liest eine Zeichenkette ohne Zwischenraumzeichen. Diese Zeichenkette wird wie mit `mbstowcs()` in eine Folge von Langzeichen umgewandelt. Das entsprechende Argument muss ein Zeiger auf das erste Byte eines Feldes vom Typ `wchar_t` sein. Das Feld muss groß genug sein, um das Ergebnis der Umwandlung und ein abschließendes Nullbyte, das automatisch hinzugefügt wird, aufnehmen zu können. Wenn die Feldgröße festgelegt ist, bestimmt sie die maximal akzeptierte Anzahl von Zeichen.

Dieses Umwandlungszeichen wird nur im XPG4-Modus erkannt.

`[` Liest eine nichtleere Folge von Bytes aus einer Menge von erwarteten Bytes (der Eingabemenge). Das entsprechende Argument sollte ein Zeiger auf das erste Byte eines `char`-Vektors sein, der groß genug ist, um die Folge und ein abschließendes Nullbyte aufzunehmen, das automatisch angefügt wird. Die Umwandlungsanweisung schließt alle folgenden Bytes in der Zeichenkette *format* ein, einschließlich der zugehörigen schließenden eckigen Klammer (`]`). Die Bytes zwischen den Klammern stellen die Eingabemenge dar, solange nicht das erste Byte nach der linken Klammer das Zeichen `^` ist. In diesem Fall enthält die Eingabemenge alle Bytes, die nicht in der Liste zwischen dem Zeichen `^` und der Klammer `]` aufgeführt sind. Als Sonderfall gilt, dass die rechte eckige Klammer in den beiden Fällen, in denen die Umwandlungsanweisung mit den Zeichenketten `[]` bzw. `[^]` beginnt, zur Liste der Bytes gehört und erst die nächste rechte eckige Klammer diejenige ist, welche die Umwandlungsanweisung abschließt. Wenn das Zeichen – in der Liste auftritt und nicht das erste Zeichen bzw. das zweite Zeichen nach dem Zeichen `^` und auch nicht das letzte Zeichen ist, ist das Verhalten undefiniert.

`c` Liest eine Folge von Bytes, deren Anzahl durch die Feldbreite bzw. durch 1, wenn keine Feldbreite angegeben ist, bestimmt wird. Das entsprechende Argument sollte ein Zeiger auf das erste Byte eines `char`-Vektors sein, der groß genug ist, um die Folge aufzunehmen. Ein abschließendes Nullbyte wird nicht angefügt. Das normale Überlesen von Zwischenraumzeichen wird in diesem Fall unterdrückt; um das nächste Byte zu lesen, das kein Zwischenraumzeichen ist, sollte `%1s` verwendet werden.

`C` liest eine Zeichenkette der Länge, die durch die Feldgröße angegeben ist (1, wenn in dieser Anweisung keine Feldgröße angegeben ist). Die eingelesene Zeichenkette wird wie mit `mbstowcs()` in eine Folge von Langzeichen umgewandelt. Das entsprechende Argument muss ein Zeiger auf das erste

Byte eines Feldes vom Typ `wchar_t` sein. Das Feld muss groß genug sein, um das Ergebnis der Umwandlung aufnehmen zu können. Es wird kein Nullbyte hinzugefügt.

Wenn die Zeichenfolge im Anfangs-Shiftzustand beginnt, ist die Umwandlung entsprechend der `mbwstowcs()`-Funktion; andernfalls ist das Verhalten undefiniert. Das übliche Überspringen von Zwischenraumzeichen unterbleibt in diesem Fall.

`p` Liest eine Menge von Folgen, die denen entsprechen sollten, die von der Umwandlungsanweisung `%p` der `printf`-Funktionen erzeugt werden. `p` muss zu der Implementierung bei `printf`-Funktionen passen. Das entsprechende Argument sollte ein Zeiger auf einen Zeiger auf `void` sein. Die Interpretation des Eingabeelements ist jeweils implementierungsabhängig; für ein Eingabeelement, das nicht früher während derselben Programmausführung umgewandelt wurde, ist das Verhalten der Umwandlungsanweisung `%p` undefiniert. Dies gilt insbesondere für Zeigerausgaben, die von anderen Systemen erzeugt worden sind.

`n` Es wird keine Eingabe verarbeitet. Das entsprechende Argument sollte ein Zeiger auf `int` sein, in das die bisher von diesem Aufruf gelesene Zahl der Eingabezeichen eingetragen wird. Die Ausführung einer Anweisung des Typs *%a* erhöht nicht den Zuweisungszähler, der bei Beendigung der Ausführung der Funktion zurückgeliefert wird.

`%` Liest ein einzelnes `%`. Dabei findet keine Umwandlung oder Zuweisung statt. Die vollständige Umwandlungsanweisung lautet *%%*.

Wenn ein Umwandlungszeichen ungültig ist, ist das Verhalten `scanf()` undefiniert.

Wenn das Dateiende während der Eingabe gefunden wird, wird die Umwandlung abgebrochen. Wenn das Dateiende auftritt, bevor irgenwelche, zur aktuellen Anweisung passenden Bytes gelesen wurden (ungleich Zwischenraumzeichen, wo diese erlaubt sind), wird die Ausführung der aktuellen Anweisung mit einem Eingabefehler abgebrochen. Andernfalls wird, falls die Bearbeitung der aktuellen Anweisung nicht mit einem Formatfehler abbricht, die folgende Anweisung mit einem Eingabefehler abgebrochen, sofern diese vorhanden ist.

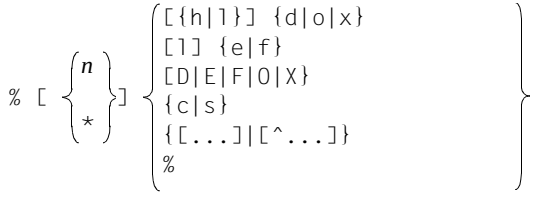
Wenn während eines `sscanf`-Aufrufs das Ende einer Zeichenkette erreicht wird, ist dies äquivalent zum Erreichen des Dateiendekennzeichens während eines `fscanf`-Aufrufs.

Wenn die Umwandlung wegen eines nicht passenden Zeichens abbricht, verbleibt dieses Byte ungelesen im Eingabestrom. Nachfolgende Zwischenraumzeichen (einschließlich der Zeilenendezeichen) bleiben ungelesen, solange dies nicht eine Anweisung macht. Der Erfolg des direkten Einlesens von Buchstaben und unterdrückten Zuweisungen kann nicht direkt bestimmt werden außer über die Anweisung *%a*.

BS2000

Umwandlungsanweisung (KR-Funktionalität)
(nur bei C/C++ Versionen kleiner V3 vorhanden)

Umwandlungsanweisungen enthalten Angaben, wie die Eingabefelder zu interpretieren und umzuwandeln sind. Sie können folgendermaßen aufgebaut sein:



Jede Umwandlungsanweisung muss mit einem Prozentzeichen (%) beginnen. Die restlichen Zeichen werden wie folgt interpretiert:

*	Überspringen einer Zuweisung. Das nächste Eingabefeld wird zwar gelesen und umgewandelt, aber in keiner Variablen abgespeichert.
n	Maximale Länge des umzuwandelnden Eingabefeldes. Tritt vorher ein Zwischenraumzeichen oder ein Zeichen auf, das nicht zur Typangabe in der Umwandlungsanweisung passt, wird die Länge entsprechend gekürzt.
`l`	`l` vor `d`, `o`, `x`: Umwandlung eines Arguments vom Typ Zeiger auf `long int` (`d`) bzw. `unsigned long int` (`o`, `x`). Die Angabe ist identisch mit den Großbuchstaben `D`, `O`, `X`. `l` vor `e`, `f`: Umwandlung eines Arguments vom Typ Zeiger auf `double`. Die Angabe ist identisch mit den Großbuchstaben `E`, `F`.
`h`	`h` vor `d`, `o`, `x`: Umwandlung eines Arguments vom Typ Zeiger auf `short int` (`d`) bzw. `unsigned short int` (`o`, `x`).
`d`	Ein dezimaler ganzzahliger Wert wird erwartet. Das entsprechende Argument muss ein Zeiger auf `int` sein.
`o`	Ein oktaler ganzzahliger Wert wird erwartet. Das entsprechende Argument kann ein Zeiger auf `unsigned int` oder `int` sein. Intern wird der Wert `unsigned` dargestellt.

`x` Ein hexadezimaler `int`-Wert wird erwartet. Das entsprechende Argument kann ein Zeiger auf `unsigned int` oder `int` sein. Intern wird der Wert `unsigned` dargestellt.

`e, f` Eine Gleitpunktzahl wird erwartet. Das entsprechende Argument muss ein Zeiger auf `float` sein. Die Gleitpunktzahl kann ein Vorzeichen enthalten sowie einen Exponenten (`E` bzw. `e`, gefolgt von einem ganzzahligen Wert). Das Dezimalzeichen wird durch die Lokalität (Kategorie `LC_NUMERIC`) beeinflusst. Voreingestellt ist der Punkt.

`c` Ein Zeichen wird erwartet. Das entsprechende Argument sollte ein Zeiger auf `char` sein. `scanf()` liest in diesem Fall auch Leerzeichen ein. Um das nächste Zeichen ungleich Leerzeichen einzulesen, ist `%1s` zu verwenden. `c` eignet sich dazu, Zeichenketten einzulesen, die auch Leerzeichen enthalten: Dazu ist als Argument ein Zeiger auf einen `char`-Vektor zu übergeben und eine Feldlänge *n* anzugeben (z.B. `%10c`). `scanf()` schließt in diesem Fall die Zeichenkette nicht automatisch mit dem Nullbyte ab.

`s` Eine Zeichenkette wird erwartet. Das entsprechende Argument muss ein Zeiger auf einen `char`-Vektor sein und groß genug, um die Zeichenkette und ein abschließendes Nullbyte aufnehmen zu können. `scanf()` schließt die Zeichenkette automatisch mit dem Nullbyte ab. Führende Zwischenraumzeichen in der Eingabe werden ignoriert, ein nachfolgendes Zwischenraumzeichen wird als Trennzeichen (Ende der Zeichenkette) interpretiert.

`[ ]` Eine Zeichenkette wird erwartet. Das entsprechende Argument muss ein Zeiger auf einen `char`-Vektor sein und groß genug, um die Zeichenkette inklusive des automatisch angefügten Nullbytes aufnehmen zu können. Im Unterschied zu `%s` fungieren bei dieser Angabe Leerzeichen nicht automatisch als Trennzeichen.

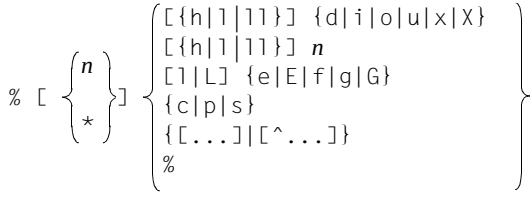
- `[...]` Bei dieser Angabe werden solange Zeichen eingelesen, bis das erste Zeichen auftritt, das nicht in den eckigen Klammern angegeben ist. D.h., die Zeichenkette darf nur aus den Zeichen in `[ ]` bestehen; alle anderen Zeichen gelten als Trennzeichen.
- `[^...]` Bei dieser Angabe werden solange Zeichen eingelesen, bis eines von den Zeichen auftritt, die in den eckigen Klammern nach `^` angegeben sind. Nur die in `[ ]` angegebenen Zeichen gelten als Trennzeichen.

`%` Eingabe des Zeichens `%`, keine Umwandlung. ❑

BS2000

Umwandlungsanweisung (ANSI-Funktionalität)

Umwandlungsanweisungen enthalten Angaben, wie die Eingabefelder zu interpretieren und umzuwandeln sind. Sie können folgendermaßen aufgebaut sein:



Führende Zwischenraumzeichen werden bei der Eingabe ignoriert.

Jede Umwandlungsanweisung muss mit einem Prozentzeichen (%) beginnen. Die restlichen Zeichen werden wie folgt interpretiert:

*	Überspringen einer Zuweisung. Das nächste Eingabefeld wird zwar gelesen und umgewandelt, aber in keiner Variablen abgespeichert.
n	Maximale Länge des umzuwandelnden Eingabefeldes. Tritt vorher ein Zwischenraumzeichen oder ein Zeichen auf, das nicht zur Typangabe in der Umwandlungsanweisung passt, wird die Länge entsprechend gekürzt.
`l`	`l` vor `d`, `i`, `o`, `u`, `x`, `X`: Umwandlung eines Arguments vom Typ Zeiger auf `long int` (`d`, `i`) bzw. `unsigned long int` (`o`, `u`, `x`, `X`). `l` vor `e`, `E`, `f`, `g`, `G`: Umwandlung eines Arguments vom Typ Zeiger auf `double`. `l` vor `n`: Das Argument ist vom Typ Zeiger auf `long int` (keine Umwandlung).
ll	`ll` vor `d`, `i`, `o`, `u`, `x`, `X` : Umwandlung eines Arguments vom Typ `long long int` bzw. `unsigned long long int`. `ll` vor `n`: Das Argument ist vom Typ Zeiger auf `long long int`.
`h`	`h` vor `d`, `i`, `o`, `u`, `x`, `X`: Umwandlung eines Arguments vom Typ Zeiger auf `short int` (`d`, `i`) bzw. `unsigned short int` (`o`, `u`, `x`, `X`). `h` vor `n`: Das Argument ist vom Typ Zeiger auf `short int` (keine Umwandlung).

`L` `L` vor `e`, `E`, `f`, `g`, `G`: Umwandlung eines Arguments vom Typ Zeiger auf `long double`.

`d` Ein dezimaler ganzzahliger Wert wird erwartet. Das entsprechende Argument muss ein Zeiger auf `int` sein.

`i` Ein ganzzahliger Wert wird erwartet. Die Basis (hexadezimal, oktal, dezimal) wird aus dem Aufbau des Eingabefeldes ermittelt. Führendes `0x` oder `0X`: hexadezimal. Führende `0`: oktal. Sonst: dezimal. Das entsprechende Argument muss ein Zeiger auf `int` sein.

`o` Ein oktaler `int`-Wert wird erwartet. Das entsprechende Argument kann ein Zeiger auf `unsigned int` oder `int` sein. Intern wird der Wert `unsigned` dargestellt.

`u` Ein dezimaler `int`-Wert wird erwartet. Das entsprechende Argument muss ein Zeiger auf `unsigned int` sein.

`x, X` Ein hexadezimaler `int`-Wert wird erwartet. Das entsprechende Argument kann ein Zeiger auf `unsigned int` oder `int` sein. Intern wird der Wert `unsigned` dargestellt.

`e, E, f, g, G`
Eine Gleitpunktzahl wird erwartet. Das entsprechende Argument muss ein Zeiger auf `float` sein. Die Gleitpunktzahl kann ein Vorzeichen enthalten sowie einen Exponenten (`E` bzw. `e`, gefolgt von einem ganzzahligen Wert). Das Dezimalzeichen wird durch die Lokalität (Kategorie `LC_NUMERIC`) beeinflusst. Voreingestellt ist der Punkt.

`c` Ein Zeichen wird erwartet. Das entsprechende Argument sollte ein Zeiger auf `char` sein. `scanf()` liest in diesem Fall auch Leerzeichen ein. Um das nächste Zeichen, das kein Leerzeichen ist, einzulesen, ist `%1s` zu verwenden. `c` eignet sich dazu, Zeichenketten einzulesen, die auch Leerzeichen enthalten: Dazu ist als Argument ein Zeiger auf einen `char`-Vektor zu übergeben und eine Feldbreite *n* anzugeben (z.B. `%10c`). `scanf()` schließt in diesem Fall die Zeichenkette nicht automatisch mit dem Nullbyte ab.

`p` Ein 8stelliger Zeigerwert wird erwartet, analog dem Format `%08.8x`. Das entsprechende Argument muss vom Typ Zeiger auf einen Zeiger auf `void` sein.

`s` Eine Zeichenkette wird erwartet. Das entsprechende Argument muss ein Zeiger auf einen `char`-Vektor sein und groß genug, um die Zeichenkette und ein abschließendes Nullbyte aufnehmen zu können. `scanf()` schließt die Zeichenkette automatisch mit dem Nullbyte ab. Führende Zwischenraumzeichen in der Eingabe werden ignoriert, ein nachfolgendes Zwischenraumzeichen wird als Trennzeichen (Ende der Zeichenkette) interpretiert.

`[]` Eine Zeichenkette wird erwartet. Das entsprechende Argument muss ein Zeiger auf einen `char`-Vektor sein und groß genug, um die Zeichenkette inklusive des automatisch angefügten Nullbytes aufnehmen zu können. Im Unterschied zu `%s` fungieren bei dieser Angabe Leerzeichen nicht automatisch als Trennzeichen.

`[...]` Bei dieser Angabe werden solange Zeichen eingelesen, bis das erste Zeichen auftritt, das nicht in den eckigen Klammern angegeben ist. D.h., die Zeichenkette darf nur aus den Zeichen in `[ ]` bestehen; alle nicht angegebenen Zeichen gelten als Trennzeichen.
Die schließende Klammer `]` kann in die Liste der einzulesenden Zeichen aufgenommen werden, wenn sie als erstes Zeichen unmittelbar nach der öffnenden Klammer angegeben wird: `[ ]...]`.

`[^...]` Bei dieser Angabe werden solange Zeichen eingelesen, bis eines von den Zeichen auftritt, die in den eckigen Klammern nach `^` angegeben sind. Nur die in `[ ]` angegebenen Zeichen gelten als Trennzeichen.
Die schließende Klammer `]` kann in die Liste der Trennzeichen aufgenommen werden, wenn sie als erstes Zeichen unmittelbar nach dem Zeichen `^` angegeben wird: `[^]...]`.

n Es werden keine Zeichen vom Eingabefeld gelesen. Das Argument ist vom Typ Zeiger auf `int`. Dieser ganzzahligen Variablen wird die Anzahl der Zeichen zugewiesen, die `scanf()` bis zu diesem Zeitpunkt verarbeitet hat.

`%` Eingabe des Zeichens `%`, keine Umwandlung.
❑

`fscanf()` und `scanf()` können die Strukturkomponente `st_atime` für die Datei aktualisieren, der *stream* zugeordnet ist. `st_atime` wird aktualisiert, sobald `fgetc()`, `fgets()`, `fgetwc()`, `fgetws()`, `fread()`, `fscanf()`, `getc()`, `getchar()`, `gets()` oder `scanf()` erfolgreich für *stream* aufgerufen werden und Daten zurückliefern, die nicht durch einen vorangegangenen Aufruf von `ungetc()` oder `ungetwc()` bereitgestellt wurden.

Returnwert Anzahl der erfolgreich gelesenen und zugewiesenen Eingabeelemente bei erfolgreicher Beendigung.

0 wenn gleich zu Beginn ein nicht zur Formatzeichenkette passendes Eingabezeichen gefunden wird.

`EOF` wenn die Eingabe vor dem ersten Konflikt zwischen Eingabezeichen und Formatzeichenkette oder vor der ersten Umwandlung endet. Im Unterschied zu XPG4 wird `errno` nicht gesetzt.

Hinweis

Wenn die Anwendung, die `fprintf()` aufruft, Objekte des Typs `wchar_t` enthält, muss sie auch `sys/types.h` oder `stddef.h` einbinden, damit dieser Datentyp definiert ist.

In Formatzeichenketten, die die Form `%` für die Umwandlungs-Anweisungen enthalten, wird jedes Argument der Argumentliste genau einmal verwendet. In Formatzeichenketten, die

die Form mit `%`*a*`$` für die Umwandlungs-Anweisungen enthalten, kann jedes nummerierte Argument der Argumentliste so oft verwendet werden, wie nötig.

Bei der Umwandlung von `int`-Werten in `unsigned int` (Umwandlungszeichen `o`, `u`, `x`, `X`) wird aus einem Wert mit negativem Vorzeichen das Zweierkomplement gebildet.
Z.B. liefert Format `%u` bei der Eingabe `-1` `X'FFFFFFFF'`.

Den Returnwert eines `scanf`-Aufrufes sollten Sie immer abfragen, um sicher zu sein, dass kein Fehler passiert ist!

Der nächste `scanf`-Aufruf startet mit dem Lesen unmittelbar nach dem Zeichen, das als letztes vom vorherigen Aufruf verarbeitet wurde.

Wenn ein Eingabezeichen nicht dem angegebenen Format entspricht, wird es in den Eingabepuffer zurückgeschrieben. Es muss dort mit `getc()` abgeholt werden, sonst erhält der nächste `scanf`-Aufruf dasselbe Zeichen noch einmal.

Ob `fscanf()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

Siehe auch `getc()`, `printf()`, `setlocale()`, `strtod()`, `strtol()`, `langinfo.h`, `stdio.h`.

fseek - Lese-/Schreibzeiger im Datenstrom auf aktuellen Wert positionieren

Name **fseek, fseek64, fseeko, fseeko64**

Syntax #include <stdio.h>

int fseek(FILE *_stream_, long int _offset_, int _whence_);
int fseek64(FILE *_stream_, long long int _offset_, int _whence_);
int fseeko(FILE *_stream_, off_t _offset_, int _whence_);
int fseeko64(FILE *_stream_, off64_t _offset_, int _whence_);

Beschreibung

Wenn POSIX-Dateien ausgeführt werden, verhält sich die Funktion XPG-konform wie folgt:

`fseek()` setzt den Lese-/Schreibzeiger für den Datenstrom, auf den *stream* zeigt.

Die neue Position, gemessen in Bytes vom Anfang der Datei, wird durch die Addition von *offset* zu der durch *whence* angegebenen Position ermittelt. Der angegebene Punkt ist der Dateianfang bei `SEEK_SET`, der aktuelle Wert des Lese-/Schreibzeigers bei `SEEK_CUR` oder das Dateiende bei `SEEK_END`.

Sollen Ein-/Ausgabe-Funktionen für Langzeichen auf *stream* angewendet werden, muss *offset* entweder `0` oder der Rückgabewert eines vorhergehenden `ftell`-Aufrufs in demselben Datenstrom sein; *whence* muss `SEEK_SET` sein

Ein erfolgreicher Aufruf von `fseek()` löscht das Kennzeichen für Dateiende und hebt jede Wirkung von `ungetc()` und `ungetwc()` für denselben Datenstrom auf. Nach einem Aufruf von `fseek()` kann die nächste Operation auf einem zum Aktualisieren geöffneten Datenstrom sowohl eine Eingabe- als auch eine Ausgabeoperation sein.

Wenn auf einem gegebenen Datenstrom die letzte Operation ungleich `ftell()` die Operation `fflush()` ist, dann wird die Dateiposition der zu Grunde liegenden Dateibeschreibung angepasst, um den durch `fseek()` vorgegebenen Ort widerzuspiegeln.

`fseek()` erlaubt, dass der Lese-/Schreibzeiger hinter das Ende der existierenden Daten in der Datei gesetzt wird. Werden später Daten an diese Stelle geschrieben, dann liefern anschließende Leseoperationen für Daten in dieser Lücke Nullbytes, bis wirklich Daten in diese Lücke geschrieben wurden.

Ist der Datenstrom zum Schreiben geöffnet und wurden gepufferte Daten noch nicht in die zu Grunde liegende Datei geschrieben, bewirkt `fseek()`, dass die noch nicht geschriebenen Daten in die Datei geschrieben werden und markiert die Felder `st_ctime` und `st_mtime` der Datei zum Aktualisieren.

Die Funktion `fseek64()` verhält sich wie `fseek()`, außer dass bei f`seek64()` der Offset-Typ `long long` verwendet wird.

Es besteht kein funktionaler Unterschied zwischen `fseeko()` und `fseeko64()`, außer dass `fseeko64()` die Struktur `off64_t` verwendet.
Die Funktion `fseeko()` entspricht der modifizierten Funktion `fseek()`, mit der Ausnahme, dass das Offset-Argument den Typ `off_t` hat und der Fehler EOVERFLOW sich geändert hat.

BS2000
Wenn BS2000-Dateien ausgeführt werden, ist Folgendes zu beachten:

`fseek()` positioniert den Lese-/Schreibzeiger für die Datei mit *stream* gemäß den Angaben in *offset* und *whence*. Damit ist die Möglichkeit gegeben, eine Datei nicht-sequenziell zu bearbeiten.

In Textdateien (SAM im Textmodus, ISAM) kann man absolut auf Dateianfang und -ende positionieren sowie auf eine vorher mit `ftell()` gemerkte Position.

In Binärdateien (SAM im Binärmodus, PAM, INCORE) kann sowohl absolut (s.o.) als auch relativ um eine gewünschte Anzahl Bytes, bezogen auf Dateianfang, aktuelle Position oder Dateiende positioniert werden.

Für die Parameter *offset* und *whence* sind Bedeutung, Kombinationsmöglichkeiten und Wirkung für Text- und Binärdateien unterschiedlich und werden deshalb im Folgenden getrennt beschrieben:

Textdateien (SAM im Textmodus, ISAM)

Mögliche Werte:

offset 0L oder Wert, der durch einen vorhergehenden *ftell*-Aufruf ermittelt wurde.

whence `SEEK_SET` (Dateianfang)
`SEEK_END` (Dateiende)

Sinnvolle Kombinationsmöglichkeiten und Wirkung:

offset	*whence*	**Wirkung**
`ftell`-Wert	`SEEK_SET`	Positionieren auf die durch `ftell()` ermittelte Position.
0L	`SEEK_SET`	Positionieren auf Dateianfang.
0L	`SEEK_END`	Positionieren auf Dateiende.

Binärdateien (SAM im Binärmodus, PAM, INCORE)

Mögliche Werte:

offset Anzahl der Bytes, um die der aktuelle Lese-/Schreibzeiger verschoben werden soll, und zwar

positive Zahl: Vorwärts positionieren Richtung Dateiende
negative Zahl: Rückwärts positionieren Richtung Dateianfang
0L: absolut Positionieren auf Dateianfang bzw. -ende.

whence Bei absoluter Positionierung auf Dateianfang oder -ende, Zielpunkt, auf den der Lese-/Schreibzeiger verschoben werden soll und
bei relativer Positionierung, Bezugspunkt, von dem aus der Lese-/Schreibzeiger um *offset* Bytes verschoben werden soll:
`SEEK_SET` (Dateianfang)
`SEEK_CUR` (aktuelle Position)
`SEEK_END` (Dateiende)

Sinnvolle Kombinationsmöglichkeiten und Wirkung:

offset	*whence*	**Wirkung**
0L	`SEEK_SET`	Positionieren auf Dateianfang.
0L	`SEEK_END`	Positionieren auf Dateiende.

positive Zahl	`SEEK_SET`
	`SEEK_CUR`
	`SEEK_END`
		Vorwärts positionieren ab Dateianfang,
		ab aktueller Position,
		ab Dateiende (über das Dateiende hinaus).

negative Zahl	`SEEK_CUR` `SEEK_END`	Rückwärts positionieren ab aktueller Position, ab Dateiende.
`ftell`-Wert	`SEEK_SET`	Positionieren auf die durch einen `ftell`-Aufruf gemerkte Position.

Returnwert 0 bei Erfolg.

-1 Ein Positionieren auf der angegebenen Datei ist nicht möglich. `errno` wird gesetzt, um den Fehler anzuzeigen.

Ungültiges Positionieren kann beispielsweise ein `fseek()` auf eine Datei sein, die nicht über `fopen()` geöffnet wurde. Insbesondere darf `fseek()` nicht für ein Terminal oder für eine Datei verwendet werden, die über `popen()` geöffnet wurde. Nachdem ein Datenstrom geschlossen wurde, sind keine weiteren Operationen auf diesem Datenstrom definiert.

`fseek()` und `fseeko()` schlagen fehl, wenn entweder der Datenstrom nicht gepuffert ist oder der Puffer geleert werden muss und durch den `fseek()` bzw. `fseeko()`-Aufruf ein zu Grunde liegendes `lseek()` oder `write()` aufgerufen wird:

`EAGAIN` Das Kennzeichen `O_NONBLOCK` für den *stream* zu Grunde liegenden Dateideskriptor ist gesetzt, und eine Schreiboperation würde den Prozess verzögern.

`EBADF` Der *stream* zu Grunde liegende Dateideskriptor ist nicht zum Schreiben geöffnet oder der Puffer des Datenstroms muss geleert werden, und die Datei ist nicht geöffnet.

`EFBIG` Es wurde versucht, in eine Datei zu schreiben, deren Größe die maximale Dateigröße oder die Grenze des Prozesses für die Dateigröße überschreitet (siehe auch `ulimit()`).

`EINTR` Die Schreiboperation wurde durch den Empfang eines Signals beendet, und es wurden keine Daten übertragen.

`EINVAL` *whence* ist ein ungültiges Argument. Der sich daraus ergebende Wert des Lese-/Schreibzeigers ist negativ.

`EIO` Ein Ein-/Ausgabefehler ist aufgetreten.

Der Prozess ist Mitglied einer Hintergrund-Prozessgruppe und versucht auf das steuernde Terminal zu schreiben, `TOSTOP` ist gesetzt, das Signal `SIGTTOU` wird vom Prozess weder ignoriert noch blockiert und die Prozessgruppe des Prozesses ist verwaist.

`ENOSPC` Auf dem Datenträger, auf dem sich die Datei befindet, ist kein freier Platz mehr vorhanden.

`EPIPE` Es wurde der Versuch unternommen, auf eine Pipe oder FIFO zu schreiben, die von keinem Prozess zum Lesen geöffnet war. Außerdem wird das Signal `SIGPIPE` an den Prozess gesendet.

Werden Threads verwendet, so wirkt sich die Funktion auf den Prozess oder auf einen Thread wie folgt aus: Beim `EPIPE`-Fehler wird das Signal `SIGPIPE` nicht an den Prozess, sondern an den aufrufenden Thread gesendet.

`ENXIO` Das Gerät existiert nicht oder es kann darauf nicht zugegriffen werden.

`EOVERFLOW` Für *fseek()*: der resultierende Datei-Offset-Wert kann in einem Objekt des Typs long nicht korrekt dargestellt werden.

`EOVERFLOW` Für *fseeko()*: der resultierende Datei-Offset-Wert kann in einem Objekt des Typs `off_t` nicht korrekt dargestellt werden.

Hinweis

Obwohl in POSIX-Dateien eine von `ftell()` zurückgegebene Dateiposition in Bytes gemessen wird und es zulässig ist, relativ zu dieser Dateiposition zu positionieren, erfordert die Portabilität auf andere Systeme, dass `fseek()` eine direkte Dateiposition (d.h. den von `ftell()` zurückgegebenen Wert) erhält. Arithmetische Operationen an einer anderen Dateiposition, die nicht unbedingt in Bytes gemessen wird, können nicht immer sinnvoll ausgeführt werden.

Ob `fseek()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

BS2000
Der Aufruf `fseek(`*stream*`,0L,SEEK_SET)` ist äquivalent zu dem Aufruf `rewind(`*stream*`)`.

Werden in eine Textdatei neue Sätze geschrieben (geöffnet zum Neuerstellen oder Anfügen) und erfolgt ein `fseek`-Aufruf, dann werden zunächst ggf. restliche Daten aus dem Puffer in die Datei geschrieben und mit Zeilenende (`\n`) abgeschlossen.

Ausnahme bei ANSI-Funktionalität:
Wenn die Daten einer ISAM-Datei im Puffer nicht mit einem Zeilenendezeichen enden, bewirkt `fseek()` keinen Zeilenwechsel bzw. Satzwechsel. D.h., die Daten werden beim Schreiben aus dem Puffer nicht automatisch mit einem Zeilenendezeichen abgeschlossen. Nachfolgende Daten verlängern den Satz in der Datei. Beim Lesen einer ISAM-Datei werden daher nur Zeilenendezeichen eingelesen, die vom Programm explizit geschrieben wurden.

Wird bei einer Binärdatei hinter das Dateiende positioniert, entsteht eine Lücke zwischen den letzten physisch gespeicherten Daten und den neu geschriebenen Daten. Lesen aus dieser Lücke liefert binäre Nullen.

Auf Systemdateien (SYSDTA, SYSLST, SYSOUT) kann nicht positioniert werden.

Ein erfolgreicher Aufruf von `fseek()` löscht das Kennzeichen `EOF` der Datei und hebt die Wirkung der vorangegangenen `ungetc`-Aufrufe für diese Datei auf.

Bei Satz-E/A kann `fseek()` nur zum Positionieren auf Dateianfang oder Dateiende benutzt werden.

`fseek(`*stream*`,0L,SEEK_SET)` positioniert auf den ersten Satz der Datei.

`fseek(`*stream*`,0L,SEEK_END)` positioniert hinter den letzten Satz der Datei.

Bei Aufrufen mit anderen Argumenten liefert `fseek() EOF`.

Siehe auch `fopen()`, `fsetpos()`, `ftell()`, `lseek()`, `rewind()`, `tell()`, `ungetc()`, `stdio.h`.

fsetpos - Lese-/Schreibzeiger im Datenstrom auf aktuellen Wert positionieren

Name **fsetpos, fsetpos64**

Syntax

```
#include <stdio.h>

int fsetpos(FILE *stream, const fpos_t *pos);
int fsetpos64(FILE *stream, const fpos64_t *pos);
```

Beschreibung

`fsetpos()` positioniert den Lese-/Schreibzeiger der Datei mit Dateizeiger *stream* auf eine zuvor mit `fgetpos()` ermittelte Position *pos*.

`fsetpos()` löscht das Dateiendekennzeichen für den Datenstrom und macht jede Wirkung der Funktion `ungetc()` auf den Datenstrom rückgängig. Nach `fsetpos()` können mit einer änderbaren Datei Ein- oder Ausgabeoperationen durchgeführt werden.

Es besteht kein funktioneller Unterschied zwischen `fsetpos()` und `fsetpos64()`, außer dass `fsetpos64()` einen `fpos64_t` Typ verwendet.

Returnwert

0	bei Erfolg.
≠ 0	bei Fehler.
	BS2000 `errno` wird auf `EBADF` gesetzt.

Hinweis

Ob `fsetpos()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

BS2000
`fsetpos()` lässt sich auf Binärdateien (SAM im Binärmodus, PAM, INCORE) und Textdateien (SAM im Textmodus, ISAM) anwenden. `fsetpos()` ist nicht anwendbar auf Systemdateien (SYSDTA, SYSLST, SYSOUT).

Ein erfolgreicher Aufruf der Funktion `fsetpos()` löscht das Kennzeichen `EOF` der Datei und hebt die Wirkung der vorangegangenen `ungetc`-Aufrufe für diese Datei auf.

Werden in eine Textdatei neue Sätze geschrieben (geöffnet zum Neuerstellen oder Anfügen) und erfolgt ein `fsetpos`-Aufruf, dann werden zunächst ggf. restliche Daten aus dem Puffer in die Datei geschrieben und mit dem Zeilenendezeichen (`\n`) abgeschlossen.

Ausnahme bei ANSI-Funktionalität:
Wenn die Daten einer ISAM-Datei im Puffer nicht mit einem Zeilenendezeichen abgeschlossen sind, bewirkt `fsetpos()` keinen Zeilenwechsel bzw. Satzwechsel. D.h., die Daten werden beim Schreiben aus dem Puffer nicht automatisch mit einem Zeilenendezei-

chen abgeschlossen. Nachfolgende Daten verlängern den Satz in der Datei. Beim Lesen einer ISAM-Datei werden daher nur Zeilenendezeichen eingelesen, die vom Programm explizit geschrieben wurden.

Nach der Positionierung kann die nächste Operation sowohl eine Lese- als auch eine Schreiboperation sein.

Für ISAM-Dateien ist das Funktionspaar `fgetpos()`/`fsetpos()` wesentlich performanter als das vergleichbare Funktionspaar `ftell()`/`fseek()`.

Bei Satz-Ein-/Ausgabe in ISAM-Dateien mit Schlüsselverdoppelung kann mit `fsetpos()` nicht auf den zweiten oder höheren Schlüssel einer Gruppe mit gleichen Schlüsseln positioniert werden. Dies lässt sich nur durch sequenzielles Lesen bzw. Löschen erreichen. Mit `fsetpos()` kann nur auf den ersten Satz oder hinter den letzten Satz einer solchen Gruppe positioniert werden.

Siehe auch `fgetpos()`, `fseek()`, `ftell()`, `open()`, `rewind()`, `ungetc()`, `stdio.h`.

fstat - Status einer offenen Datei abfragen

Name **fstat, fstat64**

Syntax #include <sys.stat.h>

Optional
#include <sys/types.h>

int fstat(int *fildes*, struct stat **buf*);
int fstat64(int *fildes*, struct stat64 **buf*);

Beschreibung

`fstat()` liefert Informationen über eine offene Datei mit einem Dateideskriptor *fildes*, der von einem erfolgreichen Systemaufruf `open()`, `creat()`, `dup()`, `fcntl()` oder `pipe()` geliefert wird.

buf ist ein Zeiger auf eine `stat`-Struktur, in die Informationen geschrieben werden, die die jeweilige Datei betreffen.

Es besteht kein funktionaler Unterschied zwischen `fstat()` und f`stat64()`, außer dass bei `fstat64()` der File Status in einer `stat64`-Struktur zurückgegeben wird.

Zum Inhalt der Struktur, auf die *buf* zeigt, gehören folgende Elemente:

```
mode_t    st_mode;    /* Dateimodus (siehe mknod()) */
ino_t     st_ino;     /* Dateikennziffer (i-Node) */
dev_t     st_dev;     /* Gerätekennung, die einen
                         Verzeichniseintrag für diese Datei enthält */
dev_t     st_rdev;    /* Gerätekennung, nur für zeichen- oder
                         blockorientierte Gerätedateien definiert */
nlink_t   st_nlink;   /* Anzahl der Verweise */
uid_t     st_uid;     /* Benutzerkennung des Dateibesitzers */
gid_t     st_gid;     /* Gruppenkennung des Dateibesitzers */
off_t     st_size;    /* Dateigröße in Bytes */
time_t    st_atime;   /* Zeit des letzten Zugriffs */
time_t    st_mtime;   /* Zeit der letzten Datenänderung */
time_t    st_ctime;   /* Zeit der letzten Änderung des Dateistatus
                         Die Zeit wird in Sekunden gemessen ab dem
                         1. Januar 1970, 00:00:00 Uhr */
long      st_blksize; /* Bevorzugte E/A-Blockgröße */
blkcnt_t   st_blocks;   /* Anzahl zugewiesener st_blksize-Blöcke */
```

Die Struktur `stat64` ist wie die von `stat` definiert, mit Ausnahme folgender Komponenten:

```
ino64_t st_ino
off64_t st_size und
blkcnt64_t st_blocks
```

Die einzelnen Elemente haben die folgende Bedeutung:

`st_mode` Der Modus der Datei ist im Systemaufruf `mknod( )` beschrieben. Zusätzlich zu den in `mknod()` beschriebenen Modi, kann der Modus einer Datei auch `S_IFLNK` sein, wenn die Datei ein symbolischer Verweis ist, oder `S_IFSOCK`, wenn es sich um einen Socket-Deskriptor handelt.

`st_ino` kennzeichnet die Datei im gegebenen Dateisystem eindeutig. Das Paar `st_ino` und `st_dev` kennzeichnet normale Dateien eindeutig.

`st_dev` kennzeichnet das Dateisystem, in dem die Datei liegt, eindeutig.

`st_rdev` darf nur von Verwaltungskommandos benutzt werden. Es ist nur für block- oder zeichenorientierte Dateien gültig und hat nur in dem System eine Bedeutung, in dem die Datei eingereicht wurde.

`st_nlink` darf nur von Verwaltungskommandos benutzt werden.

`st_uid` Benutzernummer des Eigentümers der Datei.

`st_gid` Gruppennummer der Gruppe, der die Datei zugeordnet ist.

`st_size` Für normale Dateien ist dies die Größe der Datei in Bytes. Für block- oder zeichenorientierte Dateien ist dieses nicht definiert.

`st_atime` Uhrzeit, zu der zuletzt auf die Daten der Datei zugegriffen wurde. Wird von folgenden Systemaufrufen geändert: `creat(), mknod(), pipe(), utime()` und `read()`.

`st_mtime` Uhrzeit, zu der Daten zuletzt geändert wurden. Wird von folgenden Systemaufrufen geändert: `creat(), mknod(), pipe(), utime()` und `write()`.

`st_ctime` Uhrzeit, zu der der Dateistatus zuletzt geändert wurde. Wird von folgenden Systemaufrufen geändert: `chmod()`, `chown()`, `creat()`, `link()`, `mknod()`, `pipe()`, `unlink()`, `utime()` und `write()`.

`st_blksize` Ein Hinweis auf die ’beste’ Größe einer Einheit für Ein-/Ausgabe-Operationen. Dieses Feld ist für block- oder zeichenorientierte Gerätedateien nicht definiert.

`st_blocks` Die Gesamtanzahl von physikalischen Blöcken der Größe 512 Bytes, die zurzeit auf der Platte belegt sind. Dieses Feld ist für block- oder zeichenorientierte Gerätedateien nicht definiert.

BS2000

Bei BS2000-Dateien werden folgende Elemente der `stat`-Struktur gesetzt:

mode_t *st_mode*	Dateimodus, der Zugriffsrechte und Dateityp beinhaltet. Zugriffsrechte: Hier wird das Basic ACL auf die Dateischutzbits abgebildet. Die Schutzbits sind alle 0, wenn die Datei keinen Basic ACL Schutz hat. Dateityp: Einführung eines neuen Dateityps S_IFDVSBS2=X'40000000'. Dieser Typ ist allerdings nicht disjunkt zu S_IFPOSIXBS2. Abgefragt kann mit dem Makro S_ISDVSBS2(*mode*) werden.
time_t *st_atime*	Zeitpunkt des letzten Zugriffs wie im BS2000 üblich (last access time), aber in Sekunden seit dem 1.1.1970 UTC.
time_t *st_mtime*	Zeitpunkt der letzen Änderung (last modification time).
time_t *st_ctime*	Zeitpunkt der Erzeugung (creation time).
long *st_blksize*	Blockgröße, 2k (d.h. 1 PAM Page).
long *st_blocks*	Anzahl der von der Datei belegten Blöcke auf der Platte.
dev_t *st_dev*	enthält die 4 Byte lange `catid`.

Die beiden hintereinander liegenden Felder

uid_t *st_uid*	und
gid_t *st_uid*	enthalten die 8 Byte lange BS2000-Userid.

Alle anderen Felder werden auf 0 gesetzt.

Returnwert

0	bei Erfolg.
-1	bei Fehler. Für POSIX-Dateien wird `errno` gesetzt, um den Fehler anzuzeigen.

Fehler `fstat()` schlägt fehl, wenn gilt:

`EBADF`	*fildes* ist kein gültiger Dateideskriptor.
`EFAULT`	*buf* weist auf eine ungültige Adresse
`EIO`	Beim Lesen des Dateisystems trat ein E/A-Fehler auf.
`ENOLINK`	*fildes* weist auf einen fernen Rechner zu dem die Verbindung nicht mehr aktiv ist.
`EOVERFLOW`	Eine Komponente ist zu groß und kann nicht in die Struktur, auf die *buf* zeigt, gespeichert werden.

`EINTR` Ein Signal wurde während des Systemaufrufs `fstat()` abgefangen.

Hinweis `fstat()` wird jetzt auch für BS2000-Dateien ausgeführt.

Siehe auch `chmod()`, `chown()`, `creat()`, `link()`, `lstat()`, `mknod()`, `stat()`, `unlink()`, `write()`, `sys/stat.h`, `sys/types.h`.

fstatvfs, statvfs - Dateisystem-Informationen lesen

Name **fstatvfs, fstatvfs64, statvfs, statvfs64**

Syntax
```
#include <sys/statvfs.h>
#include <sys/types.h>

int fstatvfs (int fildes, struct statvfs *buf);
int statvfs (const char *path, struct statvfs *buf);
int fstatvfs64 (int fildes, struct statvfs64 *buf);
int statvfs64 (const char *path, struct statvfs64 *buf);
```

Beschreibung

`fstatvfs()` liefert Informationen über das Dateisystem, zu dem die mit *fildes* bezeichnete Datei gehört. *buf* ist ein Zeiger auf eine Struktur, die weiter unten beschrieben wird. In diese Struktur werden während des Systemaufrufs die Informationen über das Dateisystem eingetragen.

fildes bezeichnet einen offenen Dateideskriptor, der aus einem erfolgreichen `open()`, `creat()`-, `dup()`-, `fcntl()`- oder `pipe()`-Systemaufruf resultiert. Der Typ des Dateisystems, das die *fildes* zugeordnete Datei enthält, ist dabei dem Betriebssystem bekannt. Lese-, Schreib- oder Ausführungsrechte für die angegebene Datei werden nicht benötigt.

Es besteht kein funktionaler Unterschied zwischen `fstatvfs()` / `statvfs()` und `fstatvfs64() / statvfs64()`, außer dass bei `fstatvfs64() und statvfs64()` der File Status jeweils in einer `statvfs64` Struktur zurückgegeben wird.

Die Struktur `statvfs`, auf die *buf* zeigt, enthält die folgenden Komponenten:

```
ulong_t f_bsize;            /* bevorzugte Blockgröße des Dateisystems */
ulong_t f_frsize;           /* grundlegende Blockgröße des Dateisystems
                               (falls unterstützt) */
fsblkcnt_t f_blocks;        /* gesamte Anzahl der Blöcke auf dem
                               Dateisystem in Einheiten von f_frsize */
fsblkcnt_t f_bfree;         /* gesamte Anzahl der freien Blöcke */
fsblkcnt_t f_bavail;        /* Anzahl der verfügbaren freien Blöcke
                               für einen Nicht-Systemverwalter */
fsfilcnt_t f_files;         /* gesamte Anzahl der Dateien (Inodes) */
fsfilcnt_t f_ffree;         /* gesamte Anzahl der freien Knoten */
fsfilcnt_t f_favail;        /* Anzahl der Inodes für einen
                               Nicht-Systemverwalter*/
ulong_t f_fsid;             /* Dateisystemnummer (momentan dev) */
char    f_basetype[FSTYPSZ]; /* Typname des Zieldateisystems,
                               nullterminiert */
ulong_t f_flag;             /* Bitmaske der Optionen */
```

```
ulong_t   f_namemax;         /* maximale Länge der Dateinamen */
char      f_fstr[32];        /* Dateisystemspezifische Zeichenkette */
ulong_t    f_filler[16];        /* reserviert für zukünftige Erweiterungen */
```

Die Struktur `statvfs64` unterscheidet sich von `statvfs` durch folgende Komponeneten:

```
fsblkcnt64_t f_blocks
fsblkcnt64_t f_bfree
fsblkcnt64_t f_bavail
fsfilcnt64_t f_files
fsfilcnt64_t f_ffree
fsfilcnt64_t f_favail
```

`f_basetype` enthält einen nullterminierten Typnamen des Dateisystems (FST-Name) über das eingehängte Ziel (z.B. `s5` über `rfs` eingehängt resultiert in `s5`).

Die folgenden Werte können in der Komponente `f_flag` zurückgeliefert werden:

```
ST_RDONLY     0x01    /* schreibgeschütztes Dateisystem */
ST_NOSUID     0x02    /* setuid/setgid Semantik wird nicht unterstützt */
ST_NOTRUNC    0x04    /* schneidet Dateinamen länger als NAME_MAX nicht ab */
```

`statvfs()` arbeitet genauso wie `fstatvfs()`, außer dass die Datei über den Pfadnamen angesprochen wird, auf den *path* verweist. Für jedes Verzeichnis aus dem Pfadnamen muss Sucherlaubnis vorhanden sein.

Returnwert

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

Fehler

`fstatvfs() und statvfs()` schlagen fehl, wenn gilt:

`EIO`	Beim Lesen des Dateisystems trat ein E/A-Fehler auf.
`EINTR`	Während der Ausführung der Funktion wurde ein Signal empfangen.

`fstatvfs()` schlägt fehl, wenn gilt:

`EBADF`	*fildes* ist kein geöffneter Dateideskriptor.
`EOVERFLOW`	Einer der zurückgegebenen Werte kann in der Struktur, auf die *buf* zeigt, nicht korrekt dargestellt werden.

`statvfs()` schlägt fehl, wenn gilt:

`EACCES`	Sucherlaubnis existiert für eine Komponente des Pfadpräfixes nicht.
`ELOOP`	Zu viele symbolische Verweise traten bei der Übersetzung von *path* auf.

ENAMETOOLONG
: Der Pfadname, auf den *path* zeigt, ist länger als {PATH_MAX}, oder die Länge einer Komponente des Pfadnamens überschreitet {NAME_MAX}.

ENOENT
: Eine Komponente des Pfadnamens existiert nicht, oder *path* zeigt auf eine leere Zeichenkette.

ENOTDIR
: Eine Komponente des Pfadpräfixes von *path* ist kein Verzeichnis.

statvfs() schlägt fehl, wenn gilt:

ENAMETOOLONG
: Die Auflösung symbolischer Verweise im Pfadnamen führt zu einem Zwischenergebnis, dessen Länge {PATH_MAX} überschreitet.

Hinweis Nicht alle Elemente der Struktur statvfs sind in allen Dateisystemen belegt.

Siehe auch chmod(), chown(), creat(), dup(), exec, link(), mknod(), pipe(), read(), time(), unlink(), utime(), write(), sys/statvfs.h.

fsync - Dateiänderungen synchronisieren

Syntax #include <unistd.h>

int fsync(int *fildes*);

Beschreibung

`fsync()` schreibt alle modifizierten Daten und Attribute von *fildes*, die sich in Puffern befinden, auf das physikalische Speichermedium.

Returnwert

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

Fehler `fsync()` schlägt fehl, wenn gilt:

`EBADF`	*fildes* ist kein gültiger Dateideskriptor.
`EINTR`	Während des Systemaufrufs wurde `fsync()` von einem Signal unterbrochen.
`EINVAL`	*fildes* bezieht sich auf eine Datei, für die diese Operation nicht durchführbar ist. Es wurde versucht, auf eine BS2000-Datei zuzugreifen.
`EIO`	Ein Ein-/Ausgabefehler trat während des Lesens oder Schreibens im Dateisystem auf.

Hinweis `fsync()` sollte von Programmen verwendet werden, die ihre Ausführung erst fortsetzen, wenn die Modifikation einer Datei abgeschlossen ist. Enthält ein Programm beispielsweise eine einfache Transaktionsmöglichkeit, kann mit `fsync()` sichergestellt werden, dass alle transaktionsbedingten Änderungen an einer oder mehreren Dateien durchgeführt werden.

`fsync()` wird nur für POSIX-Dateien ausgeführt.

Siehe auch `unistd.h`.

ftell - aktuellen Wert des Lese-/Schreibzeigers im Datenstrom ermitteln

Name **ftell, ftell64, ftello, ftello64**

Syntax

```
#include <stdio.h>

long int ftell(FILE *stream);
long long ftell64(FILE *stream);
off_t ftello(FILE *stream);
off64_t ftello64(FILE *stream);
```

Beschreibung

`ftell()` und `ftello()` ermitteln den aktuellen Wert des Lese-/Schreibzeigers für den Datenstrom, auf den *stream* zeigt. Auf diesen Wert kann mit `fseek()/fseeko()` positioniert werden.

Die Funktion `ftello()` entspricht der modifizierten Funktion `ftell()`, mit der Ausnahme, dass das Offset-Argument den Typ `off_t` hat und der Fehler `EOVERFLOW` sich geändert hat.

Es besteht kein funktionaler Unterschied zwischen `ftell()` und `ftell64()`, außer dass `ftell64()` den Offset-Typ `long long` verwendet.

`ftello64()` ist wie `ftello()` definiert, außer dass `ftello64()` den Offset-Typ `off64_t` verwendet.

Returnwert

aktueller Wert des Lese-/Schreibzeigers
: für den Datenstrom, d.h. die Anzahl der Bytes, die der Lese-/Schreibzeiger vom Dateianfang entfernt ist, bei Erfolg.

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

BS2000

aktueller Wert des Lese-/Schreibzeigers
d. h., die Anzahl der Bytes, die der Lese-/Schreibzeiger vom Dateianfang entfernt ist, bei Binärdateien, bei Erfolg.

absolute Position
: des Lese-/Schreibzeigers, bei Textdateien, bei Erfolg

-1
: bei Fehler. `errno` wird auf `ERANGE` gesetzt, wenn die Dateiposition nicht in 4 Bytes darstellbar ist.

Fehler `ftell()` und `ftello()` schlagen fehl, wenn gilt:

`EBADF`	Der *stream* zu Grunde liegende Dateideskriptor ist nicht zum Schreiben geöffnet, oder der Datenstrom-Puffer muss bereinigt werden, und die Datei ist nicht geöffnet.
`ESPIPE`	Der Dateideskriptor von *stream* ist mit einer Pipe oder FIFO verbunden.
`EOVERFLOW`	für *ftell()*: der resultierende Datei-Offset ist ein Wert, der in einem Objekt des Typs long nicht korrekt dargestellt werden kann.
`EOVERFLOW`	für *ftello()*: der aktuelle Datei-Offset kann in einem Objekt des Typs `off_t` nicht korrekt dargestellt werden.

Hinweis Ob `ftell()` / `ftello()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

BS2000
`ftell()` lässt sich auf Binärdateien (SAM im Binärmodus, PAM, INCORE) und Textdateien (SAM im Textmodus, ISAM) anwenden.

`ftell()` ist nicht anwendbar auf Systemdateien (SYSDTA, SYSLST, SYSOUT).

Siehe auch `fopen()`, `fseek()`, `lseek()`, `stdio.h`.

ftime - Datum und Uhrzeit ausgeben

Syntax

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

int ftime(struct timeb *tp);
```

Beschreibung

`ftime()` trägt in die Struktur, auf die *tp* zeigt, auf Millisekunden genau die Zeit ein, die seit dem 1. Januar 1970, 00:00:00 Uhr vergangen ist.

tp ist der Zeiger auf eine Struktur, die wie folgt in `sys/timeb.h` definiert ist:

```
struct timeb {
   time_t time;              /* Sekundenanteil      */
   unsigned short millitim;  /* Millisekundenanteil */
   short timezone;           /* nicht unterstützt   */
   short dstflag;            /* nicht unterstützt   */
};
```

Die Werte `timezone` und `dstflag` sind immer null. Sie können mit `ftime()` also nicht die lokale Zeitzone und die Einstellung für Sommerzeit ermitteln.

BS2000
`ftime()` liefert in einer Struktur dieselbe Zeit wie `time` (aktuelle Ortszeit als Anzahl der Sekunden, die seit dem 1. Januar 1950 00:00:00 vergangen sind) und zusätzlich die Millisekunden.

Aus Gründen der Portabilität sind weitere Möglichkeiten von `ftime()` in die Struktur aufgenommen. Sie werden jedoch in BS2000-Umgebung nicht versorgt. ❑

Returnwert

0	bei Erfolg.
-1	bei Fehler.

Hinweis

Portable Anwendungen sollten statt `ftime()` die Funktion `memcmp()` verwenden.

Bedingt durch die Auflösung der Systemuhr ist der Wert in `millitim` in der Regel nicht auf eine Millisekunde genau. Anwendungen, die von einer bestimmten Genauigkeit in `millitim` ausgehen, sind daher nicht portabel.

`ftime()` kann nicht zusammen mit der externen Variable `timezone` in einer Quelldatei verwendet werden.

Beim Übersetzen muss als `DEFINE` die Variable `_TIMEZONE_STRUCT` gesetzt werden.

BS2000

Der Speicherplatz muss für die Ergebnisstruktur explizit bereitgestellt werden.

Der Typ `time_t` ist in `sys/types.h` definiert.

Von den folgenden Strukturkomponenten werden in BS2000-Umgebung nur die Komponenten `time` und `millitim` versorgt. Die übrigen Komponenten sind aus Portabilitätsgründen in die Struktur aufgenommen:

`time`: Zeit in Sekunden seit dem 1. Januar 1950 00:00:00.

`millitim`: Angabe in Millisekunden (0 bis 999) zur Erhöhung der Genauigkeit von `time`.

`timezone`: lokale Zeitzone, gemessen in Minuten westlich von Greenwich (nicht unterstützt).

`dstflag`: Flag für Sommerzeit (nicht unterstützt). ❑

Siehe auch `ctime()`, `gettimeofday()`, `time()`, `sys/timeb.h`.

ftok - Interprozesskommunikation

Syntax #include <sys/ipc.h>

key_t ftok(const char *_path_, int _id_);

Beschreibung

`ftok()` gibt einen Schlüssel zurück, der auf *path* und *id* basiert und der in nachfolgenden Systemaufrufen `msgget()`, `semget()` und `shmget()` verwendet werden kann. *path* muss der Pfadname einer bestehenden Datei sein, auf die der Prozess zugreifen kann. *id* ist ein Zeichen, das ein Projekt eindeutig kennzeichnet.
`ftok()` gibt für alle Zeiger *path*, mit denen die gleiche Datei angesprochen wird, den gleichen Schlüssel zurück, wenn es mit der gleichen Kennung *id* aufgerufen wird.
`ftok()` gibt verschiedene Schlüssel zurück, wenn verschiedene Kennungen *id* angegeben oder wenn über *path* verschiedene Dateien angesprochen werden, die gleichzeitig im selben Dateisystem stehen. In der Regel liefert `ftok()` nicht denselben Schlüssel zurück, wenn es erneut mit denselben Argumenten *path* und *id* aufgerufen wird, die damit bezeichnete Datei aber zwischenzeitlich gelöscht und dann mit demselben Namen neu angelegt wurde.
Nur die 8 niederwertigen Bits von *id* werden verwendet. Wenn diese Bits null sind, ist das Verhalten von `ftok()` undefiniert.

Returnwert Schlüssel vom Typ key_t bei Erfolg.

`(key_t) -1` bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `ftok()` schlägt fehl, wenn gilt:

`EACCES` Sucherlaubnis existiert nicht für eine Komponente des Pfadpräfixes.

`ELOOP` Zu viele symbolische Verweise traten bei der Übersetzung von *path* auf.

`ENAMETOOLONG`
Der Pfadname, auf den *path* zeigt, ist länger als `{PATH_MAX}`, oder die Länge einer Komponente des Pfadnamens überschreitet `{NAME_MAX}`; oder die Auflösung symbolischer Verweise im Pfadnamen führt zu einem Zwischenergebnis, dessen Länge `{PATH_MAX}` überschreitet.

`ENOENT` Eine Komponente des Pfadnamens existiert nicht, oder *path* zeigt auf eine leere Zeichenkette.

`ENOTDIR` Eine Komponente des Pfadpräfixes von *path* ist kein Verzeichnis.

Hinweis Um maximale Portabilität zu erreichen, sollte die Kennung das niederwertigste Byte in *id* belegen. Die restlichen Bytes sollten auf 0 gesetzt sein.

Siehe auch `msgget()`, `semget()`, `shmget()`, `sys/ipc.h`.

ftruncate, truncate - Datei auf angegebene Länge setzen

Name **ftruncate, ftruncate64, truncate, truncate64**

Syntax #include <unistd.h>

int ftruncate (int *fildes*, off_t *length*);
int ftruncate64 (int *fildes*, off64_t *length*);
int truncate (const char **path*, off_t *length*);
int truncate64 (const char **path*, off64_t *length*);

Beschreibung

`ftruncate()` setzt die Länge einer normalen Datei mit dem Deskriptor *fildes* auf eine Länge von *length* Bytes fest.

`truncate()` unterscheidet sich von `ftruncate()` lediglich dadurch, dass die Datei über einen Zeiger *path*, der auf einen Pfadnamen verweist, angesprochen wird.

Die Auswirkung von `ftruncate()` und `truncate()` auf andere Typen von Dateien ist undefiniert. Wenn die Datei vorher länger als *length* Bytes war, kann auf die Bytes hinter der Position *length* nicht mehr zugegriffen werden. War die Datei vorher kürzer, so werden die Bytes zwischen der Dateiende-Marke vor dem Aufruf und der Dateiende-Marke nach dem Aufruf mit Nullen gefüllt. Bei `ftruncate()` muss die Datei zum Schreiben geöffnet sein; bei `truncate()` muss die effektive Benutzernummer des Prozesses das Schreibrecht für die Datei besitzen.

Wenn die Anforderung dazu führen würde, dass die Dateigröße den aktuellen für den Prozess definierten Grenzwert für die maximale Länge einer Datei überschreitet, wird die Funktion nicht ausgeführt, und das System schickt dem Prozess das Signal `SIGXFSZ`.

Diese Funktionen ändern nicht die aktuelle Position in der Datei. Bei erfolgreicher Ausführung, wenn die Dateigröße geändert wurde, aktualisieren diese Funktionen die Felder `st_ctime` und `st_mtime` der Datei. Die Bits `S_ISUID` und `S_ISGID` des Dateimodus werden evtl. gelöscht.

Es besteht kein funktionaler Unterschied zwischen `ftruncate()/ truncate()` und `ftruncate64()/ truncate64()`, außer dass bei `ftruncate64()`und `truncate64()`die Länge als Offset-Typ `off64_t` angegeben wird.

Returnwert

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

Fehler `ftruncate()` und `truncate()` schlagen fehl, wenn gilt:

`EINTR`	Während der Ausführung wurde ein Signal empfangen.
`EINVAL`	Der Wert von *length* ist negativ.

EFBIG oder EINVAL
: Der Wert von *length* ist größer als die maximal zulässige Dateigröße.

EIO
: Beim Lesen oder Schreiben des Dateisystems trat ein E/A-Fehler auf.

`ftruncate()` schlägt fehl, wenn gilt:

EBADF oder EINVAL
: *fildes* ist kein Dateideskriptor, der zum Schreiben geöffnet ist.

EINVAL
: *fildes* bezeichnet eine Datei, die nur zum Lesen geöffnet wurde.

`truncate()` schlägt fehl, wenn gilt:

EACCES
: Für eine Komponente des Pfadpräfixes existiert keine Sucherlaubnis, oder für die über *path* angesprochene Datei existiert keine Schreiberlaubnis.

EISDIR
: Die über *path* angesprochene Datei ist ein Verzeichnis.

ELOOP
: Beim Übersetzen von *path* traten zu viele symbolische Verweise auf.

ENAMETOOLONG
: Die Länge einer Komponente des Pfadnamens überschreitet `{NAME_MAX}` Zeichen, oder die Länge des Pfadnamens überschreitet `{PATH_MAX}` Zeichen.

ENOENT
: Entweder existiert eine Komponente des Pfadpräfixes nicht, oder *path* verweist auf eine leere Zeichenkette.

ENOTDIR
: Eine Komponente des Pfadpräfixes aus *path* ist kein Verzeichnis.

EROFS
: Die über *path* angesprochene Datei befindet sich in einem schreibgeschützten Dateisystem.

`truncate()` schlägt fehl, wenn gilt:

ENAMETOOLONG
: Die Auflösung symbolischer Verweise im Pfadnamen führt zu einem Zwischenergebnis, dessen Länge `{PATH_MAX}` überschreitet.

Siehe auch `open()`, `unistd.h`.

ftrylockfile - Sperren der Standardeingabe/-ausgabe

Syntax #include <stdio.h>

int ftrylockfile(FILE **file*);

Beschreibung

Siehe `flockfile()`.

ftw - Dateibaum durchwandern

Syntax

```
#include <ftw.h>

int ftw(const char *path, int (*fn) (const char *, const struct stat *ptr, int flag), int ndirs);
```

Beschreibung

`ftw()` wandert rekursiv durch die Dateiverzeichnishierarchie hinab, die bei *path* beginnt. Für jedes Objekt der Hierarchie ruft `ftw()` die Funktion auf, auf die *fn* zeigt, und übergibt ihr einen Zeiger auf eine mit dem Nullbyte abgeschlossene Zeichenkette, die den Namen des Objekts enthält, einen Zeiger auf eine Struktur vom Typ `struct stat` (siehe auch `sys/stat.h`), die Informationen über das Objekt enthält, und eine ganze Zahl. Mögliche Werte für die ganze Zahl sind die in der Include-Datei `ftw.h` definierten Werte:

`FTW_F`	für eine Datei
`FTW_D`	für ein Dateiverzeichnis
`FTW_DNR`	für ein Dateiverzeichnis, das nicht gelesen werden kann
`FTW_NS`	für ein Objekt, für das `stat()` nicht erfolgreich ausgeführt werden konnte

Wenn die ganze Zahl gleich `FTW_DNR` ist, dann werden Unterbäume dieses Dateiverzeichnisses nicht bearbeitet. Wenn die ganze Zahl gleich `FTW_NS` ist, dann enthält die Struktur *stat* undefinierte Werte. Ein Beispiel für ein Objekt, für das `FTW_NS` an die Funktion, auf die *fn* zeigt, übergeben werden würde, ist eine Datei in einem Dateiverzeichnis mit Lese- aber ohne Sucherlaubnis.

`ftw()` durchläuft zuerst ein Dateiverzeichnis, bevor einer seiner Nachfolger bearbeitet wird.

Die Baumdurchquerung dauert solange, bis der Baum vollständig durchsucht ist, ein Aufruf von *fn* einen Wert ungleich 0 zurückgibt oder in `ftw()` ein Fehler entdeckt wird.

ndirs gibt die maximale Anzahl von Dateiverzeichnisströmen und/oder Dateideskriptoren an, die für die Verwendung durch `ftw()` bei der Bearbeitung des Dateibaums zur Verfügung stehen. Wenn `ftw()` zurückkehrt, dann schließt sie alle Dateiverzeichnisströme und Dateideskriptoren, die sie verwendet hat, ohne dabei diejenigen zu berücksichtigen, die durch die Funktion *fn* des Benutzers geöffnet wurden.

Returnwert

0	bei Erfolg. Der Dateibaum ist abgearbeitet. `ftw()` liefert das Ergebnis der Funktion zurück, auf die *fn* zeigt.
-1	bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen. Wenn die Funktion, auf die *fn* zeigt, einen Wert ungleich 0 zurückliefert, dann beendet `ftw()` die Abarbeitung des Dateibaums und liefert das Ergebnis der Funktion zurück, auf die *fn* zeigt. Entdeckt `ftw()` einen Fehler, wird -1 zurückgegeben (s.oben).

Wenn die Funktion, auf die *fn* zeigt, einen Systemfehler erkennt, kann `errno` auf diesen gesetzt werden.

Fehler `ftw()` schlägt fehl, wenn gilt:

`EACCES` Das Durchsuchrecht für eine Komponente von *path* oder das Leserecht für *path* wird verweigert.

Erweiterung
`EBADF` Es wurde versucht, auf eine BS2000-Datei zuzugreifen. ❑

`ENAMETOOLONG`
Die Länge von *path* überschreitet `{PATH_MAX}` oder eine Komponente des Pfadnamens ist länger als `{NAME_MAX}`.

`ENOENT` *path* zeigt auf den Namen einer Datei, die nicht existiert, oder auf die leere Zeichenkette.

`ENOTDIR` Eine Komponente von *path* ist kein Dateiverzeichnis.

Hinweis Da `ftw()` rekursiv ist, kann diese Funktion mit einem Speicherfehler abbrechen, wenn sie auf sehr tiefe Dateibäume angewendet wird.

`ftw()` verwendet `malloc()`, um während ihres Ablaufs dynamisch Speicherplatz zu reservieren. Wenn `ftw()` zum Abbruch gezwungen wird, wie zum Beispiel durch `longjmp()` oder `siglongjmp()`, ausgeführt von der Funktion, auf die *fn* zeigt, oder aus einer Signalbehandlungsroutine heraus, dann hat `ftw()` keine Möglichkeit, diesen Speicher freizugeben, so dass dieser ständig reserviert bleibt. Ein sicherer Weg, Unterbrechungen zu behandeln, ist der, sich das Auftreten der Unterbrechung zu merken und die Funktion, auf die *fn* zeigt, zu veranlassen, bei ihrem nächsten Aufruf einen Wert ungleich 0 zurückzuliefern.

`ftw()` wird nur für POSIX-Dateien ausgeführt.

Siehe auch `longjmp()`, `malloc()`, `siglongjmp()`, `stat()`, `ftw.h`.

funlockfile - Sperren der Standardeingabe/-ausgabe

Syntax
```
#include <stdio.h>

void funlockfile(FILE *file);
```

Beschreibung

Siehe `flockfile()`.

fwide - Orientierung einer Datei festlegen

Syntax
```
#include <stdio.h>
#include <wchar.h>

int fwide(FILE *dz, int mode);
```

Beschreibung

`fwide()` legt die Orientierung der Datei mit dem Dateizeiger *dz* fest, sofern diese noch keine Orientierung hat. Ist die Orientierung bereits festgelegt – zum Beispiel durch eine vorherige Ein-/Ausgabe-Operation – verändert `fwide()` diese Orientierung nicht.

Abhängig vom Argument *mode* versucht `fwide()`, die Orientierung folgendermaßen einzustellen:

mode > 0	Datei wird Langzeichen-orientiert.
mode < 0	Datei wird Byte-orientiert.
mode = 0	die Orientierung der Datei wird nicht verändert.

Returnwert

> 0	wenn *dz* nach dem Aufruf von `fwide()` Langzeichen-orientiert ist.
< 0	wenn *dz* nach dem Aufruf von `fwide()` Byte-orientiert ist.
0	wenn *dz* keine Orientierung hat.

Hinweis

In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

fwprintf, swprintf, vfwprintf, vswprintf, vwprintf, wprintf - Langzeichen formatiert ausgeben

Syntax

```
#include <stdio.h>
#include <wchar.h>

int fwprintf(FILE *dz, const wchar_t *format [, arglist]);

#include <stdarg.h>
#include <wchar.h>

int vwprintf(const wchar_t *format, va_list arg);

#include <wchar.h>

int wprintf(const wchar_t *format [, arglist]);
int swprintf(wchar_t *s, size_t n, const wchar_t *format [, arglist]);

#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>

int vfwprintf(FILE *dz, const wchar_t *format, va_list arg);
int vswprintf(wchar_t *s, size_t n, const wchar_t *format, va_list arg);
```

Beschreibung

Die Funktionen dienen der formatierten Ausgabe.

`fwprintf()` bereitet die Argumente in der Liste *arglist* gemäß den Angaben in der Langzeichenkette *format* auf und schreibt sie in die Datei mit dem Dateizeiger *dz*. `fwprintf()` kehrt zurück, wenn das Ende von *format* erreicht wird.

`vwprintf()` entspricht der Funktion `fwprintf()` mit *dz* = `stdout`, wobei die Argumentliste durch ein Argument vom Typ `va_list` ersetzt wird, das durch das Makro `va_start` initialisiert worden sein muss (möglicherweise gefolgt von `va_arg`-Aufrufen). Die Funktion ruft nicht das Makro `va_end` auf.

`wprintf()` entspricht der Funktion `fwprintf()` mit *dz* = `stdout`.

`swprintf()` schreibt Ausgaben formatiert in die Langzeichenkette *s*. `swprintf()` entspricht ansonsten der Funktion `fwprintf()`. Es werden maximal *n* Langzeichen geschrieben, inklusive des abschließenden Nullzeichens, das für *n* > 0 automatisch angefügt wird.

`vfwprintf()` entspricht der Funktion `fwprintf()`, wobei die Liste durch ein Argument vom Typ `va_list` ersetzt wird, das durch das Makro `va_start` initialisiert worden sein muss (möglicherweise gefolgt von `va_arg`-Aufrufen). Die Funktion ruft nicht das Makro `va_end` auf.

`vswprintf()` entspricht der Funktion `swprintf()`, wobei die Liste durch ein Argument vom Typ `va_list` ersetzt wird, das durch das Makro `va_start` initialisiert worden sein muss (möglicherweise gefolgt von `va_arg`-Aufrufen). Die Funktion ruft nicht das Makro `va_end` auf.

Der Parameter *format* ist eine Langzeichenkette, die keine, eine oder mehrere Umwandlungsanweisungen und Langzeichen enthält:

– Umwandlungsanweisungen beginnen mit dem Prozentzeichen (`%`). Jede Umwandlungsanweisung wird keinem, einem oder mehreren Argumenten in *arglist* zugeordnet. Wenn in *arglist* weniger Argumente übergeben werden, als in *format* festgelegt sind, ist das Ergebnis undefiniert. Wenn in *format* weniger Argumente festgelegt sind, als in *arglist* übergeben werden, werden die überflüssigen Argumente ignoriert.
Die einer Umwandlungsanweisung zugeordneten Argumente werden gemäß der Anweisung konvertiert und formatiert in den Ausgabedatenstrom geschrieben.

– Zeichen vom Typ `wchar_t` (aber nicht %), die 1 : 1 in die Ausgabe kopiert werden.

– Zwischenraumzeichen (siehe „Zwischenraumzeichen“ auf Seite 85)

Umwandlungsanweisungen

Jede Umwandlungsanweisung wird mit dem Zeichen `%` eingeleitet; darauf folgen:

– Keines oder mehrere **Formatierungszeichen**, die die Bedeutung der Umwandlungsanweisung verändern.

– Eine optionale Ganzzahl (bestehend aus Dezimalziffern) oder ein Asterisk (*), die eine minimale **Feldbreite** für die Ausgabe eines Arguments angibt. Wenn der umgewandelte Wert aus weniger Zeichen als der Feldbreite besteht, wird links bis zur Feldbreite aufgefüllt (bzw. rechts, wenn das Formatierungszeichen "−" für linksbündige Ausrichtung angegeben wurde).

– Eine optionale **Genauigkeit**, die angibt, wie viele Ziffern mindestens für die Umwandlungen `d`, `i`, `o`, `u`, `x` oder `X` erscheinen sollen, wie viele Ziffern nach dem Dezimalzeichen für die Umwandlungen `e`, `E` und `f` erscheinen sollen, wie viele signifikante Stellen bei den Umwandlungen `g` und `G` vorhanden sind oder wie viele Zeichen maximal aus der Zeichenkette für die Umwandlung *s* ausgegeben werden sollen. Die Genauigkeit hat die Form ".", gefolgt von einer Ganzzahl aus dezimalen Ziffern oder einem Asterisk (*). Ist nur der Punkt angegeben, wird 0 als Genauigkeit eingesetzt.

– Ein optionales `h`, `l` oder `L` vor einem Umwandlungszeichen:
`l` vor `c` bedeutet, dass ein Argument vom Typ `wint_t` umgewandelt werden soll;
`l` vor `s`: bedeutet, dass ein Argument vom Typ `wchar_t` (Zeiger auf eine Langzeichenkette) umgewandelt werden soll;
`h` vor `d`, `i`, `o`, `u`, `x` oder `X`: Umwandlung eines Arguments vom Typ `short int` oder `unsigned short int` (das Argument ist entsprechend der ganzzahligen Erweiterung

erweitert worden, und sein Wert wird vor der Ausgabe in ein `short int` oder `unsigned short int` umgewandelt);
`h` vor `n`: Umwandlung eines Arguments vom Typ Zeiger auf `short int`;

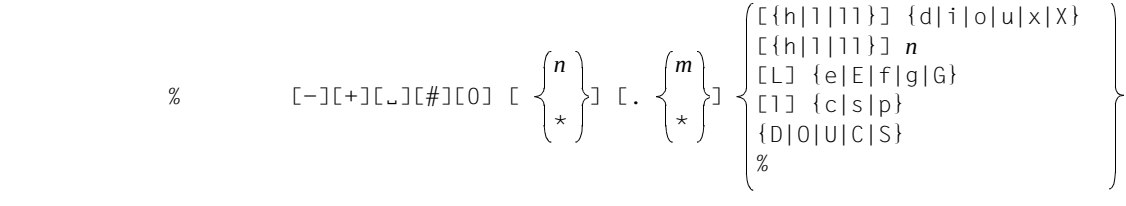
`l` vor `d`, `i`, `o`, `u`, `x` oder `X`: Umwandlung eines Arguments vom Typ `long int` oder `unsigned long int`;
`l` vor `n`: Umwandlung eines Arguments vom Typ Zeiger auf `long int`;
`ll` vor `d`, `i`, `o`, `u`, `x` oder `X`: Umwandlung eines Arguments vom Typ `long long int` oder `unsigned long long int`;
`ll` vor `n`: Umwandlung eines Arguments vom Typ Zeiger auf `long long int`;
`L` vor `e`, `E`, `f`, `g` oder `G`: Umwandlung eines Arguments vom Typ `long double`.

Wenn `h` , `l` oder `L` vor einem anderen Umwandlungszeichen steht, ist das Verhalten undefiniert.

– Ein **Umwandlungszeichen** vom Typ `wchar_t`, das den Typ der durchzuführenden Umwandlung angibt, siehe Auflistung unten.

Feldbreite, Genauigkeit oder beides können durch das Zeichen * (Asterisk) angegeben werden. In diesem Fall werden die Werte statt aus der Formatangabe aus der Argumentliste entnommen: Die (ganzzahligen) Werte für Feldbreite und/oder Genauigkeit müssen unmittelbar vor dem Argument stehen, das umgewandelt werden soll.
Ist eine negative Feldbreite angegeben, wird "–" als Formatierungszeichen interpretiert, dem eine positive Feldbreite folgt. Eine negative Genauigkeit wird interpretiert, als ob die Genauigkeit weggelassen wird.

Umwandlungsanweisungen sehen also wie folgt aus:



1. Anfang einer Umwandlungsanweisung
2. Formatierungszeichen
3. Feldbreite
4. Genauigkeit
5. Zeichen, die die eigentliche Umwandlung festlegen

Formatierungszeichen

−	Das Ergebnis der Umwandlung wird linksbündig innerhalb des Felds ausgerichtet.
+	Das Ergebnis einer Umwandlung mit Vorzeichen wird immer mit einem Vorzeichen ausgegeben (+ oder −).
␣	Wenn das erste Langzeichen einer vorzeichenbehafteten Umwandlung kein Vorzeichen ist oder das Ergebnis einer vorzeichenbehafteten Umwandlung keine Langzeichen ergibt, wird dem Resultat ein Leerzeichen vorangestellt. Wird sowohl das Leerzeichen als auch das Zeichen + angegeben, wird das Formatierungszeichen Leerzeichen ignoriert.
#	Dieses Formatierungszeichen gibt an, dass der umzuwandelnde Wert in einer "alternativen Form" darzustellen ist. Für die Umwandlung o wird die Genauigkeit so weit erhöht, dass die erste Ziffer des Ergebnisses die Ziffer 0 ist. Für x (oder X) wird einem Resultat ungleich 0 die Zeichenfolge "0x" (oder "0X") vorangestellt. Für e, E, f, g oder G enthält das Ergebnis immer ein Dezimalpunkt-Langzeichen, auch wenn keine weiteren Ziffern folgen (normalerweise erscheint ein Dezimalpunkt-Langzeichen nur dann im Ergebnis, wenn ihm eine Ziffer folgt). Für g und G werden abschließende Nullen nicht aus dem Ergebnis entfernt, wie sonst üblich. Das Verhalten bei anderen Umwandlungszeichen ist undefiniert.
0	Für d, i, o, u, x, X, e, E, f, g und G werden zum Auffüllen bis zur Feldbreite führende Nullen verwendet (nach Anzeige eines Vorzeichens oder einer Basis); es wird nicht mit Leerzeichen aufgefüllt. Wenn sowohl das Formatierungszeichen 0 als auch − angegeben werden, wird das Formatierungszeichen 0 ignoriert. Ist eine Genauigkeit angegeben, wird für d, i, o, u, x und X das Formatierungszeichen 0 ignoriert. Für andere Umwandlungen ist das Verhalten undefiniert.

Umwandlungszeichen

d, i	Das int-Argument wird in eine vorzeichenbehaftete Dezimalzahl der Form [−]*dddd* umgewandelt. Die Genauigkeit legt die minimale Anzahl von Ziffern fest, die ausgegeben werden sollen. Wenn der umzuwandelnde Wert weniger Ziffern ergibt, wird er um führende Nullen erweitert. Die voreingestellte Genauigkeit ist 1. Die Umwandlung des Werts 0 mit einer ausdrücklich genannten Genauigkeit von 0 liefert kein Langzeichen.

`o, u` Das `unsigned int`-Argument wird in eine vorzeichenlose Oktalzahl (o) oder in eine vorzeichenlose Dezimalzahl (u) der Form *dddd* umgewandelt. Die Genauigkeit legt die minimale Anzahl von Ziffern fest, die erscheinen sollen; wenn der umzuwandelnde Wert weniger Ziffern ergibt, wird er um führende Nullen erweitert. Die voreingestellte Genauigkeit ist 1.
Die Umwandlung des Werts 0 mit einer ausdrücklich genannten Genauigkeit von 0 liefert kein Langzeichen.

`x, X` Das `unsigned int`-Argument wird in ein eine vorzeichenlose Hexadezimalzahl der Form *dddd* umgewandelt; außer den Zahlen werden die Buchstaben `abcdef` (bei `x`) bzw. ABCDEF (bei `X`) als numerische Zeichen verwendet. Die Genauigkeit legt die minimale Anzahl von Ziffern fest, die erscheinen sollen; wenn der umzuwandelnde Wert weniger Ziffern ergibt, wird er um führende Nullen erweitert. Die voreingestellte Genauigkeit ist 1.
Die Umwandlung des Werts 0 mit einer ausdrücklich genannten Genauigkeit von 0 liefert kein Langzeichen.

`f` Das `double`-Argument wird in die dezimale Schreibweise der Form `[-]`*ddd*`.`*ddd* umgewandelt, wobei die Anzahl der Ziffern nach dem Dezimalzeichen gleich der angegebenen Genauigkeit ist.
Ist keine Genauigkeit angegeben, wird die Genauigkeit 6 eingesetzt.
Wenn die Genauigkeit gleich 0 ist und kein #-Formatierungszeichen gesetzt ist, wird kein Dezimalzeichen ausgegeben.
Wenn das Dezimalzeichen erscheint, wird davor mindestens eine Ziffer ausgegeben. Der Wert wird auf die entsprechende Zahl von Ziffern gerundet.

`e, E` Das `double`-Argument wird in die Form `[-]`*d*`.`*ddde*±*dd* umgewandelt, wobei genau eine Ziffer vor dem Dezimalzeichen ausgegeben wird (diese Ziffer ist ungleich 0, wenn das Argument ungleich 0 ist). Die Anzahl der Nachkommastellen ist gleich der Genauigkeit. Ist keine Genauigkeit angegeben, wird die Genauigkeit 6 eingesetzt.
Wenn die Genauigkeit gleich 0 und kein #-Formatierungszeichen gesetzt ist, wird kein Dezimalzeichen ausgegeben. Der Wert wird auf die entsprechende Zahl von Ziffern gerundet.
Das Umwandlungszeichen `E` erzeugt eine Zahl mit `E` an Stelle von `e` für die Anzeige des Exponenten. Der Exponent enthält immer mindestens zwei Ziffern. Wenn der Wert gleich 0 ist, ist der Exponent gleich 0.

`g, G` Das `double`-Argument wird in die Form von `f` oder `e` umgewandelt (bzw. in die Form `E` für das Umwandlungszeichen `G`). Die Genauigkeit gibt die Anzahl der signifikanten Stellen an. Die Angabe einer Genauigkeit 0 wird durch Genauigkeit 1 ersetzt.
Die Form hängt vom umgewandelten Wert ab; die Form `e` wird nur dann verwendet, wenn der Exponent einer solchen Umwandlung kleiner als -4 oder

größer gleich der Genauigkeit ist. Abschließende Nullen werden vom gebrochenen Teil des Ergebnisses entfernt; ein Dezimalzeichen erscheint nur dann, wenn es von einer Ziffer gefolgt wird.

`c` Ist das Zeichen `l` vorangestellt, wird das Argument vom Typ `wint_t` in den Typ `wchar_t` umgewandelt, das resultierende Zeichen wird geschrieben. Ist kein `l` vorangestellt, wird das Argument vom Typ `int` wie beim Aufruf der Funktion `btowc()` in ein Langzeichen umgewandelt; das resultierende Zeichen wird geschrieben.

`s` Ist kein Zeichen `l` vorangestellt, soll das Argument vom Typ Zeiger auf ein `char`-Feld sein. Zeichen aus dem Feld werden so konvertiert wie bei Aufrufen der Funktion `mbrtowc()`. Der Konversions-Status wird in einem Objekt vom Typ `mbstate_t` beschrieben und mit 0 initialisiert, bevor das erste Multibyte-Zeichen konvertiert wird. Es wird bis zum abschließenden Nullzeichen geschrieben (ausschließlich).
Ist das Zeichen `l` vorangestellt, soll das Argument vom Typ Zeiger auf ein `wchar_t`-Feld sein. Langzeichen aus dem Feld werden bis zum abschließenden Nullzeichen geschrieben (ausschließlich).

Wenn eine Genauigkeit m angegeben ist, werden nicht mehr als m Langzeichen geschrieben. Wird die Genauigkeit nicht angegeben oder ist diese größer als die Länge des konvertierten Feldes, sollte das Feld das Langzeichen 0 enthalten (als Endekriterium).

`S` entspricht `ls`.

`C` entspricht `lc`.

`p` Das Argument muss ein Zeiger auf `void` sein. Die Ausgabe erfolgt als 8-stellige Sedezimalzahl.

`n` Das Argument muss ein Zeiger auf `int` sein, in welches die Anzahl der bisher von `fwprintf` beim aktuellen Aufruf geschriebenen Bytes eingetragen wird. Es wird kein Argument umgewandelt.

`%` Es wird das Langzeichen `%` ausgegeben; es wird kein Argument umgewandelt. Die vollständige Umwandlungsanweisung muss die Form %% haben.

Wenn das Zeichen nach `%` kein gültiges Umwandlungszeichen ist, ist das Ergebnis der Umwandlung undefiniert.

Falls ein Argument eine `UNION` oder ein Zeiger auf eine `UNION` ist, ist das Ergebnis der Umwandlung undefiniert.
das Gleiche gilt, wenn ein Argument ein Feld oder ein Zeiger auf ein Feld ist, ausgenommen die drei folgenden Fälle:
das Argument ist ein Feld vom Typ `char` und verwendet `%s`,
das Argument ist ein Feld vom Typ `wchar_t` und verwendet `%ls` oder
das Argument ist ein Zeiger und verwendet `%p`.

In keinem Fall verursacht eine nicht existierende oder zu kleine Feldbreite das Abschneiden eines Feldes; wenn das Ergebnis einer Umwandlung breiter als die Feldbreite ist, wird das Feld einfach erweitert, um die Ausgabe aufzunehmen.

Returnwert Anzahl der ausgegebenen Langzeichen
bei erfolgreicher Beendigung.

negativer Wert bei Fehler.

Hinweis In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Siehe auch `btowc()`, `fprintf()`, `mbrtowc()`, `printf()`

fwscanf, swscanf, wscanf - formatiert lesen

Syntax

```
#include <stdio.h>
#include <wchar.h>

int fwscanf(FILE *dz, const wchar_t *format [, arglist]);


#include <wchar.h>

int swscanf(const wchar_t *s, const wchar_t *format [, arglist]);
int wscanf(const wchar_t *format [, arglist]);
```

Beschreibung

Die Funktionen dienen der formatierten Eingabe.
Sie lesen Eingaben, wandeln sie gemäß den Angaben in der Formatzeichenkette *format* um und speichern die Ergebnisse in den Bereichen ab, die in der optionalen Argumentliste *arglist* angegeben wurden.

`fwscanf()` liest Eingaben formatiert aus der Datei mit dem Dateizeiger *dz*.

`swscanf()` liest Eingaben formatiert aus der Langzeichenkette *s*. `swscanf()` entspricht ansonsten der Funktion `fwscanf()`. Das Ende der Langzeichenkette entspricht EOF.

`wscanf()` liest Eingaben formatiert aus der Standardeingabe `stdin`. `wscanf()` entspricht der Funktion `fwscanf()` mit *dz* = `stdin`.

Der Parameter *format* ist eine Zeichenkette, die in ihrem anfänglichen Umschaltmodus beginnt und endet, (sofern ein Umschaltmodus definiert ist) und keine, eine oder mehrere Umwandlungsanweisungen enthält. *format* kann drei Arten von Zeichen enthalten:

– Zeichen vom Typ `wchar_t` (aber kein Zwischenraumzeichen oder %), die 1 : 1 in den Ausgabedatenstrom kopiert werden.

– Zwischenraumzeichen, beginnend mit einem Gegenschrägstrich (`\`) (siehe `iswspace()`).

– Umwandlungsanweisungen, beginnend mit dem Prozentzeichen (%), von denen jede keinem, einem oder mehreren Argumenten in *arglist* zugeordnet wird. Wenn in *arglist* weniger Argumente übergeben werden, als in *format* festgelegt sind, ist das Ergebnis undefiniert. Wenn in *format* weniger Argumente festgelegt sind, als in *arglist* übergeben werden, werden die überflüssigen Argumente ignoriert.

Die `wscanf()`-Funktionen lesen das Eingabezeichen zunächst ohne es umzuwandeln und in einer Variablen abzuspeichern. Stimmt das Eingabezeichen nicht mit dem in *format* angegebenen Zeichen überein, wird die Eingabebearbeitung abgebrochen und die Funktion kehrt zurück. Wenn die Umwandlung wegen eines nicht passenden Langzeichens abbricht, verbleibt dieses Zeichen ungelesen im Eingabestrom.

Zwischenraumzeichen

Die Formatzeichenkette *format* kann beliebig viele oder keine Zwischenraumzeichen enthalten. Diese Zeichen haben keine Steuerfunktion.

Zwischenraumzeichen in der Eingabe werden als Trennzeichen zwischen Eingabefeldern behandelt und nicht mit umgewandelt (Ausnahme siehe `%c`, `%n` und `%[]`). Führende Zwischenraumzeichen werden bei der Eingabe ignoriert.

Umwandlungsanweisungen

Alle Formen von `fwscanf()` erlauben das Erkennen eines landessprach-spezifischen Dezimalzeichens in der Eingabezeichenkette. Das Dezimalzeichen wird durch die Lokalität des Progamms definiert (Kategorie `LC_NUMERIC`). In der Lokalität `POSIX` oder einer Lokalität, bei der das Dezimalzeichen nicht definiert ist, ist das Dezimalzeichen auf . (Punkt) voreingestellt.

Jede Umwandlungsanweisung muss mit einem Prozentzeichen (`%`) beginnen; darauf folgen:

– Ein optionales Langzeichen Stern (*) zum Überspringen einer Zuweisung.

– Eine optionale Ganzzahl (Dezimalziffern) ungleich 0, welche die maximale **Feldbreite** angibt.

– Ein optionales `h` ,`l` oder `L`, das die Größe des aufnehmenden Objekts angibt:
`l` vor den Umwandlungszeichen `c`, `s` und `[` : das entsprechende Argument ist ein Zeiger auf `wchar_t` .
`h` bzw. `l` vor `d`, `i` und `n` : das entsprechende Argument ist ein Zeiger auf `short int` (`h`) bzw. `long int` (`l`).
`h` bzw. `l` vor `o`, `u` und `x` : das entsprechende Argument ist ein Zeiger auf `unsigned short int` (`h`) bzw. `unsigned long int` (`l`).
`ll` vor `d`, `i` und `n` : das entsprechende Argument ist ein Zeiger auf `long long int`.
`ll` vor `o`, `u` und `x` : das entsprechende Argument ist ein Zeiger auf `unsigned long long int`.
`l` bzw. `L` vor `e`, `f` und `g` : das entsprechende Argument ist ein Zeiger auf `double` (`l`) bzw.`long double (L)`.

  Wenn `h` ,`l` oder `L` vor einem anderen Umwandlungszeichen steht, ist das Verhalten undefiniert.

– Ein **Umwandlungszeichen**, das den Typ der durchzuführenden Umwandlung angibt.

`fwscanf()` führt jede Anweisung einzeln aus. Wenn eine Anweisung fehlschlägt, wie unten genauer erläutert, kehrt die Funktion zurück. Fehler werden als Eingabefehler bezeichnet, wenn Eingabezeichen fehlen, oder als Formatfehler, wenn Eingabezeichen nicht zu dem Format passen.

Eine Anweisung, die aus einem Zwischenraumzeichen besteht, wird so ausgeführt, dass die Eingabe bis zum ersten Langzeichen gelesen wird, das kein Zwischenraumzeichen ist (dieses Langzeichen selbst wird nicht gelesen) oder bis keine Langzeichen mehr gelesen werden können (EOF).

Eine Anweisung, die aus einem normalen Langzeichen besteht, wird ausgeführt, indem das nächste Langzeichen aus der Eingabe gelesen wird. Wenn dieses Langzeichen nicht mit dem vorgegebenen Langzeichen übereinstimmt, schlägt die Anweisung fehl und das unpassende und alle nachfolgenden Langzeichen werden nicht gelesen.

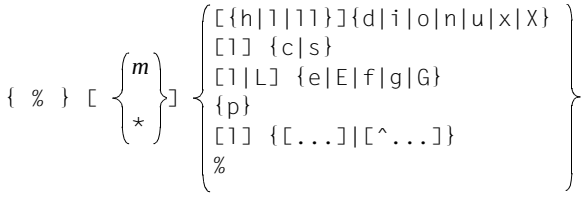
Eine Anweisung, die eine Umwandlungsanweisung ist, definiert eine Menge von passenden Eingabefolgen, wie dies unten für jede einzelne Umwandlungsanweisung beschrieben wird. Eine Umwandlungsanweisung wird in den folgenden Schritten ausgeführt:

Die Eingabe von Zwischenraumzeichen wird überlesen, solange die Anweisung weder ein [noch eines der Umwandlungszeichen c oder *n* enthält.

Eingabeelemente werden aus der Eingabe gelesen, solange die Anweisung nicht das Umwandlungszeichen *n* enthält. Ein Eingabeelement ist definiert als die längste Folge von Eingabezeichen (bis zu einer eventuell angegebenen maximalen Feldbreite), die ein Anfang einer passenden Folge ist. Das erste Langzeichen nach einem Eingabeelement bleibt, sofern es vorhanden ist, ungelesen.
Wenn die Länge des Eingabeelements gleich 0 ist, schlägt die Ausführung der Anweisung fehl; diese Bedingung bedeutet einen Formatfehler, sofern nicht ein Eingabefehler wie zum Beispiel EOF oder das Auftreten eines Lese-Fehlers weitere Eingaben verhindert.

Sofern nicht das Umwandlungszeichens % angegeben ist, wird das Eingabeelement (bzw. bei %n die Anzahl der gelesenen Eingabezeichen), umgewandelt in einen Datentyp, der dem Umwandlungszeichen entspricht. Wenn das Eingabeelement nicht zu der Umwandlungsanweisung passt, schlägt die Ausführung dieser Anweisung mit einem Formatfehler fehl.
Passt das Eingabeelement, wird - sofern die Zuweisung nicht durch das Zeichen * unterdrückt wird - das Ergebnis der Umwandlung in dem Objekt abgelegt, welches das erste auf *format* folgende Argument ist, in dem bisher noch kein Umwandlungsergebnis abgelegt wurde. Wenn dieses Objekt nicht den passenden Datentyp hat oder wenn das Ergebnis der Umwandlung nicht in dem zur Verfügung stehenden Platz dargestellt werden kann, ist das Verhalten undefiniert.

Umwandlungsanweisungen sehen also wie folgt aus:



Umwandlungszeichen

`d` Liest eine optional mit einem Vorzeichen versehene dezimale Ganzzahl ein, deren Format dasselbe ist, das die Funktion `wcstol()` erwartet (*base* = 10). Das zugehörige Argument sollte ein Zeiger auf `int` sein.

`i` Liest eine optional mit einem Vorzeichen versehene dezimale Ganzzahl ein, deren Format dasselbe ist, das die Funktion `wcstol()` erwartet (*base* = 8). Das entsprechende Argument sollte vom Typ Zeiger auf `int` sein.

`o` Liest eine optional mit einem Vorzeichen versehene oktale Ganzzahl ein, deren Format dasselbe ist, das die Funktion `wcstoul()` erwartet (*base* = 8). Das entsprechende Argument sollte vom Typ Zeiger auf `unsigned integer` sein.

`u` Liest eine optional mit einem Vorzeichen versehene dezimale Ganzzahl ein, deren Format dasselbe ist, das die Funktion `wcstoul()` erwartet (*base* = 10). Das entsprechende Argument sollte vom Typ Zeiger auf `unsigned integer` sein.

`x, X` Liest eine optional mit einem Vorzeichen versehene hexadezimale Ganzzahl ein, deren Format dasselbe ist, das die Funktion `wcstoul()` erwartet (*base* = 16). Das entsprechende Argument sollte vom Typ Zeiger auf `unsigned integer` sein.

`e, E, f, g, G` Diese Umwandlungszeichen lesen eine optional mit einem Vorzeichen versehene Gleitpunktzahl ein. Deren Format ist dasselbe, das auch `wcstod()` erwartet. Das entsprechende Argument sollte vom Typ Zeiger auf `float` sein.

`s` Liest eine Folge von Langzeichen ein, die keine Zwischenraumzeichen sind.
Ist kein `l` angegeben, werden Zeichen aus dem Eingabefeld so konvertiert wie bei Aufrufen der Funktion `wcrtomb()`. Der Konversions-Status wird dabei in einem Objekt vom Typ `mbstate_t` beschrieben und mit 0 initialisiert,

bevor das erste Langzeichen konvertiert wird. Es wird bis zum abschließenden Nullzeichen geschrieben.Das entsprechende Argument sollte ein Zeiger ein `char`-Feld sein, das groß genug ist, um die konvertierte Folge und ein abschließendes Nullzeichen aufzunehmen, das automatisch angefügt wird.
Ist `l` angegeben, sollte das entsprechende Argument ein Zeiger auf das erste Element eines `wchar_t`-Feldes sein, das groß genug ist, um die Folge und ein abschließendes Nullzeichen aufzunehmen, das automatisch angefügt wird.

`[` Liest eine nichtleere Folge von Langzeichen aus einer Menge von erwarteten Langzeichen (der Eingabemenge).
Ist kein `l` angegeben, werden Zeichen aus dem Eingabefeld so konvertiert wie bei Aufrufen der Funktion `wcrtomb()`. Der Konversions-Status wird dabei in einem Objekt vom Typ `mbstate_t` beschrieben und mit 0 initialisiert, bevor das erste Langzeichen konvertiert wird. Es wird bis zum abschließenden Nullzeichen geschrieben.Das entsprechende Argument sollte ein Zeiger ein `char`-Feld sein, das groß genug ist, um die konvertierte Folge und ein abschließendes Nullzeichen aufzunehmen, das automatisch angefügt wird.
Ist `l` angegeben, sollte das entsprechende Argument ein Zeiger auf das erste Element eines `wchar_t`-Feldes sein, das groß genug ist, um die Folge und ein abschließendes Nullzeichen aufzunehmen, das automatisch angefügt wird.
Die Umwandlungsanweisung umfasst alle auf `[` folgenden Langzeichen in der Zeichenkette *format* bis einschließlich der zugehörigen schließenden eckigen Klammer `]`. Die Langzeichen zwischen den Klammern stellen die Eingabemenge dar, sofern nicht das erste Langzeichen nach der linken Klammer das Zeichen `^` ist. In diesem Fall enthält die Eingabemenge alle Langzeichen, die nicht in der Liste zwischen dem Zeichen `^` und der Klammer `]` aufgeführt sind.
Als Sonderfall gilt, dass die rechte eckige Klammer in den beiden Fällen, in denen die Umwandlungsanweisung mit den Zeichenketten `[]` bzw. `[^]` beginnt, zur Eingabemenge gehört und erst die nächste rechte eckige Klammer diejenige ist, welche die Umwandlungsanweisung abschließt.
Wenn das Zeichen – in der Liste auftritt und weder das letzte Zeichen noch das erste Zeichen nach `[` bzw. `[^` ist, dann ist das Verhalten undefiniert.

`c` Liest eine Folge von Langzeichen, deren Anzahl durch die Feldbreite bestimmt wird. Ist keine Feldbreite angegeben, wird 1 Langzeichen gelesen.
Ist kein `l` angegeben, werden Zeichen aus dem Eingabefeld so konvertiert wie bei Aufrufen der Funktion `wcrtomb()`. Der Konversions-Status wird dabei in einem Objekt vom Typ `mbstate_t` beschrieben und mit 0 initialisiert, bevor das erste Langzeichen konvertiert wird. Das entsprechende Argument sollte ein Zeiger auf ein `char`-Feld sein, das groß genug ist, um die

konvertierte Folge aufzunehmen. Es wird kein Nullzeichen angefügt.
Ist `l` angegeben, sollte das entsprechende Argument ein Zeiger auf das erste Element eines `wchar_t`-Feldes sein, das groß genug ist, um die Folge aufzunehmen Es wird kein Nullzeichen angefügt.

Das Überlesen von Zwischenraumzeichen wird in diesem Fall unterdrückt; um das nächste Langzeichen zu lesen, das kein Zwischenraumzeichen ist, sollte `%ls` verwendet werden.

`p` Liest eine Menge von Folgen, die denen entsprechen sollten, die von der Umwandlungsanweisung `%p` der `fwprintf()`-Funktionen erzeugt werden. Das entsprechende Argument sollte ein Zeiger auf einen Zeiger auf `void` sein. Die Interpretation des Eingabeelements ist jeweils implementierungsabhängig; für ein Eingabeelement, das nicht zuvor während derselben Programmausführung umgewandelt wurde, ist das Verhalten der Umwandlungsanweisung `%p` undefiniert. Dies gilt insbesondere für Zeigerausgaben, die von anderen Systemen erzeugt worden sind.

`n` Es wird keine Eingabe verarbeitet. Das entsprechende Argument sollte ein Zeiger auf `int` sein, in das die bisher von diesem Aufruf gelesene Zahl der Langzeichen eingetragen wird. Die Ausführung einer Anweisung des Typs *%n* erhöht nicht den Zuweisungszähler, der bei Beendigung der Ausführung der Funktion zurückgeliefert wird.

`%` Liest ein einzelnes `%`. Dabei findet keine Umwandlung oder Zuweisung statt. Die vollständige Umwandlungsanweisung lautet `%%`.

Wenn ein Umwandlungszeichen ungültig ist, ist das Verhalten von `fwscanf()` undefiniert.

Wenn das Dateiende während der Eingabe gefunden wird, wird die Umwandlung abgebrochen. Wenn das Dateiende auftritt, bevor irgendwelche, zur aktuellen Anweisung passenden Langzeichen gelesen wurden (abgesehen von zulässigen Zwischenraumzeichen), wird die Ausführung der aktuellen Anweisung mit einem Eingabefehler abgebrochen. Andernfalls wird, falls die Bearbeitung der aktuellen Anweisung nicht mit einem Formatfehler abbricht, eine von `%n` verschiedene Darauf folgende Anweisung mit einem Eingabefehler abgebrochen.

Wenn während eines `swscanf()`-Aufrufs das Ende einer Zeichenkette erreicht wird, ist dies äquivalent zum Erreichen des Dateiendekennzeichens während eines `fwscanf()`-Aufrufs.

Abschließende Zwischenraumzeichen (einschließlich der Zeilenendezeichen) bleiben ungelesen, sofern nicht eine entsprechende Umwandlungsanweisung vorhanden ist.

Der Erfolg des 1:1 Einlesens von Buchstaben und von unterdrückten Zuweisungen kann nicht direkt bestimmt werden, außer über die Anweisung *%n*.

Returnwert Anzahl der eingelesenen und erfolgreich zugewiesenen Eingabeelemente
falls nicht vor der ersten Zuweisung ein Eingabefehler auftritt.
Die Anzahl ist null, wenn bereits beim ersten Eingabelement ein Formatfehler auftritt.

`EOF` falls vor der ersten Zuweisung ein Eingabefehler auftritt.

Hinweis In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Siehe auch `scanf()`, `sscanf()`, `fscanf()`, `wcstod()`, `wcstol()`, `wcstoul()`, `wcrtomb()`

fwrite - Daten binär ausgeben

Syntax #include <stdio.h>

size_t fwrite(const void *_ptr_, size_t _size_, size_t _nitems_, FILE *_stream_);

Beschreibung

`fwrite()` schreibt *nitems* Elemente der Größe *size* aus dem Vektor, auf den *ptr* zeigt, in den Datenstrom, auf den *stream* zeigt. Der Lese-/Schreibzeiger des Datenstroms wird, wenn er definiert ist, um die Anzahl von Bytes erhöht, die erfolgreich geschrieben wurden. Wenn ein Fehler auftritt, ist der Wert des Lese-/Schreibzeigers unbestimmt.

Die Strukturkomponenten `st_ctime` und `st_mtime` der Datei werden zwischen der erfolgreichen Ausführung von `fwrite()` und der nächsten erfolgreichen Beendigung eines Aufrufs von `fflush()` oder `fclose()` für denselben Datenstrom oder einem Aufruf von `exit()` oder `abort()` für die Änderung markiert (siehe `sys/stat.h`).

BS2000

Satz-Ein-/Ausgabe

– `fwrite()` schreibt einen Satz in die Datei.

– Bei sequenziellen Dateien (SAM, PAM) wird der Satz an die aktuelle Dateiposition geschrieben.

– Bei indexsequenziellen Dateien (ISAM) wird der Satz an die Position geschrieben, die dem Schlüsselwert im Satz entspricht.

– Anzahl der auszugebenden Zeichen:

  Im folgenden sei *n* die Gesamtanzahl der auszugebenden Zeichen, d.h.

  $n = size * nitems$

  – Wenn *n* größer als die maximale Satzlänge ist, wird nur ein Satz mit maximaler Satzlänge geschrieben. Die restlichen Daten gehen verloren.

  – Wenn *n* kleiner als die minimale Satzlänge ist, wird kein Satz geschrieben. Die minimale Satzlänge ist nur für ISAM-Dateien definiert und bedeutet, dass *n* mindestens den Bereich des Schlüssels im Satz umfassen muss.

  – Wenn *n* beim Neuschreiben eines Satzes in eine Datei mit fester Satzlänge kleiner als die Satzlänge ist, wird der Satz am Ende mit binären Nullen aufgefüllt.

  – Beim Update eines bestehenden Satzes in einer sequenziellen Datei (SAM, PAM) muss *n* gleich der Länge des zu aktualisierenden Satzes sein. Im anderen Fall tritt ein Fehler auf. Als Satzlänge für PAM-Dateien gilt die Länge eines logischen Blocks.

  - Beim Update eines bestehenden Satzes in einer indexsequenziellen Datei (ISAM) braucht *n* nicht gleich der Länge des zu aktualisierenden Satzes sein. Ein Satz kann also verkürzt oder verlängert werden.

- In ISAM-Dateien, für die Schlüsselverdoppelung zugelassen ist, ist kein direkter Update eines Satzes möglich. Beim Schreiben eines Satzes mit einem bereits existierenden Schlüssel wird stets ein neuer Satz geschrieben. Der alte Satz muss explizit gelöscht werden.
- `fwrite()` liefert den gleichen Returnwert wie bei Datenstrom-Ein-/Ausgabe, nämlich die Anzahl der vollständig geschriebenen Elemente. Bei Satz-Ein-/Ausgabe ist es sinnvoll, ausschließlich die Elementlänge 1 zu verwenden, da in diesem Fall der Returnwert der Länge des geschriebenen Satzes entspricht (ohne ein ggf. vorhandenes Satzlängenfeld).
  Bei fester Satzlänge wird jedoch das (ggf. notwendige) Auffüllen mit binären Nullen im Returnwert nicht berücksichtigt. ❑

Returnwert

Anzahl der erfolgreich geschriebenen Elemente	bei Erfolg. Diese kann dann kleiner als *nitems* sein, wenn ein Schreibfehler auftritt.
0	wenn *size* oder *nitems* gleich 0 sind. Der Inhalt des Vektors und der Zustand des Datenstroms bleiben unverändert.
	wenn ein Schreibfehler auftritt, wird das Fehlerkennzeichen für den Datenstrom gesetzt. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler Siehe `fputc()`.

Hinweis Um sicherzugehen, dass *size* die richtige Anzahl Bytes für ein Datenelement angibt, sollten Sie die Funktion `sizeof()` für die Größe einer Dateneinheit verwenden, auf die *ptr* zeigt.

Bei der Ausgabe in Dateien mit Datenstrom-Ein-/Ausgabe werden die Daten nicht sofort in die externe Datei geschrieben, sondern in einem C-internen Puffer zwischengespeichert (siehe Abschnitt „Pufferung von Datenströmen“ auf Seite 77).

Bei der Ausgabe in Textdateien werden die Steuerzeichen für Zwischenraum (`\n`, `\t` etc.) je nach Art der Textdatei in ihre entsprechende Wirkung umgesetzt (siehe Abschnitt „Zwischenraumzeichen“ auf Seite 85).

Ob `fwrite()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

Siehe auch `ferror()`, `fopen()`, `printf()`, `putc()`, `puts()`, `write()`, `stdio.h`, `sys/stat.h`.

gamma - Logarithmus der Gamma-Funktion berechnen

Syntax

#include <math.h>

double gamma(double x);

extern int signgam;

Beschreibung

`gamma()` berechnet die mathematische Gammafunktion für die Gleitpunktzahl x:

$$\int_0^{\infty} e^{-t}\ t^{x-1}\ dt$$

Das Vorzeichen dieses Wertes wird in der C-internen Variablen `signgam` als +1 oder -1 abgelegt. `signgam` darf nicht vom Anwender definiert werden.

`gamma()` ist nicht reentrant.

Returnwert

`gamma(x)`	bei Erfolg.
`HUGE_VAL`	falls der korrekte Returnwert einen Überlauf ergibt. `errno` wird gesetzt, um den Fehler anzuzeigen.
`HUGE_VAL`	falls x eine nichtpositive Ganzzahl ist. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler

`gamma()` schlägt fehl, wenn gilt:

`ERANGE`	Überlauf, der Returnwert ist zu groß.
`EDOM`	x ist eine nichtpositive Ganzzahl.

Siehe auch

`lgamma()`, `math.h`.

garbcoll - Speicherbereich an das System freigeben *(BS2000)*

Syntax #include <stdlib.h>

void garbcoll(void);

Beschreibung

Die Funktionen `calloc()`, `malloc()`, `realloc()` und `free()` bilden das C-spezifische Speicherverwaltungspaket. Dieses Paket besteht im Wesentlichen aus einer internen Freispeicherverwaltung.
Der mit `free()` freigegebene Speicher wird nicht an das System zurückgegeben (RELM-SVC), sondern durch die Freispeicherverwaltung erfasst.
Die Funktionen für Speicheranforderungen (`calloc()`, `malloc()`, `realloc()`) versuchen, den Speicher zuerst über die Freispeicherverwaltung zu besorgen und erst in zweiter Linie vom Betriebssystem (REQM-SVC).

Falls auch vom System kein Speicher mehr erhältlich ist, wird der in der Freispeicherverwaltung erfasste Speicher so weit wie möglich seitenweise an das System zurückgegeben (Garbage Collection).

Dieser Garbage-Collection-Mechanismus wird im Adressraum ≤ 2 GB wirksam und ist mit der Funktion `garbcoll()` auch explizit aufrufbar.

Hinweis `garbcoll()` gibt alle Speicherbereiche an das System zurück, die zuvor mit `free()` freigegeben wurden und sich zu freien Seiten zusammenstellen lassen.

Siehe auch `calloc()`, `malloc()`, `realloc()`, `free()`.

gcvt - Gleitpunktzahl in Zeichenkette umwandeln

Syntax #include <stdlib.h>

char *gcvt(double *value*, int *ndigit*, char **buf*);

Beschreibung

Siehe `ecvt()`.

getc - Byte aus Datenstrom lesen

Syntax

```
#include <stdio.h>

int getc(FILE *stream);
```

Beschreibung

`Die Funktion getc()` ist äquivalent zu `fgetc()`, außer dass sie, wenn sie als Makro definiert ist, das Argument *stream* öfter als einmal auswertet. Daher sollte dieses Argument niemals ein Ausdruck mit Seiteneffekten sein.

`getc()` ist sowohl als Funktion als auch als Makro definiert.

`getc(stdin)` ist identisch mit `getchar()`.

`Die Funktion getc_unlocked()` ist funktional gleichwertig mit `getc()`, mit der Ausnahme, dass sie nicht threadsicher implementiert ist. Sie kann deshalb in einem Multithread-Programm nur sicher genutzt werden, wenn der Thread, der sie aufruft, das entsprechende (FILE *) Objekt besitzt. Dies ist der Fall nach einem erfolgreichen Aufruf der Funktionen `flockfile()` oder `ftrylockfile()`.

Returnwert Siehe `fgetc()`.

Fehler Siehe `fgetc()`.

Hinweis Wenn der ganzzahlige Returnwert von `getc()` in einer Variablen vom Typ `char` abgelegt und dann mit der ganzzahligen Konstanten `EOF` verglichen wird, ist dieser Vergleich nicht erfolgreich, da die Vorzeichen-Erweiterung einer Variablen vom Typ `char` bei der Umwandlung in einen `int`-Typ rechnerabhängig ist. Deshalb sollten portable Anwendungen darauf achten, dass der Returnwert von `getc()` stets in einer Variablen vom Typ `int` abgelegt wird.

Wenn z.B. in einem Programm der folgende Vergleich verwendet wird, muss die Variable *c* als `int`-Größe vereinbart werden:

```
while((c = fgetc(dz)) != EOF)
```

Wenn nämlich *c* als `char`-Größe definiert werden würde, würde die Bedingung `EOF` aus folgendem Grund nie erfüllt: -1 wird in den `char`-Wert 0xFF (also +255) konvertiert. `EOF` ist jedoch -1.

Da `getc()` als Makro implementiert sein kann, kann *stream* mit Seiteneffekten inkorrekt behandelt werden. Inbesondere kann `getc(*f++)` anders funktionieren, als man es erwartet. Daher wird der Gebrauch von `getc()` in solchen Situationen nicht empfohlen; stattdessen sollte `fgetc()` benutzt werden.

Wenn `fgetc()` in der POSIX-Umgebung von `stdin` liest und `EOF` das Einlese-Endekriterium ist, erreicht man die `EOF`-Bedingung durch folgende Maßnahmen:

- am blockorientierten Terminal durch Eingabe der Tastensequenz @ @ d
- am zeichenorientierten Terminal durch Eingabe von CTRL + D

BS2000
Wenn `fgetc()` in der BS2000-Umgebung von `stdin` liest und `EOF` das Einlese-Endekriterium ist, erreicht man die `EOF`-Bedingung durch folgende Maßnahmen am Terminal:

1. K2 drücken.
2. Die Systemkommandos `EOF` und `RESUME-PROGRAM` eingeben. ❑

Ob `getc()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

Siehe auch `fgetc()`, `putc()`, `putchar_unlocked()`, `stdio.h`.

getc_unlocked, getchar_unlocked, putc_unlocked, putchar_unlocked - Standardeingabe/-ausgabe mit expliziter Sperrung durch den Client

Syntax

```
#include <stdio.h>

int getc_unlocked(FILE *stream);

int getchar_unlocked(void);

int putc_unlocked(int c, FILE *stream);

int putchar_unlocked(int c);
```

Beschreibung

Die Funktionen `getc_unlocked()`, `getchar_unlocked()`, `putc_unlocked()` bzw. `putchar_unlocked()` sind funktional gleichwertig mit den Originalversionen `getc()`, `getchar()`, `putc()` und `putchar()` sind mit der Ausnahme, dass sie nicht threadsicher implementiert werden müssen.

Sie können deshalb in einem Multithread-Programm nur sicher genutzt werden, wenn der Thread, der sie aufruft, das entsprechende (FILE *) Objekt besitzt. Dies ist der Fall nach einem erfolgreichen Aufruf der Funktionen `flockfile()` oder `ftrylockfile()`.

Returnwert Siehe `getc()`, `getchar()` [beide in `getc()`], `putc()` und `putchar()` [beide in `putc()`].

Siehe auch `getc()`, `putc()`, `flockfile()`, `pthread_intro()`, `stdio()`.

getchar - Byte aus Standard-Eingabestrom lesen

Syntax

```
#include <stdio.h>

int getchar(void);
```

Beschreibung

Der Funktionsaufruf `getchar(void)` ist äquivalent zu `getc(stdin)`, d.h. `getchar()` liest 1 Byte aus dem Standard-Eingabestrom.

Returnwert Siehe `fgetc()`.

Fehler Siehe `fgetc()`.

Hinweis Wenn der ganzzahlige Returnwert von `getchar()` in einer Variablen vom Typ `char` abgelegt und dann mit der ganzzahligen Konstanten `EOF` verglichen wird, kann es sein, dass dieser Vergleich nicht erfolgreich ist, da die Vorzeichen-Erweiterung einer Variablen vom Typ `char` bei der Umwandlung in einen `int`-Typ rechnerabhängig ist. Deshalb sollten portable Anwendungen darauf achten, dass der Returnwert von `getc()` stets in einer Variablen vom Typ `int` abgelegt wird.

Wenn z.B. in einem Programm der folgende Vergleich verwendet wird, muss die Variable *c* als `int`-Größe vereinbart werden:

```
while((c = fgetc(dz)) != EOF)
```

Wenn nämlich *c* als `char`-Größe definiert werden würde, würde die Bedingung `EOF` aus folgendem Grund nie erfüllt: -1 wird in den `char`-Wert 0xFF (also +255) konvertiert. `EOF` ist jedoch -1.

Wenn `fgetc()` in der POSIX-Umgebung von `stdin` liest und `EOF` das Einlese-Endekriterium ist, erreicht man die `EOF`-Bedingung durch folgende Maßnahmen:

- am blockorientierten Terminal durch Eingabe der Tastensequenz @ @ d
- am zeichenorientierten Terminal durch Eingabe von CTRL + D

BS2000

Wenn `fgetc()` in der BS2000-Umgebung von `stdin` liest und `EOF` das Einlese-Endekriterium ist, erreicht man die `EOF`-Bedingung durch folgende Maßnahmen am Terminal:

1. K2 drücken.
2. Die Systemkommandos `EOF` und `RESUME-PROGRAM` eingeben. ❑

Ob `getchar()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

Siehe auch `fgetc()`, `getc()`, `stdio.h`.

getchar_unlocked - Standardeingabe mit expliziter Sperrung durch den Client

Syntax #include <stdio.h>

int getchar_unlocked(void);

Beschreibung

siehe `getc_unlocked()`.

getcontext, setcontext - Benutzerkontext anzeigen oder ändern

Syntax

#include <ucontext.h>

int getcontext(ucontext_t *ucp);

int setcontext(const ucontext_t *ucp);

Beschreibung

Diese Funktionen dienen im Zusammenhang mit den in `makecontext()` definierten Funktionen zur Implementierung der Kontextwechsel auf Benutzerebene zwischen mehreren Kontrollflüssen eines Prozesses.

`getcontext()` initialisiert die Struktur, auf die *ucp* zeigt, als aktuellen Benutzerkontext des aufrufenden Prozesses. Die Struktur `ucontext_t`, auf die *ucp* zeigt, definiert den Benutzerkontext und enthält die Inhalte der Maschinenregister, der Signalmaske und des Stacks des aufrufenden Prozesses.

`setcontext()` restauriert den Benutzerkontext, auf den *ucp* zeigt. Ein erfolgreicher Aufruf von `setcontext()` kehrt nicht zurück; die Programmausführung fährt an der Stelle fort, auf die die Kontextstruktur aus `setcontext()` zeigt. Die Kontextstruktur sollte durch einen vorhergehenden Aufruf von `getcontext()` erzeugt werden oder wurde vom System als drittes Argument an eine Signalbehandlungsroutine (siehe `sigaction()`) geliefert.

- Wenn die Kontextstruktur mit `getcontext()` erzeugt wurde, wird die Programmausführung wieder aufgenommen, als ob der entsprechende Aufruf von `getcontext()` zurückgekehrt wäre.
- Wenn die Kontextstruktur mit `makecontext()` erzeugt wurde, wird die Programmausführung mit der mit `makecontext()` angegebenen Funktion wieder aufgenommen. Wenn diese Funktion zurückkehrt, wird der Prozess wie nach einem Aufruf von `setcontext()` mit dem Argument *ucp* fortgesetzt, das auch Argument für `makecontext()` war.
- Wenn das Argument *ucp* einer Signalbehandlungsroutine übergeben wurde, wird die Programmausführung mit dem nächsten Befehl fortgesetzt, der auf den durch das Signal unterbrochenen Befehl folgt.

Wenn die Komponente `uc_link` aus der Struktur `ucontext_t`, auf die *ucp* zeigt, den Wert 0 hat, dann handelt es sich um den Basisprozess, und der Prozess beendet sich, wenn dieser Kontext beendet wird. Die Verwendung eines Arguments *ucp*, das anders als oben beschrieben erzeugt wurde, führt zu unvorhersagbaren Ergebnissen.

Werden Threads verwendet, so wirkt sich die Funktion auf den Prozess oder auf einen Thread wie folgt aus:

- `getcontext()` holt den aktuellen Benutzerkontext des aufrufenden Threads.
- `setcontext()` setzt den aktuellen Benutzerkontext des aufrufenden Threads.

Returnwert `getcontext()`:

0	bei Erfolg.
-1	bei Fehler.

`setcontext()`:

kehrt nicht zurück bei Erfolg.

-1	bei Fehler.

Hinweis Wenn eine Signalbehandlungsroutine ausgeführt wird, wird der Benutzerkontext gespeichert und ein neuer Kontext erzeugt. Wenn der Prozess die Signalbehandlungsroutine über `longjmp()` verlässt, so wird der ursprüngliche Kontext nicht restauriert, und zukünftige Aufrufe von `getcontext()` sind nicht mehr zuverlässig. Signalbehandlungsroutinen sollten deshalb `siglongjmp()` oder `setcontext()` verwenden.

Portable Anwendungen sollten auf die Komponente uc_mcontext der Struktur `ucontext_t` weder zugreifen noch diese verändern. Eine portable Anwendung kann nicht davon ausgehen, dass `getcontext()` in ucp statische Daten des Prozesses speichert, auch nicht `errno`.
Bei der Manipulation von Kontexten ist Vorsicht geboten.

Siehe auch `bsd_signal()`, `makecontext()`, `setjmp()`, `sigaction()`, `sigaltstack()`, `sigprocmask()`, `sigsetjmp()`, `ucontext.h`.

getcwd - Pfadnamen des aktuellen Dateiverzeichnisses ermitteln

Syntax #include <unistd.h>

char *getcwd(char **buf*, int *size*);

Beschreibung

`getcwd()` gibt einen Zeiger auf den Pfadnamen des aktuellen Dateiverzeichnisses zurück. Der Wert von *size* muss wenigstens um eins größer als die Länge des zurückzugebenden Pfadnamens sein.

Wenn *buf* nicht null ist, wird der Pfadname in dem Speicherplatz gespeichert, auf den *buf* zeigt.

Wenn *buf* ein Nullzeiger ist, erhält `getcwd()` durch Aufruf von `malloc()` *size* Bytes Speicherplatz. In diesem Fall kann der von `getcwd ()` zurückgegebene Zeiger als Argument in einem nachfolgenden Aufruf von `free()` verwendet werden.

Das aktuelle Dateiverzeichnis entspricht nur solange dem Home-Verzeichnis, bis ein Aufruf von `chdir()` erfolgt. Das Home-Verzeichnis kann mit `getpwuid()` oder `getpwnam()` abgefragt werden. Beide Funktionen geben eine Struktur zurück, die auch einen Zeiger auf das ursprüngliche Arbeitsdateiverzeichnis enthält.

Beim Starten eines C-Programms wird als aktuelles Dateiverzeichnis das in der Datei `SYSSRPM` hinterlegte so genannte Home-Verzeichnis eingestellt. Falls die Umgebungsvariable `HOME` für ein C-Programm definiert ist, wird das Home-Verzeichnis auf diesen Wert eingestellt.

Existiert das in der Datei `SYSSRPM` eingetragene Verzeichnis nicht, wird der Schrägstrich (`/`) zurückgegeben.

BS2000
Wenn es eine SDF-P-Variable `SYSPOSIX.HOME` gibt, so wird die Variable `HOME` der C-Programmumgebung mit dem Wert der Variable `SYSPOSIX.HOME` initialisiert. ❑

Mit einem Aufruf von `chdir()` kann das aktuelle Dateiverzeichnis jederzeit gewechselt werden. Ein Aufruf von `chdir()` hat nur eine Wirkung für die Dauer des aufrufenden Programms. Das Home-Verzeichnis wird dabei nicht verändert.

Returnwert 0 wenn *size* nicht groß genug ist oder wenn ein Fehler in einer unten liegenden Funktion auftritt. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `getcwd()` schlägt fehl, wenn gilt:

`EACCES` Ein übergeordnetes Verzeichnis kann nicht gelesen werden, um seinen Namen zu erhalten.

`EINVAL` *size* ist gleich 0.

ENOMEM	Es ist nicht genügend Speicherplatz vorhanden.
ERANGE	ist kleiner als 0 oder größer als 0 und kleiner als die Länge des Pfadnamens plus 1.

Hinweis `getcwd()` wird nur für POSIX-Dateien ausgeführt.

Siehe auch `malloc()`, `unistd.h`.

getdate - Zeit und Datum in Benutzerformat umwandeln

Syntax

```
#include <time.h>

struct tm *getdate (const char *string);

extern int getdate_err;
```

Beschreibung

`getdate()` wandelt benutzerdefinierbare Datums- und/oder Zeitangaben aus *string* in eine `tm`-Struktur um. Die Strukturdeklaration befindet sich in der Datei `time.h` (siehe auch `ctime()`).

Zum Zerlegen und Interpretieren der Eingabezeichenkette werden benutzerdefinierte Schablonen verwendet. Diese Schablonen sind Textdateien, welche der Benutzer anlegt; diese Textdateien werden über die Umgebungsvariable `DATEMSK` angegeben. Jede Zeile der Schablone stellt eine akzeptierbare Datums- und/oder Zeitangabe dar; dabei werden einige der Felddeskriptoren verwendet, die auch das Kommando `date` verwendet. Die erste Zeile in der Schablone, die der Eingabespezifikation entspricht, wird zur Interpretation und Umwandlung in das interne Zeitformat verwendet. Ist die Operation erfolgreich, so liefert die Funktion `getdate()` einen Zeiger auf eine Struktur vom Typ `tm` zurück; ansonsten wird `NULL` zurückgegeben und die globale Variable `getdate_err` gesetzt.

Die folgenden Felddeskriptoren werden unterstützt:

`%%`	das Gleiche wie `%`
`%a`	abgekürzter Wochentagsname
`%A`	ausgeschriebener Wochentagsname
`%b`	abgekürzter Monatsname
`%B`	ausgeschriebener Monatsname
`%c`	lokale Datums- und Zeitdarstellung
`%d`	Monatstag (01 - 31; die führende 0 ist optional)
`%e`	das Gleiche wie `%d`
`%D`	Datum als `%m/%d/%y`
`%h`	abgekürzter Monatsname
`%H`	Stunde (00 - 23)
`%I`	Stunde (01 - 12)
`%m`	Monatsnummer (01 - 12)
`%M`	Minute (00 - 59)
`%n`	das Gleiche wie `\n`
`%p`	lokales Äquivalent zu AM oder PM
`%r`	Zeit als `%I:%M:%S %p`
`%R`	Zeit als `%H:%M`
`%S`	Sekunde (00-61). Schaltsekunden sind erlaubt, jedoch sind Folgeeffekte bei der Benutzung von Algorithmen nicht vorhersehbar.

`%t` Tabulator einfügen
`%T` Zeit als `%H:%M:%S`
`%w` Wochentagsnummer (0 - 6; Sonntag = 0)
`%x` lokale Datumsrepräsentation
`%X` lokale Zeitrepräsentation
`%y` Jahr im aktuellen Jahrhundert (00 - 99)
`%Y` Jahr als `ccyy` (z.B. 1997)
`%Z` Zeitzonenname oder keine Zeichen, wenn keine Zeitzone existiert. Wenn die Zeitzone unter `%Z` nicht die Zeitzone ist, die `getdate()` erwartet, tritt ein Eingabefehler auf. `getdate()` berechnet eine passende Zeitzone ausgehend von den Daten, die der Funktion übergeben wurden, (wie z.B Stunde, Tag und Monat).

Beim Vergleich zwischen der Schablone und der Eingabespezifikation unterscheidet `getdate()` nicht zwischen Klein- und Großbuchstaben.

Die Monats- und Wochentagsnamen können aus einer beliebigen Kombination von kleinen und großen Buchstaben bestehen. Der Benutzer kann bestimmen, dass die Angabe der Eingabezeit oder des Eingabedatums sprachabhängig ist. Dies geschieht durch Setzen der Werte `LC_TIME` und `LC_CTYPE` bei `setlocale()`.

Die Deskriptoren, bei denen Ziffern angegeben werden müssen, haben höchstens zwei Stellen. Führende Nullen sind erlaubt, können aber auch weggelassen werden. Leerstellen in der Schablone oder in *string* werden ignoriert.

Die Felddeskriptoren `%c`, `%x` und `%X` werden abgewiesen, wenn sie unzulässige Felddeskriptoren enthalten.

Die folgenden Regeln gelten für die Umwandlung von Eingabespezifikationen in das interne Format:

– Wenn `%Z` angegeben wird, setzt `getdate()` die Elemente der `tm`-Struktur auf die aktuelle Zeit der angegebenen Zeitzone. Andernfalls wird die formatierte Zeit mit der aktuellen Ortszeit initialisiert, als ob`localtime()` ausgeführt worden wäre.

– Ist nur der Wochentag angegeben, wird der aktuelle Tag angenommen, wenn der angegebene Wochentag identisch mit dem aktuellen Tag ist. Liegt der angegebene Tag vor dem aktuellen, wird der Wochentag aus der nächsten Woche genommen.

– Ist nur der Monat angegeben, wird der aktuelle Monat angenommen, wenn der angegebene Monat gleich dem aktuellen Monat ist. Ist der angegebene Monat kleiner als der aktuelle Monat, wird das nächste Jahr angenommen, wenn ansonsten kein Jahr angegeben ist. (Der erste Tag des Monats wird angenommen, wenn kein Tag angegeben ist.)

– Wird keine Stunde, Minute und Sekunde angegeben, wird die aktuelle Stunde, Minute und Sekunde übernommen.

– Wird kein Datum angegeben, wird der aktuelle Tag angenommen, wenn die angegebene Stunde größer als die aktuelle Stunde ist. Ist die angegebene Stunde kleiner als die aktuelle, so wird der nächste Tag angenommen.

`getdate()` benutzt die externe Variable oder das Makro `getdate_err`, um das Fehlergewicht zurückzugeben.

Returnwert Zeiger auf eine Struktur `tm`
bei Erfolg.

Nullzeiger bei Fehler. `getdate_err` wird gesetzt, um den Fehler anzuzeigen.

Fehler `getdate()` schlägt fehlt, wenn einer der folgenden Fehler auftritt. Die Fehlergewichte werden in `getdate_err` zurückgeliefert. Der Inhalt von `errno` ist dabei ohne Bedeutung.

1 Die Umgebungsvariable `DATEMSK` ist undefiniert oder null.

2 Die Schablonendatei kann nicht zum Lesen geöffnet werden.

3 Der Dateistatus konnte nicht gelesen werden.

4 Die Schablonendatei ist keine reguläre Datei.

5 Ein Fehler trat beim Lesen der Schablonendatei auf.

6 `malloc()` konnte nicht erfolgreich ausgeführt werden, da zu wenig Speicherplatz verfügbar war.

7 Es gibt keine Zeile aus der Schablonendatei, die der Eingabe entspricht.

8 Das Eingabeformat ist ungültig, z.B. `February 31`, oder es wurde eine Zeit angegeben, die nicht in einem Typ `time_t` dargestellt werden kann; `time_t` enthält die Zeit in Sekunden seit 00:00:00 UTC, was dem 1. Januar 1970 entspricht.

Hinweis Nachfolgende Aufrufe von `getdate()` ändern den Inhalt von `getdate_err`.

Die Deklaration der externen Variablen `getdate_err` ist in der Include-Datei `time.h` enthalten. `getdate_err` sollte daher nicht explizit im Programm deklariert werden, sondern es sollte `time.h` eingefügt werden.

Daten vor 1970 und nach 2037 sind ungültig.

Beispiel 1 Möglicher Inhalt einer Schablone:

```
%m
%A %B %d, %Y, %H:%M:%S
%A
%B
%m/%d/%y %I %p
%d,%m,%Y %H:%M
at %A the %dst of %B in %Y
run job at %I %p,%B %dnd
%A den %d. %B %Y %H.%M Uhr
```

Beispiel 2 Einige Beispiele für gültige Eingabespezifikationen für die Schablone aus Beispiel 1:

```
getdate("10/1/87 4 PM");
getdate("Friday");
getdate("Friday September 19 1987, 10:30:30");
getdate("24,9,1986 10:30");
getdate("at monday the 1st of december in 1986");
getdate("run job at 3 PM, december 2nd");
```

Wenn die Umgebungsvariable `LC_TIME` gesetzt ist bzw. `LANG` auf `german` gesetzt wird, ist folgende Angabe gültig:

```
getdate("Freitag den 10. Oktober 1986 10.30 Uhr");
```

Beispiel 3 Lokale Zeit- und Datumsangaben werden ebenfalls unterstützt. Beispiel 3 zeigt, wie lokale Datums- und Zeitangaben in Schablonen definiert werden können.

Aufruf	**Zeile in Schablonendatei**
`getdate("11/27/86");`	`%m/%d/%y`
`getdate("27.11.86");`	`%d.%m.%y`
`getdate("86-11-27");`	`%y-%m-%d`
`getdate("Friday 12:00:00");`	`%A %H:%M:%S`

Beispiel 4 Die folgenden Beispiele verdeutlichen die obigen Regeln. Es wird angenommen, dass das aktuelle Datum Montag 22. September 12:19:47 EDT 1986 ist und die Umgebungsvariablen `LANG` und `LC_TIME` nicht gesetzt sind.

Eingabe	**Zeile in Schablonendatei**	**Datum**
Mon	%a	Mon Sep 22 12:19:48 EDT 1986
Sun	%a	Sun Sep 28 12:19:49 EDT 1986
Fri	%a	Fri Sep 26 12:19:49 EDT 1986
September	%B	Mon Sep 1:19:49 EDT 1986
January	%B	Thu Jan 1:19:49 EST 1987
December	%B	Mon Dec 1:19:49 EST 1986
Sep Mon	%b %a	Mon Sep 1:19:50 EDT 1986
Jan Fri	%b %a	Fri Jan 2 12:19:50 EST 1987
Dec Mon	%b %a	Mon Dec 1:19:50 EST 1986
Jan Wed 198	%b %a %Y	Wed Jan 4 12:19:51 EST 1989
Fri 9	%a %H	Fri Sep 26 09:00:00 EDT 1986
Feb 10:30	%b %H:%S	Sun Feb 1 10:00:30 EST 1987
10:30	%H:%M	Tue Sep 23 10:30:00 EDT 1986
13:30	%H:%M	Mon Sep 22 13:30:00 EDT 1986

Siehe auch `ctime()`, `localtime()`, `setlocale()`, `strftime()`, `times()`, `time.h`.

getdents - Verzeichniseinträge umwandeln

Name **getdents, getdents64**

Syntax #include <sys/dirent.h>

int getdents(int *fildes*, struct dirent **buf*, size_t *nbyte*);
int getdents64(int *fildes*, struct dirent64 **buf*, size_t *nbyte*);

Beschreibung

fildes ist ein Dateideskriptor, der von einem `open()`- oder `dup()`-Systemaufruf geliefert wird.
`getdents()` versucht, *nbyte* Bytes aus dem zu *fildes* gehörenden Verzeichnis zu lesen und diese als vom Dateisystem unabhängige Verzeichnis-Einträge in den Puffer zu bringen, auf den *buf* zeigt. Da die vom Dateisystem unabhängigen Verzeichnis-Einträge unterschiedlich lang sind, ist die tatsächliche Anzahl zurückgegebener Bytes in den meisten Fällen wesentlich kleiner als *nbyte*.

Sehen Sie in `dirent()` (Referenzhandbuch für Systemverwalter) nach, um die Anzahl der Bytes zu berechnen.
Der dateisystemunabhängige Verzeichnis-Eintrag wird durch die Struktur `dirent` angegeben.
Eine Beschreibung hiervon ist in `dirent()`zu finden.
Bei Geräten, die positionieren können, beginnt `getdents()` an der Stelle in der Datei, die durch den *fildes* zugeordneten Schreib-/Lesezeiger angegeben wird. Nach Rückkehr von `getdents()` wird der Schreib-/Lesezeiger erhöht, damit er auf den nächsten Verzeichnis-Eintrag zeigt. Dieser Systemaufruf wurde für die Implementierung der Funktion `readdir()` entwickelt (eine Beschreibung ist in `directory()` zu finden) und sollte daher nicht für andere Zwecke verwendet werden.

Es besteht kein funktionaler Unterschied zwischen `getdents()` und `getdents64()`, außer dass bei `getdents64()` *buf* auf eine `dirent64`-Struktur zeigt.

Fehler Die folgenden Beschreibungen der Fehlercodes sind funktionsspezifisch. Eine allgemeingültige Beschreibung finden Sie in `intro_prm2()` bzw. in `errno()`.

`getdents()` und `getdents64()` sind erfolglos, wenn einer oder mehrere der nachstehenden Punkte zutreffen:

`EBADF`	*fildes* ist kein zum Lesen geöffneter, gültiger Dateideskriptor.
`EFAULT`	*buf* weist über den zugewiesenen Adressraum hinaus.
`EINVAL`	*nbyte* ist für einen Verzeichnis-Eintrag nicht groß genug.
`ENOENT`	Der aktuelle Schreib-/Lesezeiger für das Verzeichnis befindet sich nicht auf einem gültigen Eintrag.

ENOLINK	*fildes* weist auf einen fernen Rechner, und die Verbindung zu diesem Rechner ist nicht mehr aktiv.
ENOTDIR	*fildes* ist kein Verzeichnis.
EIO	Während des Zugriffs auf das Dateisystem ist ein E/A-Fehler aufgetreten.

Returnwert

Nach erfolgreicher Beendigung wird eine nicht negative ganze Zahl zurückgegeben, die die Anzahl der tatsächlich gelesenen Bytes angibt. Der Wert 0 zeigt an, dass das Ende des Verzeichnisses erreicht wurde. War der Systemaufruf erfolglos, wird -1 zurückgegeben und `errno` zur Anzeige des Fehlers gesetzt.

Siehe auch `directory()`, `dirent()`.

getdtablesize - Größe der Deskriptor-Tabelle abrufen

Syntax #include <unistd.h>

int getdtablesize(void);

Beschreibung

`getdtablesize()` entspricht der Funktion `getrlimit()`, wenn `RLIMIT_NOFILE` angegeben wird.

`getdtablesize()` ist nicht threadsicher.

Returnwert Aktueller Grenzwert für die Anzahl gleichzeitig geöffneter Dateideskriptoren pro Prozess bei Erfolg.

-1 bei Fehler.

Hinweis Es besteht kein unmittelbarer Zusammenhang zwischen dem Wert, den `getdtablesize()` zurückliefert und der Konstanten `{OPEN_MAX}`, die in `limits.h` definiert ist.

Siehe auch `close()`, `getrlimit()`, `open()`, `select()`, `setrlimit()`, `limits.h`, `unistd.h`.

getegid - effektive Gruppennummer eines Prozesses ermitteln

Syntax #include <unistd.h>

Optional
#include <sys/types.h> ❑

gid_t getegid(void);

Beschreibung

`getegid()` gibt die effektive Gruppennummer des aufrufenden Prozesses zurück.

Returnwert effektive Gruppennummer des aufrufenden Prozesses
Die Funktion ist immer erfolgreich.

Siehe auch `getgid()`, `setgid()`, `sys/types.h`, `unistd.h`., Handbuch „POSIX Grundlagen (BS2000/OSD)“.

getenv - Wert einer Umgebungsvariablen ermitteln

Syntax

```
#include <stdlib.h>

char *getenv(const char *name);
```

Beschreibung

`getenv()` durchsucht die aktuelle Umgebung des Prozesses, d.h. den Zeichenkettenvektor, auf den `environ` zeigt, nach einer Zeichenkette der Form "*name=value*" und gibt einen Zeiger auf die Zeichenkette zurück, die den Wert *value* für den angegebenen Variablennamen *name* enthält.

`getenv()` ist nicht threadsicher.

Returnwert

Wert von *name*
: wenn eine entsprechende Zeichenkette vorhanden ist.

Nullzeiger
: wenn keine entsprechende Zeichenkette vorhanden ist,
oder wenn die Anwendung mit BS2000-Funktionalität aufgerufen wird (siehe Abschnitt „Umfang der unterstützten C-Bibliothek“ auf Seite 18).

Hinweis

Die Zeichenkette "*name=value*" darf nicht verändert werden. Sie kann jedoch von nachfolgenden `putenv`-Aufrufen überschrieben werden. Andere Bibliotheksfunktionen überschreiben die Zeichenkette nicht.

BS2000
Der Inhalt des Zeichenkettenvektors, auf den `environ` zeigt, kann beim Programmstart mit Werten aus der SDF-P-Variablen `SYSPOSIX.`*name* besetzt werden (siehe `environ` und Abschnitt „Umgebungsvariablen“ auf Seite 71). ❑

Siehe auch

`exec`, `environ`, `putenv()`, `stdlib.h`, siehe Abschnitt „Umfang der unterstützten C-Bibliothek“ auf Seite 18 und Abschnitt „Umgebungsvariablen“ auf Seite 71.

geteuid - effektive Benutzernummer eines Prozesses ermitteln

Syntax #include <unistd.h>

Optional
#include <sys/types.h> ❑

uid_t geteuid(void);

Beschreibung
`geteuid()` gibt die effektive Benutzernummer des aufrufenden Prozesses zurück.

Returnwert effektive Benutzernummer des aufrufenden Prozesses
Die Funktion ist immer erfolgreich.

Siehe auch `getuid()`, `setuid()`, `sys/types.h`, `unistd.h`, Handbuch „POSIX Grundlagen (BS2000/OSD)“.

getgid - reale Gruppennummer eines Prozesses ermitteln

Syntax #include <unistd.h>

Optional
#include <sys/types.h> ❑

gid_t getgid(void);

Beschreibung
`getgid()` gibt die reale Gruppennummer des aufrufenden Prozesses zurück.

Returnwert reale Gruppennummer des aufrufenden Prozesses
Die Funktion ist immer erfolgreich.

Siehe auch `getegid()`, `getuid()`, `setgid()`, `sys/types.h`, `unistd.h`, Handbuch „POSIX Grundlagen (BS2000/OSD)“.

getgrent - Gruppendatei-Eintrag bestimmen

Syntax #include <grp.h>

struct group *getgrent (void);

Beschreibung

Siehe `endgrent()`.

getgrgid - Gruppendateieintrag für Gruppennummer ermitteln

Syntax

```
#include <grp.h>

Optional
#include <sys/types.h> ❑

struct group *getgrgid(gid_t gid);
```

Beschreibung

`getgrgid()` durchsucht die Gruppendatei nach einem Eintrag, dessen Komponente *gr_gid* mit *gid* übereinstimmt (siehe `grp.h` und Handbuch „POSIX Grundlagen (BS2000/OSD)“).

`getgrgid()` ist nicht threadsicher. Verwenden Sie bei Bedarf die reentrante Funktion `getgrgid_r()`.

Returnwert

Zeiger auf ein Objekt der Struktur `group`
: wenn ein Eintrag gefunden wurde, dessen Komponente *gr_gid* mit *gid* übereinstimmt.

Nullzeiger
: wenn ein Fehler auftritt oder kein Eintrag gefunden wurde, dessen Komponente *gr_gid* mit *gid* übereinstimmt. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler

`getgrgid()` schlägt fehl, wenn gilt:

`EIO`
: Ein Ein-/Ausgabefehler ist aufgetreten.

`EINTR`
: Während des Ablaufs von `getgrgid()` wurde ein Signal abgefangen.

`EMFILE`
: Für den aktuellen Prozess sind derzeit zu viele Dateideskriptoren offen.

`ENFILE`
: Die Dateitabelle des Systems ist derzeit voll.

Hinweis

Der Returnwert kann auf einen statischen Bereich zeigen, der durch einen späteren `getgrid`- oder `getgrnam`-Aufruf überschrieben werden kann.

Da `getgrgid()` Funktionen zur Dateiverarbeitung aufruft, die auf Fehler laufen können, sollte `errno` vor dem Aufruf von `getgrgid()` auf 0 gesetzt werden. Wenn `errno` nach Rückkehr der Funktion einen anderen Wert hat, dann ist ein Fehler aufgetreten.

Siehe auch

`getgrgid_r()`, `getgrnam()`, `grp.h`, `limits.h`, `sys/types.h`, Handbuch „POSIX Grundlagen (BS2000/OSD)“.

getgrgid_r - Gruppendateieintrag für eine Gruppen-ID threadsicher ermitteln

Syntax

#include <grp.h>

int getgrgid_r(gid_t *gid*, struct group **grp*, char **buffer*,
size_t *bufsize*, struct group ***result*);

Beschreibung

Die Funktion `getgrgid_r()` aktualisiert die Gruppenstrukur, auf die *grp* zeigt und speichert einen Zeiger auf diese Struktur an der Adresse, auf die *result* zeigt, ab. Die Struktur enthält den Eintrag aus der Gruppendatei, dessen Komponente `gr_gid` mit *gid* übereinstimmt. Die gefundene Gruppenstruktur aus der Gruppendatei wird in den Speicher, der mit dem Parameter *buffer* in der Länge *bufsize* übergeben wurde, kopiert.
Die maximal für diesen Puffer benötigte Größe kann über den `sysconf()`-Parameter `{_SC_GETGR_R_SIZE_MAX}` ermittelt werden. Im Fehlerfall oder wenn der gesuchte Eintrag nicht gefunden werden konnte, wird ein Nullzeiger im Datenbereich, auf den *result* zeigt, zurückgegeben.

Returnwert

0 bei Erfolg.

Fehlernummer sonst. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler

Die Funktion `getgrgid_r()` schlägt fehl, wenn gilt:

`ERANGE` Der über *buffer* und *bufsize* zur Verfügung gestellte Speicher reicht für die Aufnahme der Daten, auf die die resultierende Gruppenstruktur verweist, nicht aus.

Hinweis

Anwendungen, bei denen eine Überprüfung auf Fehlersituationen vorgesehen ist, müssen `errno` auf 0 setzen, bevor `getgrgid_r()` aufgerufen wird. Ist `errno` bei der Rückkehr auf einen Wert ungleich null gesetzt, tritt ein Fehler auf.

Siehe auch

`getgrgid()`, `getgrnam()`, `grp.h`, `limits.h`, `sys/types.h`.

getgrnam - Gruppendateieintrag für Gruppenname ermitteln

Syntax

```
#include <grp.h>

Optional
#include <sys/types.h> ❑

struct group *getgrnam(const char *name);
```

Beschreibung

Die Funktion `getgrnam()` durchsucht die Gruppendatei nach einem Eintrag, dessen Komponente *gr_name* mit *name* übereinstimmt (siehe auch `grp.h` und Handbuch „POSIX-Grundlagen (BS2000/OSD)“.

`getgrnam()` ist nicht threadsicher. Verwenden Sie bei Bedarf die reentrante Funktion `getgrnam_r()`.

Returnwert

Zeiger auf ein Objekt der Struktur `group` (siehe `grp.h`) bei Erfolg.

Nullzeiger — wenn ein Fehler auftritt oder kein Eintrag gefunden wurde, dessen Komponente *gr_gid* mit *gid* übereinstimmt. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler

Die Funktion `getgrnam()` schlägt fehl, wenn gilt:

`EIO`	Ein Ein-/Ausgabefehler ist aufgetreten.
`EINTR`	Während des Ablaufs der Funktion `getgrnam()` wurde ein Signal abgefangen.
`EMFILE`	Für den aktuellen Prozess sind derzeit zu viele Dateideskriptoren offen.
`ENFILE`	Die Dateitabelle des Systems ist derzeit voll.

Hinweis

Der Returnwert kann auf einen statischen Bereich zeigen, der durch einen späteren `getgrid`- oder `getgrnam`-Aufruf überschrieben werden kann.

Wenn Fehlersituationen untersucht werden sollen, muss `errno` vor dem Aufruf von `getgrnam()` auf 0 gesetzt werden.

Siehe auch

`getgrnam_r()`, `getgrgid()`, `grp.h`, `limits.h`, `sys/types.h`, Handbuch „POSIX-Grundlagen (BS2000/OSD)“.

getgrnam_r - Gruppendateieintrag für Gruppenname threadsicher ermitteln

Syntax

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

#include <grp.h>

int getgrnam_r(const char * name, struct group * grp, char * buffer,
               size_t bufsize, struct group ** result);
```

Beschreibung

Die Funktion `getgrnam_r()` aktualisiert die Gruppenstruktur, auf die *grp* zeigt und speichert einen Zeiger auf diese Struktur an der Adresse, auf die *result* zeigt, ab. Die Struktur enthält den Eintrag aus der Gruppendatei, dessen Komponente `gr_name` mit *name* übereinstimmt.

Die gefundene Gruppenstruktur aus der Gruppendatei wird in den Speicher, der mit dem Parameter *buffer* in der Länge *bufsize* übergeben wurde, kopiert. Die maximal für diesen Puffer benötigte Größe kann über den `sysconf()`-Parameter `{_SC_GETGR_R_SIZE_MAX}` ermittelt werden.

Im Fehlerfall oder wenn der gesuchte Eintrag nicht gefunden werden konnte, wird ein Nullzeiger im Datenbereich, auf den *result* zeigt, zurückgegeben.

Returnwert

0 bei Erfolg.

Fehlernummer sonst. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler

Die Funktion `getgrnam_r()` schlägt fehl, wenn gilt:

`ERANGE` Der Speicherplatz, der über *buffer* und *bufsize* bereitgestellt wurde, ist zu klein, um die Daten der resultierenden Gruppenstruktur aufzunehmen.

Siehe auch

`getgrnam()`, `getgrgid_r()`, `grp.h`, `limits.h`, `sys/types.h`,
Handbuch „POSIX Grundlagen (BS2000/OSD)“.

getgroups - zusätzliche Gruppennummern ermitteln

Syntax

```
#include <unistd.h>

Optional
#include <sys/types.h> ❑

int getgroups(int gidsetsize, gid_t grouplist[ ]);
```

Beschreibung

`getgroups()` ermittelt die aktuellen zusätzlichen Gruppennummern des aufrufenden Prozesses und speichert das Ergebnis im Vektor *grouplist*.

gidsetsize legt die Anzahl der Vektorelemente von *grouplist* fest. *gidsetsize* muss groß genug sein, um die komplette Liste aufzunehmen. Diese Liste kann nicht größer als `{NGROUPS_MAX}` sein. Die tatsächliche Anzahl der im Vektor gespeicherten Gruppennummern wird zurückgegeben. Die Werte der Vektoreinträge mit Indizes größer oder gleich dem Returnwert sind undefiniert.

Wenn *gidsetsize* gleich 0 ist, liefert `getgroups()` die Anzahl der Gruppennummern, zu denen der aufrufende Prozess gehört, ohne dass der Vektor *grouplist* verändert wird.

Returnwert

Anzahl der zusätzlichen Gruppennummern	bei Erfolg. Der Returnwert ist ungleich 0 und kleiner als die Anzahl der Gruppennummern für den aufrufenden Prozess.
-1	bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler

`getgroups()` schlägt fehl, wenn gilt:

`EINVAL`	Der Wert von *gidsetsize* ist ungleich 0 und kleiner als *gr_number* für den aufrufenden Prozess.

Hinweis

Die effektive Gruppennummer des aufrufenden Prozesses ist in *grouplist* enthalten.

Siehe auch

`getegid()`, `getuid()`, `setgid()`, `sys/types.h`, `unistd.h`, Handbuch „POSIX Grundlagen (BS2000/OSD)“.

gethostid - Kennung des aktuellen Rechners abfragen

Syntax #include <unistd.h>

long gethostid(void);

Beschreibung

`gethostid()` gibt eine 32-Bit-Kennung für den aktuellen Rechner aus. Die Kennung wird aus der CPU-Seriennummer (3 Bytes) und aus der VM-ID (1 Byte) gebildet, somit sind mehrere VMs einer Anlage untereinander eindeutig.

Returnwert eindeutige Kennung für den aktuellen Rechner
bei Erfolg.

Siehe auch `random()`, `unistd.h`.

gethostname - Name des aktuellen Rechners abfragen

Syntax #include <unistd.h>

int gethostname(char *_name_, size_t _namelen_);

Beschreibung

`gethostname()` ermittelt den Standardnamen des aktuellen Rechners. Der Parameter *namelen* gibt die Größe des Feldes an, auf das *name* zeigt. Dem Namen wird eine abschließende Null angefügt, sofern *namelen* dafür ausreicht.
Überschreitet der Rechnername den Wert *namelen*, wird der Name abgeschnitten und es ist nicht sichergestellt, dass eine abschließende Null angehängt wird.

Returnwert 0 bei Erfolg.

-1 sonst.

Siehe auch `gethostid()`, `unistd.h`.

getitimer, setitimer - lesen bzw. setzen

Syntax #include <sys/time.h>

int getitimer(int *which*, struct itimerval **value*);

int setitimer(int *which*, const struct itimerval **value*, struct itimerval **ovalue*);

Beschreibung

Das System bietet jedem Prozess drei Intervall-Timer an, die in der Datei `sys/time.h` vereinbart werden. Der Aufruf `getitimer()` speichert den aktuellen Wert des Timers *which* in der Struktur, auf die *value* zeigt. Der Aufruf `setitimer()` setzt den Wert von *which* auf den Wert, der in der Struktur steht, auf die *value* zeigt; ist *ovalue* ungleich `NULL`, wird der vorherige Wert des Timers in der Struktur abgelegt, auf die *ovalue* zeigt.

Die Einstellung eines Timers wird durch die Struktur `itimerval` (siehe `sys/time.h`) definiert, welche mindestens die folgenden Komponenten enthält:

```
struct timeval   it_interval;    /* Uhrintervall */
struct timeval   it_value;       /* aktueller Wert */
```

Wenn `it_value` ungleich null ist, wird die Zeit bis zum nächsten Ablauf des Timers angegeben. Wenn `it_interval` ungleich null ist, wird ein Wert angegeben, auf den `it_value` gesetzt wird, wenn der Timer abläuft. Wird `it_value` auf null gesetzt, so wird der Timer deaktiviert, unabhängig vom Wert von `it_interval`. Das Setzen von `it_interval` auf null deaktiviert den Timer nach seinem nächsten Ablauf (vorausgesetzt, dass `it_value` ungleich null ist).

Sind Zeitwerte kleiner als die Auflösung der Systemuhr, so werden diese auf die Auflösung der Systemuhr gerundet.

Jedem Prozess stehen drei Timer zur Verfügung, die über die folgenden Werte für *which* angesprochen werden:

`ITIMER_REAL` dekrementiert in Echtzeit. Das Signal `SIGALRM` wird gesendet, wenn dieser Timer abläuft.

`ITIMER_VIRTUAL` dekrementiert in der virtuellen Prozesszeit. Dieser Timer läuft nur, wenn der Prozess ausgeführt wird. Das Signal `SIGVTALRM` wird gesendet, wenn dieser Timer abläuft.

`ITIMER_PROF` dekrementiert in virtuelle Prozesszeit, unabhängig von `ITIMER_VIRTUAL`. Jedes Mal, wenn der Timer `ITIMER_PROF` abläuft, wird das Signal `SIGPROF` gesendet. Da dieses Signal Systemaufrufe des Prozesses unterbricht, müssen diejenigen Programme, die diesen Timer verwenden, darauf vorbereitet sein, die unterbrochenen Systemaufrufe zu wiederholen.

`setitimer()` und `sleep()` oder `usleep()` sollten nicht zusammen benutzt werden, da es zu unerwünschten Wechselwirkungen kommen kann, insbesondere meldet ein Aufruf von `sleep()` eine eigene Signalbehandlungsroutine an, so dass die Signalbehandlungsroutine des Anwenders nicht aktiviert wird.

Returnwert

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

Fehler `setitimer()` schlägt fehl, wenn gilt:

`EINVAL`	Die Werte, auf die das Argument *value* zeigt, sind ungültig. (Für die Mikrosekunden muss eine nicht-negative Ganzzahl kleiner 1.000.000 angegeben werden, für die Sekunden eine nicht-negative Ganzzahl.)

`getitimer()` und `setitimer()` schlagen fehl, wenn gilt:

`EINVAL`	Der Parameter *which* wurde nicht erkannt

Hinweis Das Feld mit den Mikrosekunden darf keinen Wert enthalten, der gleich oder größer als eine Sekunde ist.

Siehe auch `alarm()`, `sleep()`, `ualarm()`, `usleep()`, `signal.h`, `sys/time.h`.

getlogin - Benutzerkennung ermitteln

Syntax

```
#include <unistd.h>

Optional
#include <stdlib.h> ❑

char *getlogin(void);
```

Beschreibung

`getlogin()` liefert einen Zeiger auf eine Zeichenkette mit dem Benutzernamen des aufrufenden Prozesses, der der Benutzerkennung des aufrufenden Prozesses entspricht. Wenn `getlogin()` nicht den Nullzeiger zurückliefert, dann zeigt dieser Zeiger auf den Namen, unter dem sich der Benutzer angemeldet hat (Benutzerkennung), selbst wenn es mehrere Benutzernamen mit derselben Benutzernummer gibt.

`getlogin()` ist nicht threadsicher. Verwenden Sie bei Bedarf die reentrante Funktion `getlogin_r()`.

Returnwert

Zeiger auf die Benutzerkennung
: Die Funktion ist immer erfolgreich.

Nullzeiger
: bei Fehler, wenn `getlogin()` z.B. aus einem Prozess heraus aufgerufen wird, dessen Benutzerkennung nicht herausgefunden werden kann. `errno` wird nicht gesetzt.

Hinweis

Das Ergebnis zeigt normalerweise auf statische Daten, deren Inhalt von jedem Aufruf überschrieben werden. Eine portable Anwendung sollte daher die Benutzerkennung umspeichern, wenn dieser über einen weiteren Aufruf der Funktion hinaus benötigt wird.

Drei zum aktuellen Prozess gehörende Namen können bestimmt werden: `getpwuid(geteuid())` liefert den Namen, der der effektiven Benutzernummer des Prozesses zugeordnet ist; `getlogin()` liefert den Namen, der den aktuellen Anmeldungs-Aktivitäten zugeordnet ist und `getpwuid (getuid())` liefert den Namen, der zur realen Benutzernummer des Prozesses gehört.

Siehe auch

`getlogin_r()`, `getpwnam()`, `getpwuid()`, `geteuid()`, `getuid()`, `limits.h`, `unistd.h`.

getlogin_r - Benutzerkennung threadsicher ermitteln

Syntax

```
#include <unistd.h>

int getlogin_r(char * name, size_t namesize);
```

Beschreibung

Die Funktion `getlogin_r()` schreibt den Benutzernamen des aufrufenden Prozesses, der der Benutzerkennung des aufrufenden Prozesses entspricht, in den vom Aufrufer bereitgestellten Datenbereich, auf den *name* zeigt. Der Datenbereich ist *namesize* Zeichen lang und sollte genug Platz bieten für den Namen und das abschließende Nullzeichen. Die maximale Größe des Login-Namens ist `{LOGIN_NAME_MAX}`.

Wenn `getlogin_r()` erfolgreich ist, zeigt *name* auf den Namen, den der Benutzer bei den aktuellen Anmeldungs-Aktivitäten verwendet hat, auch wenn es mehrere Namen mit derselben Benutzerkennung gibt.

Returnwert

0 bei Erfolg.

Fehlernummer sonst.

Fehler

Die Funktion `getlogin_r()` schlägt fehl, wenn gilt:

`ERANGE` Der Wert von *namesize* ist kleiner als die Länge des ermittelten Benutzernamens einschließlich des abschließenden Nullzeichens.

Siehe auch `getlogin()`, `getpwnam_r()`, `getpwuid_r()`.

getmsg - Nachricht von einer STREAMS-Datei lesen

Syntax

#include <stropts.h>

int getmsg(int *fildes*, struct strbuf **ctlptr*, struct strbuf **dataptr*, int **flagsp*);

int getpmsg(int *fildes*, struct strbuf **ctlptr*, struct strbuf **dataptr*, int **bandp*, int **flagsp*);

Beschreibung

`getmsg()` holt den Inhalt einer Nachricht, die in der Lese-Queue des Stream-Kopfs einer STREAMS-Datei steht, und schreibt den Inhalt in einen vom Benutzer angegebenen Puffer. Die Nachricht enthält entweder einen Datenteil, einen Steuerteil oder beide Teile. Der Daten- und der Steuerteil der Nachricht werden, wie nachstehend beschrieben, in separate Puffer geschrieben. Die Semantik der Teile wird durch das STREAMS-Modul definiert, das die Nachricht generiert hat.

Die Funktion `getpmsg()` führt das Gleiche aus wie `getmsg()`, aber sie liefert eine genauere Kontrolle über die Priorität der erhaltenen Meldungen. Außer, wenn es speziell vermerkt wurde, gelten alle Informationen, die `getmsg()` betreffen, auch für `getpmsg()`.

fildes gibt einen Dateideskriptor an, der auf einen offenen Stream zeigt.

ctlptr und *dataptr* verweisen je auf eine `strbuf`-Struktur, die nachstehende Elemente aufweist:

```
int maxlen;   /* Maximum Puffergröße */
int len;      /* Länge der Daten */
char *buf;    /* Zeiger auf den Puffer */
```

`buf` weist auf einen Puffer, in den die Daten bzw. Steuerinformationen geschrieben werden sollen. `maxlen` zeigt die größtmögliche Anzahl Bytes an, die dieser Puffer aufnehmen kann. Bei der Rückgabe enthält `len` die Byte-Anzahl der tatsächlich empfangenen Daten bzw. Steuerinformationen, oder der Wert ist 0, wenn der Steuer- oder Datenteil eine Nulllänge aufweist, oder der Wert ist -1, wenn die Nachricht keine Daten- oder Steuerinformationen enthält.

Wenn `getmsg()` aufgerufen wird, sollte *flagsp* auf eine Ganzzahl verweisen, welche die Art der Nachricht, die der Benutzer erhalten kann, anzeigt. Dieses wird später beschrieben.

ctlptr wird zur Aufnahme des Steuerteils der Nachricht und *dataptr* zur Aufnahme des Datenteils der Nachricht verwendet. Wenn *ctlptr* (oder *dataptr*) `NULL` ist oder das `maxlen`-Feld -1 ist, wird der Steuer- (bzw. Daten-)teil der Nachricht nicht verarbeitet und bleibt in der Lese-Queue des Stream-Kopfes. Wenn *ctlptr* (oder *dataptr*) nicht `NULL` ist und es keinen korrespondierenden Steuer- (oder Daten-)teil der Nachricht in der Lese-Queue des Stream-Kopfes gibt, wird `len` auf -1 gesetzt. Wenn das `maxlen`-Feld auf 0 gesetzt ist und ein Steuer- (oder Daten-)teil mit einer Nulllänge vorliegt, wird dieser Nulllängenteil aus der Lese-Queue entfernt und `len` auf 0 gesetzt. Wenn das `maxlen`-Feld auf 0 gesetzt ist und mehr als 0 Byte Steuer- (oder Daten-)Informationen vorhanden sind, bleiben diese Informa-

tionen in der Lese-Queue, und `len` wird auf 0 gesetzt. Wenn das `maxlen`-Feld in *ctlptr* bzw. *dataptr* kleiner als der Steuer- oder Datenteil der Nachricht ist, werden `maxlen` Bytes geholt. In diesem Fall wird der Rest der Nachricht in der Lese-Queue des Stream-Kopfes gelassen und ein Rückgabewert von ungleich null geliefert (siehe Returnwert).

Standardmäßig verarbeitet `getmsg()` die erste Meldung, die in der Lese-Queue zur Verfügung steht. Wenn die Ganzzahl, auf die *flagsp* zeigt, auf `RS_HIPRI` gesetzt ist, empfängt der Prozess nur Meldungen hoher Priorität. In diesem Fall verarbeitet `getmsg()` die nächste Nachricht nur, wenn diese eine Nachricht hoher Priorität ist. Wenn die Ganzzahl, auf die durch *flagsp* verwiesen wird, 0 ist, bringt `getmsg()` jede verfügbare Nachricht in der Lese-Queue des Stream-Kopfes. In diesem Fall wird bei Rückkehr die Ganzzahl, auf die durch *flagsp* verwiesen wird, auf `RS_HIPRI` gesetzt, wenn eine Nachricht hoher Priorität angetroffen wurde, andernfalls auf 0.

Für `getpmsg()` gibt es andere Optionen als für `getmsg()`. *flagsp* verweist auf eine Bitmaske mit den folgenden Optionen, die sich gegenseitig ausschließen: `MSG_HIPRI`, `MSG_BAND` und `MSG_ANY`. Ebenso wie `getmsg()` verarbeitet `getpmsg()` die als nächste zur Verfügung stehende Nachricht in der Lese-Queue des Stream-Kopfes. Wiederum kann der Benutzer wählen, nur Nachrichten hoher Priorität zu erhalten, indem er die Ganzzahl, auf die mit *flagsp* verwiesen wird, auf `MSG_HIPRI` setzt und diejenige, auf die *bandp* verweist, auf 0. In diesem Fall verarbeitet `getpmsg()` nur dann die nächste Nachricht, wenn es eine Nachricht hoher Priorität ist. In ähnlicher Weise kann der Benutzer eine Nachricht aus einem speziellen Prioritätsbereich aufrufen, indem er die Ganzzahl, auf die durch *flagsp* verwiesen wird, auf `MSG_BAND` setzt, und die Ganzzahl, auf die durch *bandp* verwiesen wird, auf den gewünschten Prioritätsbereich setzt. In diesem Fall verarbeitet `getpmsg()` nur dann die nächste Nachricht, wenn sie sich in einem Prioritätsbereich befindet, welcher gleich oder größer als die Ganzzahl ist, auf welche durch *bandp* verwiesen wird, oder wenn es sich um eine Nachricht hoher Priorität handelt. Wenn ein Benutzer lediglich die erste Meldung der Queue abrufen möchte, sollte die Ganzzahl, auf welche durch *flagsp* verwiesen wird, auf `MSG_ANY` gesetzt sein, und die Ganzzahl, auf welche durch *bandp* verwiesen wird, sollte auf 0 gesetzt sein. Falls die erhaltene Nachricht eine Nachricht hoher Priorität war, ist bei der Rückkehr die Ganzzahl, auf welche durch *flagsp* verwiesen wird, auf `MSG_HIPRI`, und die Ganzzahl, auf welche durch *bandp* verwiesen wird, auf 0 gesetzt. Bei allen anderen Nachrichten ist die Ganzzahl, auf die *flagsp* zeigt, auf `MSG_BAND` gesetzt, und die Ganzzahl, auf die *bandp* zeigt, ist auf den Prioritätsbereich der Nachricht gesetzt.

Wenn `O_NDELAY` und `O_NONBLOCK` nicht gesetzt wurde, blockieren `getmsg()` und `getpmsg()`, bis eine Nachricht des mit *flagsp* angegebenen Typs in der Lese-Queue des Stream-Kopfes vorhanden ist. Wenn `O_NDELAY` oder `O_NONBLOCK` gesetzt wurde und keine Nachricht des angegebenen Typs in der Lese-Queue vorhanden ist, bleiben `getmsg()` und `getpmsg()` erfolglos, und `errno` wird auf `EAGAIN` gesetzt.

Wenn auf dem Stream, aus dem die Nachrichten geholt werden sollen, ein Verbindungsabbruch auftritt, arbeiten `getmsg()` und `getpmsg()` normal weiter, wie oben beschrieben, bis die Lese-Queue entleert ist. Danach wird 0 in den `len`-Feldern von *ctlptr* und *dataptr* zurückgegeben.

Wird eine Nachricht mit einem `getmsg()`- bzw. `getpmsg()`-Aufruf nicht vollständig gelesen, so kann der Rest der Nachricht mit anschließenden `getmsg()`- bzw. `getpmsg()`-Aufrufen geholt werden. Wenn jedoch eine Nachricht hoher Priorität in dem Stream-Kopf der Lese-Queue eingetroffen ist, bearbeitet der nächste `getmsg()`- bzw. `getpmsg()`-Aufruf die Nachricht hoher Priorität vorrangig, bevor der Rest der vorher empfangenen Teilnachricht bearbeitet wird.

Returnwert Nicht negativer Wert
bei Erfolg.

0 wenn eine vollständige Nachricht erfolgreich gelesen wurde.

`MORECTL` zeigt an, dass weitere Steuerinformationen auf einen Abruf warten.

`MOREDATA` zeigt an, dass weitere Daten auf den Abruf warten.

bitweises ODER von `MORECTL` und `MOREDATA`
zeigt an, dass noch beide Arten übrig sind.

Fehler `getmsg()` oder `getpmsg()` schlagen fehl, wenn gilt:

`EAGAIN` `O_NDELAY` oder `O_NONBLOCK` ist gesetzt, und es stehen keine Nachrichten zur Verfügung.

`EBADF` *fildes* ist kein zum Lesen offener, gültiger Dateideskriptor.

`EBADMSG` Die zu lesende Nachricht in der Queue ist für `getmsg()` bzw. `getpmsg()` nicht gültig.

`EINTR` Ein Signal wurde während des Systemaufrufs `getmsg()` bzw. `getpmsg()` abgefangen.

`EINVAL` Ein ungültiger Wert wurde in *flagsp* angegeben, oder der durch *fildes* angegebene Stream oder Multiplexer ist direkt oder indirekt stream-abwärts mit einem Multiplexer verbunden.

`ENOSTR` Dem Dateideskriptor *fildes* ist kein Stream zugeordnet.

`getmsg()` und `getpmsg()` kann auch dann erfolglos sein, wenn vor dem Aufruf von `getmsg()` eine STREAMS-Fehlermeldung am Stream-Kopf empfangen wurde. In diesem Fall zeigt `errno` den zuvor aufgetretenen STREAMS-Fehler an.

Siehe auch `poll()`, `putmsg()`, `read()`, `write()`, `stropts.h`.

getopt, optarg, optind, opterr, optopt - Kommandooptionen syntaktisch analysieren

Syntax

```
#include <unistd.h>

int getopt(int argc, char * const argv[ ], const char *optstring);
extern char *optarg;
extern int optind, opterr, optopt;
```

Beschreibung

`getopt()` ist ein Parser für die Kommandozeile, der für Anwendungen benutzt werden kann, die sich an die XPG4-Kommandoeingabe-Konventionen halten (siehe Handbuch „POSIX Kommandos (BS2000/OSD)“). Für darüber hinausgehende Richtlinien ist die Anwendung verantwortlich.

`getopt()` gibt das nächste Optionszeichen in *argv* zurück, das einem Zeichen in *optstring* entspricht.

argc ist der Argumentzähler, wie er an `main()` übergeben wird (siehe `exec`).

argv zeigt auf einen Vektor von *argc*+1 Elementen, der *argc* Zeiger auf Zeichenketten, gefolgt vom Nullzeiger, enthält. Er enthält die Optionsnamen, wie sie an `main()` übergeben werden (siehe `exec`).

optstring ist eine Zeichenkette aus zulässigen Optionszeichen (siehe Handbuch „POSIX Kommandos (BS2000/OSD)“). Folgt in dieser Zeichenkette auf ein Zeichen ein Doppelpunkt (:), wird erwartet, dass die Option ein oder mehrere Argumente hat.

`optind` ist eine externe Variable, die den Index für das nächste Element des Vektors *argv*`[]` repräsentiert, das ausgewertet werden soll. Sie wird vom System auf 1 initialisiert. `getopt()` aktualisiert `optind()` nach der Auswertung jedes Elements von *argv*`[]`. Wenn ein Element von *argv*`[]` mehrere Optionszeichen enthält, ist nicht festgelegt, wie `getopt()` bestimmt, welche Optionen schon ausgewertet wurden.

`optarg` ist eine externe Variable, die von `getopt()` gesetzt wird, wenn eine Option ein Argument hat. Dies geschieht, wie folgt:

1. Wenn die betreffende Option das letzte Zeichen in der Zeichenkette ist, auf die ein Element aus *argv* zeigt, zeigt `optarg` auf das nächste Element aus *argv* und `optind` wird um 2 erhöht. Wenn der Wert von `optind` nicht kleiner ist als *argc*, fehlt ein Optionsargument und `getopt()` meldet einen Fehler.
2. Ansonsten wird `optarg` so gesetzt, dass es auf die Zeichenkette zeigt, die dem Optionszeichen folgt. Dann wird `optind` um 1 erhöht.

`opterr` ist eine externe Variable, die im Fehlerfall die Ausgabe einer Fehlermeldung steuert. Wenn `opterr` gleich 0 gesetzt wird, wird die Ausgabe einer Fehlermeldung unterdrückt.

`optopt` ist eine externe Variable, die das Optionszeichen enthält, durch das `getopt()` nicht erfolgreich beendet werden konnte.

Returnwert nächstes Optionszeichen aus der Kommandozeile
bei erfolgreicher Beendigung.

: wenn ein Optionsargument fehlt und das erste Zeichen in *optstring* ein Doppelpunkt (:) ist. `getopt()` setzt dann die Variable `optopt` auf das Optionszeichen, das den Fehler verursacht hat.

? wenn ein Optionszeichen gefunden wird, das nicht in *optstring* enthalten ist, oder wenn ein Optionsargument fehlt und das erste Zeichen in *optstring* kein Doppelpunkt ist.
`getopt()` setzt in diesen Fällen die Variable `optopt` auf das Optionszeichen, das den Fehler verursacht hat. Wenn `opterr` von der Anwendung nicht auf 0 gesetzt wurde, gibt `getopt()` eine Fehlermeldung auf `stderr` aus, und zwar in dem Format, das für das `getopts`-Kommando vereinbart ist (siehe Handbuch „POSIX Kommandos (BS2000/OSD)“).

-1 wenn *argv*`[optind]` ein Nullzeiger ist
oder wenn **argv*`[optind]` ungleich dem Zeichen "–" ist
oder wenn *argv*`[optind]` auf die Zeichenkette "–" zeigt;
`optind` wird in diesen Fällen nicht verändert.

-1 wenn *argv*`[optind]` auf die Zeichenkette "– –" zeigt. `optind` wird in diesem Fall erhöht.

Hinweis `getopt()` überprüft nicht vollständig auf notwendige Argumente. Wenn z.B. eine Optionszeichenkette `a:b` und die Eingabe `–a –b` gegeben sind, nimmt `getopt()` an, dass `–b` das notwendige Argument für die Option `–a` ist, und nicht, dass ein notwendiges Argument für `–a` fehlt.

Mehrere Optionen dürfen nicht zusammengefasst werden, wenn die letzte Option ein Argument benötigt. Wenn `a` und `b` normale Optionen sind und die Option `o` das Argument *xxx* benötigt, sollte nicht *cmd* `–abo` *xxx* angegeben werden, sondern *cmd* `–ab –o` *xxx* . Die zusammengeschriebene Form wird zwar von der aktuellen Implementierung unterstützt, aber eventuell in zukünftigen nicht mehr.

BS2000
Wenn ein Programm in der BS2000-Umgebung gestartet wird, werden die Programmparameter, wie bisher bei C-Programmen, versorgt (siehe C- und C++-Benutzerhandbücher). ❑

Wenn der von `getopt()` zurückgegebene ganzzahlige Wert in einer Variablen vom Typ `char` gespeichert und mit der ganzzahligen Konstanten `EOF` verglichen wird, ist dieser Vergleich nie erfolgreich, weil beim Übergang von `char` zu `int` keine Vorzeichenpropagierung stattfindet.

Siehe auch `exec`, `unistd.h`, Kommando `getopts` (siehe Handbuch „POSIX Kommandos (BS2000/OSD)“).

getpagesize - aktuelle Seitengröße ausgeben

Syntax

```
#include <unistd.h>

int getpagesize(void);
```

Beschreibung

`getpagesize()` gibt die Anzahl Byte auf einer Speicherseite zurück.

Ein Aufruf von `getpagesize()` entspricht dem von `sysconf(_SC_PAGE_SIZE)` und von `sysconf(_SC_PAGESIZE)`.

`getpagesize()` ist nicht threadsicher.

Returnwert Aktuelle Seitengröße

Die Funktion ist immer erfolgreich.

Hinweis Die von `getpagesize()` zurückgelieferte Seitengröße muss nicht mit der Größe der hardwaremäßig eingeteilten Speicherseiten übereinstimmen.
Unter POSIX entspricht diese Größe allerdings der hardwaremäßig eingestellten.

Diese Seitengröße muss weder der Mindestgröße entsprechen, die mit `malloc()` angefordert werden kann, noch darf sich eine Anwendung darauf verlassen, dass ein Objekt dieser Größe mit `malloc()` allokiert werden kann.

Siehe auch `brk()`, `getrlimit()`, `mmap()`, `mprotect()`, `munmap()`, `msync()`, `sysconf()`, `unistd.h`.

getpass - Zeichenkette ohne Echo lesen

Syntax #include <unistd.h>

char *getpass(const char **prompt*);

Beschreibung

`getpass()` führt folgende Aktionen aus:

- öffnet das Terminal, das den Prozess steuert
- schreibt die mit dem Nullbyte abgeschlossene Zeichenkette *prompt* auf dieses Gerät
- schaltet das lokale Echo ab
- liest eine Zeichenkette bis zum nächsten Zeilenendezeichen oder bis `EOF` ein
- stellt den ursprünglichen Zustand des Terminals wieder her
- schließt die Gerätedatei für das Terminal

Returnwert Zeiger auf eine mit dem Nullbyte abgeschlossene Zeichenkette bei erfolgreicher Beendigung. Der Returnwert besteht aus höchstens `{PASS_MAX}` vom Terminal eingelesenen Bytes.

Nullzeiger bei Fehler. Der ursprüngliche Zustand des Terminals wird wiederhergestellt. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `getpass()` schlägt fehl, wenn gilt:

`EINTR` `getpass()` wurde von einem Signal unterbrochen.

`EIO` Der Prozess ist Mitglied einer Hintergrund-Prozessgruppe, die von ihrem steuernden Terminal zu lesen versucht. Dabei ignoriert oder blockiert der Prozess das Signal `SIGTTIN` oder die Prozessgruppe ist verwaist.

`EMFILE` `{OPEN_MAX}`-Dateideskriptoren sind im Augenblick im aufrufenden Prozess offen.

`ENFILE` Das Maximum der erlaubten Dateianzahl ist im Augenblick im System offen.

`ENXIO` Der Prozess besitzt kein steuerndes Terminal.

Hinweis Der Returnwert zeigt auf statische Daten, deren Inhalt bei jedem Aufruf überschrieben wird. `pclose()` wird nur für POSIX-Dateien ausgeführt.

`getpass()` ist nicht threadsicher. Wird zukünftig vom X/Open-Standard nicht mehr unterstützt.

Siehe auch `limits.h`, `unistd.h`.

getpgid - Prozessgruppennummer lesen

Syntax #include <unistd.h>

pid_t getpgid(pid_t *pid*);

Beschreibung

`getpgid()` gibt die Prozessgruppennummer des Prozesses zurück, dessen Prozessnummer gleich *pid* ist. Falls *pid* 0 ist, wird die Prozessgruppennummer des aufrufenden Prozesses zurückgeliefert.

Returnwert Prozessgruppennummer
bei Erfolg.

`(pid_t)-1` bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `getpgid()` schlägt fehl, wenn gilt:

`EPERM` Der Prozess, dessen Prozessnummer gleich *pid* ist, befindet sich nicht in der gleichen Sitzung wie der aufrufende Prozess, und die Implementierung erlaubt keinen Zugriff auf die Prozessgruppennummer dieses Prozesses vom aufrufenden Prozess aus.

`ESRCH` Es gibt keinen Prozess mit einer Prozessnummer *pid*.

`EINVAL` Der Wert von *pid* ist ungültig.

Siehe auch `exec`, `fork()`, `getpgrp()`, `getpid()`, `getsid()`, `setpgid()`, `setsid()`, `unistd.h`.

getpgmname - Programmnamen ermitteln *(BS2000)*

Syntax #include <stdlib.h>

char *getpgmname(void);

Beschreibung

`getpgmname()` liefert den Namen des aufrufenden Programmes. Das Ergebnis entspricht `argv[0]` der Funktion `main`.

Returnwert Zeiger auf den Programmnamen
Die Funktion ist immer erfolgreich.

getpgrp - Prozessgruppennummer ermitteln

Syntax #include <unistd.h>

Optional
#include <sys/types.h> ❑

pid_t getpgrp(void);

Beschreibung
`getpgrp()` gibt die Gruppennummer des aufrufenden Prozesses zurück.

Returnwert Prozessgruppennummer
Die Funktion ist immer erfolgreich.

Siehe auch `exec`, `fork()`, `getpid()`, `getppid()`, `kill()`, `setpgid()`, `setsid()`,`sys/types.h`, `unistd.h`, Handbuch „POSIX Grundlagen (BS2000/OSD)“.

getpid - Prozessnummer ermitteln

Syntax #include <unistd.h>

Optional
#include <sys/types.h> ❑

pid_t getpid(void);

Beschreibung
`getpid()` liefert die Prozessnummer des aufrufenden Prozesses.

Returnwert Prozessnummer des aufrufenden Prozesses
Die Funktion ist immer erfolgreich.

Siehe auch `exec`, `fork()`, `getpgrp()`, `getppid()`, `kill()`, `setpgid()`, `setsid()`, `sys/types.h`, `unistd.h`.

getpmsg - Nachricht von einer STREAMS-Datei lesen

#include <pwd.h>

int getpmsg(int *fildes*, struct strbuf **ctlptr*, struct strbuf **dataptr*, int **bandp*, int **flagsp*);

Beschreibung

Siehe `getmsg()`.

getppid - Vaterprozessnummer ermitteln

Syntax #include <unistd.h>

Optional
#include <sys/types.h> ❑

pid_t getppid(void);

Beschreibung

`getppid()` liefert die Vaterprozessnummer des aufrufenden Prozesses.

Returnwert Vaterprozessnummer des aufrufenden Prozesses
Die Funktion ist immer erfolgreich.

Siehe auch `exec`, `fork()`, `getpgrp()`, `getpid()`, `kill()`, `setpgid()`, `setsid()`, `sys/types.h`, `unistd.h`.

getpriority, setpriority - Prozesspriorität abrufen bzw. setzen

Syntax

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

int getpriority(int which, id_t who);

int setpriority(int which, id_t who, int priority);
```

Beschreibung

`getpriority()` ruft die aktuelle Scheduling-Priorität des Prozesses, der Prozessgruppe oder des Benutzers ab.

`setpriority()` setzt die Scheduling-Priorität des Prozesses, der Prozessgruppe oder des Benutzers.

Die Argumente *which* und *who* legen fest, welcher Prozess angesprochen wird. *which* kann die folgenden Werte annehmen: `PRIO_PROCESS`, `PRIO_PGRP` oder `PRIO_USER`. Davon abhängig wird der Inhalt von *who* interpretiert als Prozess-ID, Prozessgruppen-ID oder Benutzer-ID. Ein Nullwert für *who* bezeichnet den aktuellen Prozess, die aktuelle Prozessgruppe oder den aktuellen Benutzer.

`getpriority()` gibt die höchste Priorität (den niedrigsten numerischen Wert) zurück, die von einem der angegebenen Prozesse in Anspruch genommen wird. `setpriority()` setzt die Prioritäten aller angegebenen Prozesse auf den durch *priority* angegebenen Wert.

Die Standardpriorität ist 0; niedrigere Prioritäten bedeuten ein verbessertes Scheduling. Wenn prio unter -20 liegt, wird der Wert -20 verwendet; liegt er über 20, wird der Wert 20 verwendet.

Nur entsprechend berechtigte Benutzer können Prioritäten vermindern.

Bei der Verwendung von Threads wirken sich die Funktionen `getpriority()` und `setpriority()` bezüglich Wirkung auf den Prozess oder auf einen Thread:

– Abfragen bzw. setzen der Scheduling-Priorität des Prozesses.

– Wenn der Prozess ”multithreaded” ist, wirkt sich die Scheduling-Priorität auf alle Threads des Prozesses aus.

Returnwert `getpriority()`:

-20 ≤ Returnwert ≤ 20
: bei Erfolg.

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

`setpriority()`:

0
: bei Erfolg.

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

`getpriority()` und `setpriority()` schlagen fehl, wenn gilt:

`ESRCH` Es wurde kein Prozess gefunden, auf den die angegebenen Werte *which* und *who* zutreffen.

`EINVAL` *which* war weder `PRIO_PROCESS`, `PRIO_PGRP` noch `PRIO_USER`, oder *who* enthielt keine gültige Prozess-ID, Prozessgruppen-ID oder Benutzer-ID.

Zusätzlich kann `setpriority()` fehlschlagen, wenn gilt:

`EPERM` Es wurde ein Prozess gefunden, aber weder die effektive noch die reale Benutzer-ID stimmt mit der effektiven Benutzer-ID des Prozesses überein, dessen Priorität geändert werden soll.

`EACCES` Es wurde versucht, die Priorität auf einen kleineren Wert zu setzen, was einer höheren Priorität entspricht, aber der aktuelle Prozess hat nicht die entsprechende Berechtigung.

Hinweis Wie sich das Ändern der Scheduling-Priorität auswirkt, hängt von dem Algorithmus des Prozess-Scheduling ab.

Da `getpriority()` legitimerweise auch den Wert -1 zurückgeben kann, muss die externe Variable `errno` vor dem Aufruf gelöscht und anschließend geprüft werden, um festzustellen, ob es sich bei dem Wert -1 um einen Fehler oder einen zulässigen Wert handelt.

Siehe auch `nice()`, `sys/resource.h`.

getpwent - Benutzerdaten aus dem Benutzerkatalog lesen

Syntax #include <pwd.h>

struct passwd *getpwent(void);

Beschreibung

`Siehe endpwent().`

getpwnam - Benutzername ermitteln

Syntax

#include <pwd.h>

Optional
#include <sys/types.h> ❑

struct passwd *getpwnam(const char **name*);

Beschreibung

`getpwnam()` durchsucht den Benutzerkatalog nach einem Eintrag, dessen Komponente `pw_name` mit *name* übereinstimmt (siehe auch `pwd.h` und Handbuch „POSIX Grundlagen (BS2000/OSD)“).

`getpwnam()` ist nicht threadsicher. Verwenden Sie bei Bedarf die reentrante Funktion `getpwnam_r()`.

Returnwert

Zeiger auf eine Struktur vom Typ `passwd` (siehe `pwd.h`) bei Erfolg.

Nullzeiger — wenn beim Lesen ein Fehler auftritt oder kein passender Eintrag gefunden wurde. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler

`getpwnam()` schlägt fehl, wenn gilt:

`EINVAL` — *name* ist zu lang.

`EFAULT` — Fehler beim Anlegen der `passwd`-Struktur oder fehlerhafte Zeichenkette *name*.

`ENOENT` — Benutzer ist unbekannt.

Hinweis

Der Returnwert kann auf einen statischen Bereich zeigen, der durch einen späteren `cuserid`-, `getpwnam`- oder `getpwuid`-Aufruf überschrieben werden kann.

Wenn Fehlersituationen untersucht werden sollen, muss `errno` vor dem Aufruf von `getpwnam()` auf 0 gesetzt werden. Wenn der Fehler-Returnwert ungleich 0 ist, ist ein Fehler aufgetreten.

Die drei Namen eines aktuellen Prozesses können wie folgt festgestellt werden: `getpwuid(geteuid())` gibt u. a. den Namen zurück, der mit der effektiven Benutzernummer des Prozesses verbunden ist, `getlogin()` gibt die Benutzerkennung der aktuellen Login-Aktivität zurück, und `getpwuid(getuid())` gibt u.a. den Namen zurück, der mit der realen Benutzernummer des Prozesses verbunden ist.

Siehe auch

`geteuid()`, `getlogin()`, `getpwnam_r()`, `getpwuid()`, `getuid()`, `limits.h`, `pwd.h`, `sys/types.h`, Handbuch „POSIX Grundlagen (BS2000/OSD)“.

getpwnam_r - Benutzernamen threadsicher ermitteln

Syntax

```
#include <sys/types.h>
#include <pwd.h>

int getpwnam_r(const char *nam, struct passwd *pwd, char *buffer,
               size_t bufsize, struct passwd **result);
```

Beschreibung

Die Funktionen `getpwnam_r()` und `getpwuid_r()` aktualisieren die `passwd`-Strukur, auf die *pwd* zeigt und speichern einen Zeiger auf diese Struktur an der Adresse, auf die *result* zeigt, ab. Die Struktur enthält den Eintrag aus dem Benutzerkatalog, dessen Komponente `pw_name` bzw. `pw_uid` mit *nam* bzw. *uid* übereinstimmt.

Die gefundene `passwd`-Struktur aus dem Benutzerkatalog wird in den Speicher kopiert, der mit dem Parameter *buffer* in der Länge *bufsize* übergeben wurde. Die maximal für diesen Puffer benötigte Größe kann über den `sysconf()`-Parameter `{_SC_GETPW_R_SIZE_MAX}` ermittelt werden.

Returnwert

0 bei Erfolg.

Fehlernummer sonst. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler

Die Funktionen `getpwnam_r()` und `getpwuid_r()` schlagen fehl, wenn gilt:

`ERANGE` Der über *buffer* und *bufsize* zur Verfügung gestellte Speicher reicht für die Aufnahme der Daten, auf die die resultierende Gruppenstruktur verweist, nicht aus.

Hinweis

Bei einem Fehler, oder wenn der angeforderte Eintrag nicht gefunden wird, wird ein Nullzeiger an der Adresse zurückgegeben, auf die *result* zeigt.

Siehe auch `getpwnam(), getpwuid(), pwd(), types()`.

getpwuid - Benutzernummer ermitteln

Syntax

```
#include <pwd.h>

Optional
#include <sys/types.h> ❑

struct passwd *getpwuid(uid_t uid);
```

Beschreibung

`getpwuid()` durchsucht den Benutzerkatalog nach einem Eintrag, dessen Komponente *pw_uid* (siehe Struktur `passwd` in `pwd.h`) mit *uid* übereinstimmt. Nachfolgende Strukturen mit derselben Benutzernummer werden nicht gefunden.

Returnwert

Zeiger auf eine Struktur vom Typ `passwd` (siehe `pwd.h`) bei Erfolg.

Nullzeiger — wenn beim Lesen ein Fehler auftritt oder im Benutzerkatalog keine zu *uid* passende Komponente `pw_uid` gefunden wurde.

Fehler

`getpwuid()` schlägt fehl, wenn gilt:

`EFAULT` — Fehler beim Anlegen der `passwd`-Struktur

`ENOENT` — Benutzer ist unbekannt.

Hinweis

Der Returnwert kann auf einen statischen Bereich zeigen, der durch einen späteren `cuserid`-, `getpwnam`- oder `getpwuid`-Aufruf überschrieben werden kann.

Wenn Fehlersituationen untersucht werden sollen, muss `errno` vor dem Aufruf von `getpwuid()` auf 0 gesetzt werden.

Die drei Namen eines aktuellen Prozesses können wie folgt festgestellt werden: `getpwuid(geteuid())` gibt u.a. den Namen zurück, der mit der effektiven Benutzernummer des Prozesses verbunden ist, `getlogin()` gibt die Benutzerkennung der aktuellen Login-Aktivität zurück, und `getpwuid(getuid())` gibt u a. den Namen zurück, der mit der realen Benutzernummer des Prozesses verbunden ist.

Siehe auch

`cuserid()`, `getpwuid_r()`, `getpwnam()`, `geteuid()`, `getuid()`, `getlogin()`, `limits.h`, `pwd.h`, `sys/types.h`, Handbuch „POSIX Grundlagen (BS2000/OSD)“.

getpwuid_r - Benutzernummer threadsicher ermitteln

Syntax

```
#include <sys/types.h>
#include <pwd.h>

int getpwuid_r(uid_t uid, struct passwd *pwd, char *buffer,
               size_t bufsize, struct passwd **result);
```

Beschreibung

Siehe `getpwuid()`.

getrlimit, setrlimit - Grenzwert für ein Betriebsmittel ermitteln bzw. setzen

Name **getrlimit, getrlimit64, setrlimit, setrlimit64**

Syntax #include <sys/resource.h>

int getrlimit (int *resource*, struct rlimit **rlp*);
int getrlimit64 (int *resource*, struct rlimit64 **rlp*);
int setrlimit (int *resource*, const struct rlimit **rlp*);
int setrlimit64 (int *resource*, const struct rlimit64 **rlp*);

Beschreibung

Dieser Aufruf limitiert die Benutzung einer Vielzahl von Betriebsmitteln durch einen Prozess und aller seiner Sohnprozesse; mit `getrlimit()` werden die Grenzwerte gelesen und mit `setrlimit()` gesetzt.

Jeder Aufruf von `getrlimit()` oder `setrlimit()` gibt ein bestimmtes Betriebsmittel *resource* und einen bestimmten Grenzwert dafür an, auf den *rlp* verweist. Der Grenzwert setzt sich aus einem Wertepaar zusammen, das in der Struktur `rlimit` steht. *rlp* muss ein Zeiger auf eine solche Struktur sein.
`rlimit` enthält die folgenden Komponenten:

```
rlim_t  rlim_cur;       /* aktueller Grenzwert */
rlim_t  rlim_max;       /* maximaler Grenzwert */
```

`rlim_t` ist ein arithmetischer Datentyp, in den Objekte des Typs `int`, `size_t` und `off_t` konvertiert werden können, ohne dass Informationen verlorengehen.
`rlim_cur` gibt den aktuellen oder weichen Grenzwert an, `rlim_max` den maximalen oder harten Grenzwert. Weiche Grenzwerte können von einem Prozess auf einen Wert gesetzt werden, der kleiner oder gleich dem harten Grenzwert ist. Ein Prozess kann seinen harten Grenzwert verringern (nicht umkehrbar), so dass er größer oder gleich dem weichen Grenzwert wird. Nur ein Prozess mit entsprechender Privilegierung kann einen harten Grenzwert erhöhen. Sowohl harter als auch weicher Grenzwert können durch einen einzigen Aufruf von `setrlimit()` verändert werden, abhängig von den oben beschriebenen Beschränkungen.

Der Wert `RLIM_INFINITY`, der in `sys/resource.h` definiert ist, entspricht einem unendlich großen Grenzwert, d.h. wenn `getrlimit()` für ein Betriebsmittel `RLIM_INFINITY` zurückliefert, dann sieht die Implementierung keinen Grenzwert für dieses Betriebsmittel vor. Wird `setrlimit()` mit `RLIM_INFINITY` für ein Betriebsmittel erfolgreich ausgeführt, dann wird für dieses Betriebsmittel die Einhaltung eines Grenzwerts nicht mehr abgeprüft.

Kann bei Verwendung der Funktion `getrlimit()` der Grenzwert für ein Betriebsmittel in einem Objekt des Typs `rlimit_t` korrekt dargestellt werden, so wird diese Darstellung zurückgeliefert. Entspricht jedoch der Grenzwert dem Wert des zugehörigen, gesicherten harten Grenzwert, ist der zurückgelieferte Wert `RLIM_SAVED_MAX`. Ansonsten wird der Wert `RLIM_SAVED_CUR` zurückgeliefert.

Ist bei der Funktion `setrlimit()` der angeforderte Grenzwert `RLIM_INFINITY`, so ist kein Wert als neuer Grenzwert vorgesehen. Lautet der angefordete Grenzwert `RLIM_SAVED_MAX`, entspricht der neue Grenzwert dem zugehörigen, gesicherten harten Grenzwert. Wird `RLIM_SAVED_CUR` als Grenzwert angefordert, entspricht der neue Grenzwert dem zugehörigen, gesicherten weichen Grenzwert. Ansonsten entspricht der neue Wert dem angeforderten Wert. Außerdem wird der entsprechende gesicherte Grenzwert, wenn er in einem Objekt des Typs `rlim_t` korrekt dargestellt werden kann, durch den neuen Grenzwert überschrieben.

Wird ein Grenzwert auf `RLIM_SAVED_MAX` oder `RLIM_SAVED_CUR` gesetzt, ist das Ergebnis unbestimmt, es sei denn, ein vorheriger Aufruf von `getrlimit()` hat diesen Wert als harten oder weichen Grenzwert für den entsprechenden Betriebsmittelgrenzwert zurückgegeben.

Analog gilt dies alles auch für die Funktionen `getrlimit64()`**,** `setrlimit64()` **und für die Werte** `RLIM64_INFINITY`**,** `RLIM64_SAVED_MAX` **und** `RLIM64_SAVED_CUR`**.**

Die möglichen Betriebsmittel, deren Beschreibungen und die resultierenden Maßnahmen beim Überschreiten eines Grenzwertes werden in der folgenden Tabelle zusammengefasst:

Betriebsmittel	**Beschreibung**	**Maßnahme**
`RLIMIT_CORE`	Die maximale Größe einer Speicherabzugsdatei in Bytes, die von einem Prozess erzeugt werden darf. Eine Größe von 0 verhindert die Erzeugung von Speicherabzugsdateien.	Das Schreiben einer Speicherabzugsdatei wird bei dieser Größe beendet.
`RLIMIT_CPU`	Die maximale Dauer der CPU-Zeit, die von einem Prozess verbraucht wird.	`SIGXCPU` wird an den Prozess gesendet. Wenn der Prozess `SIGXCPU` blockiert, abfängt oder ignoriert, ist das Verhalten undefiniert.
`RLIMIT_DATA`	Die maximale Größe des Da-tensegments eines Prozesses in Bytes. Unter POSIX ist die Größe unbegrenzt, da `sbrk()`, `brk()` und `malloc()` unabhängigen Speicher verwenden.	`brk()`, `malloc()` und `sbrk()` schlagen fehl, und `errno` enthält `ENOMEM`.

Betriebsmittel	Beschreibung	Maßnahme
RLIMIT_FSIZE	Die maximale Länge einer Datei in Bytes, die von einem Prozess erzeugt werden kann. Eine Länge von 0 verhindert die Erzeugung von Dateien.	SIGXFSZ wird an den Prozess gesendet. Wenn der Prozess SIGXFSZ blockiert, abfängt oder ignoriert, schlagen weitere Versuche, die Datei zu vergößern fehl, und errno enthält EFBIG.
RLIMIT_NOFILE	Die maximale Anzahl der geöffneten Dateideskriptoren, die ein Prozess besitzen kann.	Funktionen, die neue Dateideskriptoren anlegen, schlagen fehl, und errno enthält EMFILE.
RLIMIT_STACK	Die maximale Größe des Prozess-Stacks in Bytes. Das System lässt den Stack nicht automatisch über diesen Grenzwert hinauswachsen.	SIGSEGV wird an den Prozess gesendet. Wenn der Prozess SIGSEGV blockiert, ignoriert oder abfängt und keinen alternativen Stack verwendet (siehe sigaltstack()), wird als Behandlungsmodus von SIGSEGV SIG_DFL gesetzt.
RLIMIT_AS	Die maximale Länge des Adressbereichs eines Prozesses in Bytes.	Die Funktionen brk(), malloc(), mmap() und sbrk() schlagen fehl, und errno enthält ENOMEM. Außerdem kann der Stack nicht mehr anwachsen, und die oben genannten Effekte treten auf.

Da die Grenzwertinformationen für jeden Prozess verwaltet werden, muss die Shell-Anweisung ulimit direkt diesen Systemaufruf ausführen, um alle zukünftigen Prozesse zu beeinflussen, die von der Shell erzeugt werden.

Der Wert des aktuellen Grenzwerts der folgenden Betriebsmittel beeinflusst diese implementierungsabhängigen Konstanten:

Grenzwert	Implementierungsabhängige Konstante
RLIMIT_FSIZE	FCHR_MAX
RLIMIT_NOFILE	OPEN_MAX

Es besteht kein funktionaler Unterschied zwischen getrlimit() / setrlimit() und getrlimit64()/setrlimit64(), außer dass getrlimit64() und setrlimit64() eine rlimit64-Struktur verwenden.

Die Struktur rlimit64 ist analog zu rlimit definiert:

```
rlim64_t rlim_cur
rlim64_t rlim_max
```

Werden Threads verwendet, so wirkt sich die Funktion auf den Prozess oder auf einen Thread wie folgt aus:

- `RLIMIT_CPU`: ... Wenn der Prozess das Signal `SIGXCPU` abfängt oder ignoriert oder alle Threads, die zu diesem Prozess gehören, dieses Signal blockieren, kommt es zu undefiniertem Verhalten.
- `RLIMIT_FSIZE`: ..., wird das Signal `SIGXFSZ` für den Thread generiert. Wenn der Thread das Signal `SIGXFSZ` blockiert oder der Prozess dieses abfängt bzw. ignoriert, schlagen weitere Versuche, die Datei zu vergrößern, fehl und `errno` erhält `EFBIG`.
- `RLIMIT_STACK`:..., wird das Signal `SIGSEGV` für den Thread generiert. Wenn der Thread das Signal `SIGSEGV` blockiert oder der Prozess dieses abfängt bzw. ignoriert und keinen alternativen Stack verwendet, wird als Behandlungsmodus von `SIGSEGV SIG_DFL` gesetzt.

Returnwert

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

Fehler `getrlimit() und setrlimit() schlagen fehl, wenn gilt:`

`EINVAL`	Ein ungültiges Betriebsmittel wurde angegeben, oder bei einem Aufruf von `setrlimit()` ist der neue Wert in `rlim_cur` größer als der in `rlim_max`.
`EPERM`	Der Grenzwert, der in `setrlimit()` angegeben ist, würde den maximalen Grenzwert erhöhen, aber der aufrufende Prozess verfügt nicht über die entsprechende Privilegierung.

Zusätzlich schlägt `setrlimit() fehl, wenn gilt:`

`EINVAL`	Der angegebene Grenzwert kann nicht vermindert werden, da aktuell bereits ein höherer Wert benutzt wird.

Siehe auch `brk()`, `exec`, `fork()`, `getdtablesize()`, `malloc()`, `open()`, `sigaltstack()`, `sysconf()`, `ulimit()`, `stropts.h`, `sys/resource.h`.

getrusage - Informationen über die Verwendung von Betriebsmitteln abfragen

Syntax

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

int getrusage(int who, struct rusage *r_usage);
```

Beschreibung

`getrusage()` gibt Informationen über die vom aktuellen Prozess verwendeten Betriebsmittel oder seiner beendeten Kindprozesse und der Kindprozesse, auf deren Beendigung gewartet wird, zurück.

Das Argument *who* kann den Wert `RUSAGE_SELF` oder `RUSAGE_CHILDREN` enthalten. Im ersten Fall werden Informationen über die Betriebsmittel des aktuellen Prozesses zurückgeliefert. Im zweiten Fall gibt `getrusage()` Informationen aus über die Betriebsmittel der beendeten Kindprozesse des aktuellen Prozesses und die der Kindprozesse, auf die der aktuelle Prozess wartet. Wird auf den Kindprozess niemals gewartet, weil z.B. im Elternprozess `SA_NOCLDWAIT` gesetzt ist oder `SIGCHLD` auf `SIG_IGN` gesetzt wird, dann wird auch keine Information über den Betriebsmittelverbrauch des Kindprozesses zurückgegeben.

Das Argument *r_usage* zeigt auf eine Sruktur `rusage`, die die folgenden Komponenten enthält:

`struct timeval ru_utime`	`Die Gesamtzeit, in der die Ausführung im Benutzermodus stattfindet. Die Zeitdauer wird in Sekunden und Mikrosekunden angegeben.`
`struct timeval ru_stime`	`Die Gesamtzeit, in der die Ausführung im Systemmodus stattfindet. Die Zeitdauer wird in Sekunden und Mikrosekunden angegeben.`

Returnwert

0	bei Erfolg. Die Struktur `rusage` wird mit den entsprechenden Werten aufgefüllt.
-1	bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `getrusage()` schlägt fehl, wenn gilt:

`EINVAL`	Das Argument *who* enthält keinen gültigen Wert.
Erweiterung `EFAULT`	Die vom Argument *r_usage* angegebene Adresse ist kein gültiger Bereich des Adressbereichs des Prozesses. ❑

Siehe auch `exit()`, `gettimeofday()`, `read()`, `sigaction()`, `time()`, `times()`, `wait()`, `write()`, `sys/resource.h`.

gets - Zeichenkette aus Standard-Eingabestrom lesen

Syntax

```
#include <stdio.h>

char *gets(char *s);
```

Beschreibung

`gets()` liest Zeichen aus dem Standard-Eingabestrom, bis ein Zeilenendezeichen oder das Dateiende erreicht wird. Die gelesene Zeichenkette wird in den Vektor eingetragen, auf den *s* zeigt. Ein Zeilenendezeichen wird durch das Nullbyte überschrieben.

`gets()` kann die Strukturkomponente `st_atime` für die Datei, der *stream* zugeordnet ist, zum Ändern markieren (siehe `sys/stat.h`). Die Strukturkomponente `st_atime` wird aktualisiert, sobald `fgetc()`, `fgets()`, `fgetwc()`, `fgetws()`, `fread()`, `fscanf()`, `getc()`, `getchar()`, `gets()` oder `scanf()` erfolgreich für *stream* aufgerufen werden und Daten zurückliefern, die nicht durch einen vorangegangenen Aufruf von `ungetc()` oder `ungetwc()` bereitgestellt wurden.

Returnwert

Zeiger auf die Ergebniszeichenkette
: bei erfolgreicher Beendigung. `gets()` schließt die Zeichenkette mit dem Nullbyte ab.

Nullzeiger
: wenn der Datenstrom das Dateiende erreicht hat. Das Dateiendekennzeichen dieses Datenstroms wird gesetzt. `errno` wird nicht gesetzt.

  Wenn ein Lesefehler auftritt, wird das Fehlerkennzeichen des Datenstroms gesetzt. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler

Siehe `fgetc()`.

Hinweis

Das Lesen einer Zeile, die die Länge des Vektors *s* überschreitet, liefert undefinierte Ergebnisse. Die Verwendung von `fgets()` wird empfohlen.

Wenn `gets()` in der POSIX-Umgebung von `stdin` liest und `EOF` das Einlese-Endekriterium ist, erreicht man die `EOF`-Bedingung durch folgende Maßnahmen:

- am blockorientierten Terminal durch Eingabe der Tastensequenz @ @ d
- am zeichenorientierten Terminal durch Eingabe von CTRL + D

BS2000

Wenn `fgetc()` in der BS2000-Umgebung von `stdin` liest und `EOF` das Einlese-Endekriterium ist, erreicht man die `EOF`-Bedingung durch folgende Maßnahmen am Terminal:

1. K2 drücken.
2. Die Systemkommandos `EOF` und `RESUME-PROGRAM` eingeben. ❑

Ob `gets()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

Siehe auch `feof()`, `ferror()`, `fgets()`, `stdio.h`.

getsid - Prozessgruppen-ID lesen

Syntax

#include <unistd.h>

pid_t getsid(pid_t *pid*);

Beschreibung

Die Funktion `getsid()` liefert die Prozessgruppen-ID des Prozesses, der Sitzuungsleiter des Prozesses mit der Nummer *pid* ist. Wenn *pid* gleich `(pid_t)0` ist, liefert `getsid()` die Sitzungsnummer des aufrufenden Prozesses zurück.

Returnwert

Prozessgruppen-ID — bei Erfolg.

`(pid_t)-1` — bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler

`getsid()` schlägt fehl, wenn gilt:

`EPERM` — Der Prozess mit der Prozessnummer *pid* ist nicht in derselben Sitzung wie der aufrufende Prozess, und die Implementierung unterstützt den Zugriff des aufrufenden Prozesses auf die Sitzungsnummer des angegebenen Prozesses nicht.

`ESRCH` — Es gibt keinen Prozess mit der Prozessnummer *pid*.

Siehe auch `exec`, `fork()`, `getpid()`, `getpdig()`, `setpgid()`, `setsid()`, `unistd.h`.

getsubopt - Unteroptionen aus einer Zeichenkette heraustrennen

Syntax

```
#include <stdlib.h>

int getsubopt (char **optionp, char * const *tokens, char **valuep);
```

Beschreibung

`getsubopt()` trennt Unteroptionen aus einem Schalterargument heraus, welches zuerst durch `getopt()` verarbeitet wurde. Diese Unteroptionen müssen durch Kommas getrennt sein und dürfen entweder aus einem einzelnen Token oder einem Token-Wert-Paar bestehen, das durch ein Gleichheitszeichen getrennt wird. Da Kommas Unteroptionen in der Optionszeichenkette begrenzen, dürfen sie nicht Teil der Unteroption oder des Wertes einer Unteroption sein. Dementsprechend darf ein Token kein Gleichheitszeichen enthalten, da Token und zugehöriger Wert durch ein Gleichheitszeichen getrennt werden.

`getsubopt()` erhält die Adresse eines Zeigers auf die Optionszeichenkette, die einen Vektor möglicher Tokens darstellt, und die Adresse eines Zeigers auf eine Wertzeichenkette. Der Index des Tokens, das der Unteroption aus der übergebenen Zeichenkette entspricht, wird zurückgeliefert; wird keine entsprechende Unteroption gefunden, wird -1 zurückgegeben. Wenn die Optionszeichenkette bei **optionp* nur eine Unteroption enthält, aktualisiert `getsubopt()` **optionp* so, dass auf das Nullzeichen am Ende der Zeichenkette gezeigt wird; ansonsten wird die Suboption isoliert, indem das trennende Komma durch ein Nullzeichen ersetzt wird, und **optionp* zeigt auf den Anfang der nächsten Unteroption. Wird der Unteroption ein Wert zugewiesen, aktualisiert `getsubopt()` **valuep*, so dass auf das erste Zeichen des Wertes gezeigt wird. Ansonsten wird **valuep* auf `NULL` gesetzt.

Der Token-Vektor wird als Folge von Zeigern auf durch Null abgeschlossene Zeichenketten organisiert. Das Ende des Token-Vektors wird durch einen Nullzeiger gekennzeichnet.

`getsubopt()` liefert dann, wenn *valuep* nicht `NULL` ist, die Unteroption zurück, der ein Wert zugewiesen wurde. Das aufrufende Programm kann diese Information verwenden, um zu bestimmen, ob das Vorhandensein oder das Fehlen eines Wertes für diese Unteroption einen Fehler darstellt.
Findet `getsubopt()` keine Unteroption im Vektor *tokens*, sollte das aufrufende Programm entscheiden, ob es sich hierbei um einen Fehler handelt, oder ob die nichterkannte Option an ein anderes Programm übergeben werden sollte.

Returnwert Index des passenden Tokens bei Erfolg.

-1 falls kein passendes Token gefunden wurde.

Hinweis Während der Verarbeitung der Tokens werden Kommas aus der Optionszeichenkette in Nullzeichen geändert. Leerzeichen in Tokens oder Token-Wert-Paaren müssen vor der Shell durch Anführungszeichen geschützt werden.

Siehe auch `getopt()`, `stdlib.h`.

gettimeofday - Datum und Uhrzeit lesen

Syntax #include <sys/time.h>

int gettimeofday(struct timeval **tp*, void **tzp*);

Beschreibung

`gettimeofday()` liest die aktuelle Zeit für das System. Die aktuelle Zeit wird in verstrichenen Sekunden und Mikrosekunden seit dem 1. Januar 1970, 00:00 (Universal Time Coordinated) angegeben. Die Auflösung der Systemuhr ist hardwareabhängig; die Zeit kann stetig oder in Zeittakten aktualisiert werden.

tp zeigt auf eine Struktur vom Typ `timeval`, die folgende Komponenten enthält:

```
long    tv_sec;    /* Sekunden seit dem 1. Januar 1970 */
long    tv_usec;   /* und Mikrosekunden */
```

Wenn *tp* ein Nullzeiger ist, wird die aktuelle Zeit nicht gelesen.

tzp muss ein Nullzeiger sein, sonst ist das Verhalten undefiniert.

Die Umgebungsvariable `TZ` enthält Zeitzoneninformationen. Siehe `timezone`.

Returnwert

0	bei Erfolg.
-1	bei Fehler.

Hinweis Auf den Returnwert -1 im Fehlerfall sollten sich Programme, die portabel sein wollen, nicht verlassen.

Siehe auch `ctime()`, `ftime()`, `timezone`, `sys/time.h`.

gettsn - TSN ermitteln *(BS2000)*

Syntax #include <stdlib.h>

char *gettsn(void);

Beschreibung

`gettsn()` liefert die Task-Sequence-Number (TSN) des aufrufenden Programms.

Returnwert Task-Sequence-Number (TSN) des aufrufenden Programms.

Hinweis `gettsn()` schreibt sein Ergebnis in einen C-internen Datenbereich, der bei jedem Aufruf überschrieben wird.

getuid - reale Benutzernummer ermitteln

Syntax #include <unistd.h>

Optional
#include <sys/types.h> ❑

uid_t getuid(void);

Beschreibung

`getuid()` gibt die reale Benutzernummer des aufrufenden Prozesses zurück.

Returnwert reale Benutzernummer des aufrufenden Prozesses
Die Funktion ist immer erfolgreich.

Siehe auch `getegid()`, `geteuid()`, `getgid()`, `setuid()`, `sys/types.h`, `unistd.h`.

getutxent, getutxid, getutxline - auf utmpx-Eintrag zugreifen

Syntax #include <utmpx.h>

struct utmpx *getutxent (void);

struct utmpx *getutxid (const struct utmpx **id*);

struct utmpx *getutxline (const struct utmpx **line*);

Siehe auch

Siehe `endutxent()`.

getw - Maschinenwort aus Datenstrom lesen

Syntax

#include <stdio.h>

int getw(FILE **stream*);

Beschreibung

`getw()` liest das nächste Maschinenwort aus dem Datenstrom *stream*. Ein Maschinenwort hat die Größe eines `int`-Datentyps; sie kann von Rechner zu Rechner unterschiedlich sein. `getw()` nimmt für die Datei keine besondere Ausrichtung an.

`getw()` kann die Strukturkomponente `st_atime` für die Datei, der *stream* zugeordnet ist, zum Ändern markieren (siehe `sys/stat.h`). Die Strukturkomponente `st_atime` wird aktualisiert, sobald `fgetc()`, `fgets()`, `fgetwc()`, `fgetws()`, `fread()`, `fscanf()`, `getc()`, `getchar()`, `gets()` oder `scanf()` erfolgreich für *stream* aufgerufen werden und Daten zurückliefern, die nicht durch einen vorangegangenen Aufruf von `ungetc()` oder `ungetwc()` bereitgestellt wurden.

`getw()` ist nicht threadsicher.

Returnwert

nächstes Wort aus dem Eingabestrom, auf den *stream* zeigt (als `int`) bei erfolgreicher Beendigung.

`EOF` wenn der Datenstrom das Dateiende erreicht hat. Das Dateiendekennzeichen dieses Datenstroms wird gesetzt. `errno` wird nicht gesetzt.

`EOF` wenn ein Lesefehler auftritt. Das Fehlerkennzeichen des Datenstroms wird gesetzt. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler

Siehe `fgetc()`.

Hinweis

Wegen möglicher Unterschiede in Wortlänge und Ausrichtung sind Dateien, die mit `putw()` geschrieben wurden, rechnerabhängig; sie können unter Umständen auf einem anderen Rechner nicht korrekt mit `getw()` gelesen werden.

Weil die Darstellung von `EOF` eine gültige ganze Zahl ist, sollten Anwendungen, die auf Fehler überprüfen wollen, `ferror()` und `feof()` benutzen.

Ob `getw()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

Siehe auch

`ferror()`, `getc()`, `putw()`, `stdio.h`.

getwc - Langzeichen aus Datenstrom lesen

Syntax

```
#include <wchar.h>

Optional
#include <stdio.h> ❑

wint_t getwc(FILE *stream);
```

Beschreibung

`getwc()` ist sowohl als Makro als auch als Funktion implementiert. `getwc()` entspricht `fgetwc()` mit dem Unterschied, dass `getwc()` als Makro *stream* mehrmals auswerten kann. Das Argument sollte also niemals ein Ausdruck mit Seiteneffekten sein.

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Returnwert Siehe `fgetwc()`.

Fehler Siehe `fgetwc()`.

Hinweis Diese Schnittstelle wird zur Verfügung gestellt, um einige aktuelle Implementierungen und mögliche zukünftige ISO-Standards zu unterstützen.

Wenn `getwc()` als Makro verwendet wird, kann *stream* mit Seiteneffekten inkorrekt behandelt werden. Inbesondere kann `getwc(*f++)` anders funktionieren, als man es erwartet. Daher wird in solchen Situationen empfohlen, stattdessen `fgetwc()` zu benutzen.

Siehe auch `fgetwc()`, `stdio.h`, `wchar.h`.

getwchar - Langzeichen aus Standard-Eingabestrom lesen

Syntax

```
#include <wchar.h>

wint_t getwchar(void);
```

Beschreibung

Der Funktionsaufruf `getwchar(void)` entspricht `getwc(stdin)`, d.h. es wird ein Langzeichen aus dem Standard-Eingabestrom gelesen.

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Returnwert Siehe `fgetwc()`.

Fehler Siehe `fgetwc()`.

Hinweis Wenn der von `getwchar()` zurückgegebene Wert in einer Variablen vom Typ `wchar_t` abgelegt wird und dann mit dem `wint_t`-Makro `WEOF` verglichen wird, ist der Vergleich nicht erfolgreich.

Siehe auch `fgetwc()`, `getwc()`, `wchar.h`.

getwd - Pfadname des aktuellen Arbeitsverzeichnisses abfragen

Syntax #include <unistd.h>

char *getwd(char **path_name*);

Beschreibung

`getwd()` ermittelt den absoluten Pfadnamen des aktuellen Arbeitsverzeichnisses des aufrufenden Prozesses und kopiert diesen in eine Zeichenkette, auf die das Argument *path_name* zeigt.

Ist die Länge des Pfadnamens des aktuellen Arbeitsverzeichnisses einschließlich des Nullbytes größer als `{PATH_MAX}+1`, schlägt `getwd()` fehl und gibt einen Nullzeiger zurück.

Returnwert Zeiger auf eine Zeichenkette
: bei Erfolg. Die Zeichenkette enthält den absoluten Pfadnamen des aktuellen Arbeitsverzeichnisses.

Nullzeiger
: bei Fehler. Die Zeichenkette, auf die *path_name* zeigt, enthält einen Fehlertext in englischer Sprache.

Hinweis Portable Anwendungen sollten statt `getwd()` die Funktion `getcwd()` verwenden.

Siehe auch `getcwd()`, `unistd.h`.

gmatch - Muster global vergleichen *(Erweiterung)*

Syntax #include <libgen.h>

int gmatch(const char **str*, const char **pattern*);

Beschreibung

`gmatch()` überprüft, ob die mit einem Nullzeichen abgeschlossene Zeichenkette *str* mit dem mit einem Nullzeichen abgeschlossenen Zeichenketten-Muster *pattern* übereinstimmt. In Muster-Zeichenketten wird ein Gegenschrägstrich `\` als Entwertungszeichen verwendet

Returnwert ≠ 0
: wenn die Zeichenkette auf das Muster passt.

0
: wenn keine Übereinstimmung gefunden wurde.

gmtime - Datum und Uhrzeit in UTC umwandeln

Syntax

```
#include <time.h>

struct tm *gmtime(const time_t *clock);
```

Beschreibung

Die Funktion `gmtime()` akzeptiert Argumente vom Typ `time_t`, auf die *clock* zeigt und die die Zeit in Sekunden seit 00:00:00 UTC (Universal Time Coordinated, 1. Januar 1970) liefern. `gmtime()` gibt Zeiger auf `tm`-Strukturen zurück, die weiter unten erläutert werden. `gmtime()` wandelt direkt in die UTC um, d.h. die vom SINIX-System benutzte Zeit.

In der Include-Datei `time.h` sind die Vereinbarungen aller Funktionen und externer Werte sowie der `tm`-Struktur enthalten. Die Strukturvereinbarung ist wie folgt:

```
struct      tm {
    int     tm_sec;         /* Sekunden - [0, 61] für übersprungene Sek.*/
    int     tm_min;         /* Minuten - [0, 59] */
    int     tm_hour;        /* Stunden - [0, 23] */
    int     tm_mday;        /* Tag des Monats - [1, 31] */
    int     tm_mon;         /* Monate - [0, 11] */
    int     tm_year;        /* Jahre seit 1900 */
    int     tm_wday;        /* Tage seit Sonntag - [0, 6] */
    int     tm_yday;        /* Tage seit dem 1. Januar - [0, 365] */
    int     tm_isdst;       /* Option für Sommerzeit */
};
```

`tm_isdst` ist positiv, wenn Sommerzeit eingestellt ist,
null, wenn Sommerzeit nicht eingestellt ist,
und negativ, wenn die Information nicht verfügbar ist.

BS2000
`gmtime()` interpretiert die Zeitangabe vom Typ `time_t` als Anzahl der Sekunden, die seit dem 1. Januar 1950 00:00:00 vergangen sind. `gmtime()` berechnet daraus Datum und Uhrzeit und speichert das Ergebnis in einer Struktur vom Typ `tm`. `gmtime()` entspricht in dieser Implementierung `localtime()`, beide liefern die lokale Zeit.
❑

`gmtime()` ist nicht threadsicher. Verwenden Sie bei Bedarf die reentrante Funktion `gmtime_r()`.

Returnwert Zeiger auf eine Struktur vom Typ `struct tm`.

Hinweis Die Rückgabewerte von `gmtime()` zeigen auf statische Daten, die bei jedem Aufruf überschrieben werden.

`gmtime()` unterstützt nicht die lokalen Datums- und Zeit-Formate. Um maximale Portabilität zu erreichen, sollte `strftime()` verwendet werden.

`gmtime()` schreibt sein Ergebnis in einen C-internen Datenbereich, der bei jedem Aufruf überschrieben wird.
Außerdem verwenden `gmtime()` und `localtime()` denselben Datenbereich, d.h., wenn sie hintereinander aufgerufen werden, wird das Ergebnis des ersten Aufrufs überschrieben.

Siehe auch `altzone()`, `ctime()`, `daylight`, `gmtime_r()`, `localtime()`, `strftime()`, `tzname`, `tzset()`.

gmtime_r - Datum und Uhrzeit threadsicher in UTC umwandeln

Syntax #include <time.h>

struct tm *gmtime_r(const time_t * *clock*, struct tm * *result*);

Beschreibung
Die Funktion `gmtime_r()` wandelt die Zeit (Anzahl der Sekunden seit der Epoche), auf die *clock* zeigt, um in eine UTC-Zeit (Coordinated Universal Time) im durch die Struktur `struct tm` beschriebenen Format. Das Ergebnis wird im Datenbereich, auf den *result* zeigt, abgelegt.

Returnwert Adresse der Struktur, auf die *result* zeigt,
bei Erfolg.

Nullzeiger Wenn ein Fehler gefunden wurde oder wenn UTC nicht verfügbar ist.

Siehe auch `gmtime()`.

grantpt - Zugriff auf das Slave-Pseudoterminal erlauben

Syntax

```
#include <stdlib.h>

int grantpt(int fildes);
```

Beschreibung

Die Funktion `grantpt()` ändert die Zugriffsrechte und den Eigentümer des Slave-Pseudoterminals, die ihrem Master-Gegenstück zugeordnet ist. *fildes* ist ein Dateideskriptor, der von einem erfolgreichen Öffnen des Haupt-Pseudoterminals geliefert wurde. Ein Programm mit gesetztem `s`-Bit für `root` wird aufgerufen (`/usr/lib/pt_chmod`). Die Benutzernummer des Slave-Geräts wird gleich der effektiven Benutzernummer des aufrufenden Prozesses, die Gruppennummer wird auf eine reservierte Gruppennummer gesetzt. Die Zugriffsrechte werden so gesetzt, dass für das Slave-Pseudoterminal das Lesen und Schreiben für den Eigentümer und das Schreiben für die Gruppe erlaubt sind.

Returnwert

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

Fehler `grantpt()` schlägt fehl, wenn gilt:

EBADF	*fildes* ist kein gültiger offener Dateideskriptor.
EINVAL	*fildes* ist keinem Haupt-Pseudo-Terminal zugeordnet.
EACCES	Auf das entsprechende Slave-Gerät konnte nicht zugegriffen werden.

Hinweis Zusätzlich schlägt `grantpt()` fehl, falls die Anwendung eine Signalbehandlungsroutine implementiert hat, um `SIGCHLD`-Signale abzufangen.

Siehe auch `open()`, `ptsname()`, `setuid()`, `unlockpt()`, `stdlib.h`.

hsearch, hcreate, hdestroy - Hash-Tabelle verwalten

Syntax

```
#include <search.h>

ENTRY *hsearch(ENTRY item, ACTION action);
int hcreate(size_t nel);
void hdestroy(void);
```

Beschreibung

`hsearch()` ist eine Suchfunktion für Hash-Tabellen. Sie gibt einen Zeiger in eine Hash-Tabelle zurück, der die Stelle anzeigt, an der ein Eintrag gefunden wurde. Die von `hsearch()` benutzte Vergleichsfunktion ist `strcmp()`. *item* ist eine Struktur des in der Include-Datei `search.h` definierten Typs `ENTRY`, die zwei Zeiger enthält: *item.key* weist auf den Vergleichsschlüssel (vom Typ `char*`), und *item.data* (`void*`) weist auf alle anderen Daten in Zusammenhang mit diesem Schlüssel.

Erweiterung
Zeiger auf Typen, die nicht `void` sind, sind zu Zeigern auf `void` umzuwandeln. ❑

action ist ein Element des Aufzählungstyps `ACTION` (definiert in `search.h`), das die Behandlung des Eintrags angibt, wenn dieser nicht in der Tabelle gefunden werden kann. `ENTER` zeigt an, dass *item* an einem geeigneten Punkt in die Tabelle eingetragen werden soll.

Erweiterung
Ist ein Duplikat eines existierenden Eintrags vorhanden, so wird der neue Eintrag nicht eingetragen und `hsearch()` gibt den Zeiger zu dem existierenden Eintrag zurück. ❑

`FIND` zeigt an, dass kein Eintrag vorgenommen werden soll. Eine erfolglose Suche wird durch die Rückgabe eines Nullzeigers gemeldet.

`hcreate()` weist ausreichend Speicher für die Tabelle zu und muss vor `hsearch()` aufgerufen werden. *nel* schätzt die größtmögliche Anzahl von Einträgen, die eine Tabelle enthalten wird. Diese Zahl kann durch den Algorithmus nach oben justiert werden, damit bestimmte, mathematisch günstige Umstände erreicht werden.

`hdestroy()` zerstört die Suchtabelle. Ein weiterer Aufruf von `hcreate()` kann folgen. Nach einem Aufruf von `hdestroy()` kann auf die Tabelle nicht mehr zugegriffen werden.

Returnwert

Nullzeiger

`hsearch()`: wenn die Aktion `FIND` (suchen) ist und der Eintrag nicht gefunden werden konnte oder wenn die Aktion `ENTER` (eintragen) ist und die Tabelle voll ist.
`hcreate()` : wenn es nicht genug Speicherplatz für die Tabelle zuweisen kann.
`hdestroy()` gibt keinen Wert zurück.

Fehler `hsearch()` schlägt fehl, wenn gilt:

`ENOMEM` Es ist nicht genügend Speicherplatz vorhanden.

Hinweis `hsearch()` und `hcreate()` verwenden `malloc()`, um Speicherplatz zuzuweisen.

Erweiterung
Es kann jeweils nur eine Hash-Suchtabelle aktiv sein. ❑

Siehe auch `bsearch()`, `lsearch()`, `malloc()`, `strcmp()`, `tsearch()`, `search.h`.

hypot - euklidischen Abstand berechnen

Syntax #include <math.h>

double hypot(double *x*, double *y*);

Beschreibung
`hypot()` berechnet den euklidischen Abstand. *x* und *y* sind Koordinaten des Punktes, dessen Abstand berechnet werden soll.

Returnwert `sqrt`(*x***x* + *y***y*) bei Erfolg.

`HUGE_VAL` bei Überlauf.
`errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `hypot()` schlägt fehl, wenn gilt:

`ERANGE` Überlauf, das Resultat ist zu groß.

Hinweis Bei Überlauf bricht das Programm ab (Signal `SIGFPE`).

Siehe auch `cabs()`, `sqrt()`, `math.h`.

iconv - Zeichen umwandeln

Syntax #include <iconv.h>

size_t iconv(iconv_t *cd*, const char **inbuf*, size_t **inbytesleft*, char ***outbuf*, *size_t *outbytesleft*);

Beschreibung

`iconv()` wandelt eine Zeichenfolge eines Zeichensatzes in eine entsprechende Zeichenfolge eines anderen Zeichensatzes um. Die Ausgangszeichenfolge steht im durch *inbuf* angegebenen Feld, die umgewandelte Zeichenfolge wird in das durch *outbuf* angegebene Feld abgelegt. Es werden die Zeichensätze verwendet, die im Aufruf `iconv_open()` angegeben sind, der den Umwandlungsdeskriptor *cd* zurückgegeben hat. Das Argument *inbuf* zeigt auf eine Variable, die auf das erste Zeichen im Eingabe-Puffer zeigt. *inbytesleft* gibt die Anzahl der Bytes an, die umgewandelt werden müssen. Das Argument *outbuf* zeigt auf eine Variable, die auf das erste Byte im Ausgabe-Puffer zeigt; *outbytesleft* gibt die Anzahl der Bytes an.

Bei zustandsabhängigen Codierungen wird der Umwandlungsdeskriptor *cd* durch einen Aufruf, für den *inbuf* ein Nullzeiger ist oder für den *inbuf* auf einen Nullzeiger zeigt, in den ursprünglichen Shift-Zustand versetzt. Wenn `iconv()` so aufgerufen wird, *outbuf* kein Nullzeiger oder Zeiger auf einen Nullzeiger ist und *outbytesleft* auf einen positiven Wert zeigt, bringt `iconv()` die Bytefolge in den Ausgabe-Puffer, um den Ausgabe-Puffer in den ursprünglichen Shift-Zustand zu versetzen. Wenn der Ausgabe-Puffer nicht groß genug ist, um die gesamte Reset-Folge aufnehmen zu können, schlägt `iconv()` fehl. `errno` wird auf `E2BIG` gesetzt. Weitere Aufrufe, bei denen *inbuf* kein Nullzeiger oder kein Zeiger auf einen Nullzeiger ist, haben zur Folge, dass die Umwandlung auf dem aktuellen Zustand des Umwandlungsdeskriptors aufsetzt.

Wenn eine Folge von Eingabe-Bytes im angegebenen Zeichensatz kein gültiges Zeichen ergibt, hält die Umwandlung nach dem zuvor erfolgreich umgewandelten Zeichen an. Wenn der Eingabe-Puffer mit einem unvollständigen Zeichen oder einer unvollständigen Shift-Sequenz endet, so hält die Umwandlung nach den zuvor erfolgreich umgewandelten Bytes an. Wenn der Ausgabe-Puffer nicht ausreichend groß ist, um die gesamte, umgewandelte Eingabe aufnehmen zu können, hält die Umwandlung unmittelbar vor den Eingabe-Bytes an, die einen Überlauf des Ausgabe-Puffers zur Folge hätten. Die Variable, auf die *inbuf* zeigt, wird aktualisiert und zeigt dann auf das Byte nach dem letzten in der Umwandlung erfolgreich verwendeten Byte. Der Wert, auf den *inbytesleft* zeigt, wird verringert und gibt die Anzahl der Bytes an, die sich noch im Eingabe-Puffer befinden und noch nicht umgewandelt sind. Die Variable, auf die *outbuf* zeigt, wird aktualisiert und zeigt dann auf das Byte nach dem letzten Byte mit umgewandelten Ausgabedaten. Der Wert, auf den *outbytesleft* zeigt, wird verringert und gibt dann die Anzahl der Bytes an, die noch im Ausgabe-Puffer zur Verfügung stehen.

Bei zustandsabhängigen Codierungen wird der Umwandlungsdeskriptor aktualisiert und gibt dann den Shift-Zustand an, der am Ende der letzten erfolgreich umgewandelten Bytefolge gültig ist.

Wenn `iconv()` im Eingabe-Puffer ein Zeichen findet, das zwar gültig ist, für das es aber im Ziel-Zeichensatz kein entsprechendes Zeichen gibt, führt `iconv()` für dieses Zeichen eine implementierungsabhängige Umwandlung aus.

Returnwert `iconv()` aktualisiert die Variablen, auf die die Argumente zeigen. Diese geben dann das Ausmaß der Umwandlung an. Es wird die Anzahl der durchgeführten, nicht-identischen Umwandlungen zurückgegeben. Wenn die gesamte Zeichenkette im Eingabe-Puffer umgewandelt wird, ist der Wert, auf den *inbytesleft* zeigt, null. Wenn die Umwandlung der Eingabe auf Grund einer der oben angegebenen Bedingungen angehalten wird, ist der Wert, auf den *inbytesleft* zeigt, ungleich null. In diesem Fall gibt `errno` die Fehlerbedingung an. Wenn ein Fehler auftritt, gibt i`conv()` (size_t)-1 zurück und setzt `errno`, um den Fehler anzuzeigen.

Fehler `iconv()` schlägt fehl, wenn gilt:

`EILSEQ`	Die Umwandlung der Eingabe wurde auf Grund eines Eingabe-Bytes angehalten, das nicht zum Eingabe-Zeichensatz gehört.
`E2BIG`	Die Umwandlung der Eingabe wurde angehalten, weil im Ausgabe-Puffer nicht genügend Platz zur Verfügung steht.
`EINVAL`	Die Umwandlung der Eingabe wurde auf Grund eines unvollständigen Zeichens oder einer unvollständigen Shift-Sequenz am Ende des Eingabe-Puffers angehalten.
`EBADF`	Das Argument *cd* ist kein gültiger Umwandlungsdeskriptor für eine offene Datei.

Siehe auch `iconv_open()`, `iconv_close()`, `iconv.h`.

iconv_close - Deskriptor für Zeichenumwandlung freigeben

Syntax #include <iconv.h>

int iconv_close(iconv_t *cd*);

Beschreibung

`iconv_close()` löst die Zuweisung des Umwandlungsdeskriptors *cd* und aller anderen Betriebsmittel auf, die durch die Funktion `iconv_open()` gesetzt wurden.

Returnwert Bei erfolgreicher Beendigung wird der Wert 0 zurückgegeben.
Andernfalls wird der Wert -1 zurückgegeben. In diesem Fall gibt `errno` den Fehler an.

Fehler `iconv_close()` schlägt fehl, wenn gilt:

`EBADF` Der Umwandlungsdeskriptor ist ungültig.

Siehe auch `iconv()`, `iconv_open()`, `iconv.h`.

iconv_open - Deskriptor für Zeichenumwandlung erzeugen

Syntax

```
#include <iconv.h>

iconv_t iconv_open(const char *tocode, const char *fromcode);
```

Beschreibung

`iconv_open()` gibt einen Umwandlungsdeskriptor zurück, der eine Umwandlung beschreibt. Diese Umwandlung erfolgt von dem Zeichensatz, auf den das Argument *fromcode* zeigt, in den Zeichensatz, auf den das Argument *tocode* zeigt. Bei statusabhängigen Codierungen befindet sich der Umwandlungsdeskriptor in einem ursprünglichen, vom Zeichensatz abhängigen Shift-Status. Er kann unmittelbar für die Funktion `iconv()` verwendet werden.

Ein Umwandlungsdeskriptor bleibt in einem Prozess gültig, bis er von diesem Prozess geschlossen wird.

`iconv_open()` verwendet die Funktion `malloc()` zur Zuweisung von Speicherplatz für interne Pufferbereiche. Die Funktion `iconv_open()` schlägt fehl, wenn für diese Puffer nicht genügend Speicherplatz zur Verfügung steht.

Returnwert Umwandlungsdeskriptor
der für spätere Aufrufe von `iconv()` verwendet werden kann.
Andernfalls gibt `iconv_open()` den Wert (`iconv_t`)-1 zurück.
`errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `iconv_open()` schlägt fehl, wenn gilt:

`EMFILE`	Im aufrufenden Prozess sind derzeit {`OPEN_MAX`}-Dateideskriptoren geöffnet.
`ENFILE`	Derzeit sind zu viele Dateien im System geöffnet.
`ENOMEM`	Es steht nicht genügend Speicher zur Verfügung.
`EINVAL`	Die durch *fromcode* und *tocode* angegebene Umwandlung wird nicht von dieser Version unterstützt.

Siehe auch `iconv()`, `iconv_close()`, `iconv.h`.

ilogb - Exponententeil einer Gleitpunktzahl ermitteln

Syntax #include <math.h>

int ilogb (double *x*)

Beschreibung

Die Funktion `ilogb()` gibt den Exponententeil von *x* zurück. Formal ist der Returnwert, für alle *x* ungleich null, der ganzzahlige, vorzeichenbehaftete Teil von $\log_r |x|$, wobei *r* die Basis der Gleitpunktarithmetik des Prozessors (in BS2000 gilt: *r* = 16) ist.

Der Funktionsaufruf `ilogb(x)` ist gleichwertig mit dem Aufruf `(int)logb(x)`.

Returnwert Exponententeil von *x*
bei Erfolg.

`INT_MIN` falls *x* = 0.0.

Siehe auch `logb()`, `math.h`.

index - erstes Vorkommen eines Zeichens in Zeichenkette ermitteln

Syntax #include <strings.h>

char *index(const char **s*, int *c*);

Beschreibung

`index()` sucht das erste Vorkommen des Zeichens *c* in der Zeichenkette *s* und liefert bei Erfolg einen Zeiger auf die gesuchte Position in *s*.

Das abschließende Nullbyte (`\0`) wird als Zeichen mitberücksichtigt.

Returnwert Zeiger auf die Position von *c* in der Zeichenkette *s*,
bei Erfolg.

Nullzeiger wenn *c* in der Zeichenkette *s* nicht enthalten ist.

Hinweis `index()` und `strchr()` sind äquivalent.

Portable Anwendungen sollten statt `index()` die Funktion `strchr()` verwenden.

Siehe auch `rindex()`, `strchr()`, `strings.h`.

initstate, random, setstate, srandom - Pseudozufallszahlen generieren

Syntax

#include <stdlib.h>

char *initstate(unsigned int *seed*, char **state*, size_t *size*);

long random(void);

char *setstate(const char **state*);

void srandom(unsigned int *seed*);

Beschreibung

`random()` verwendet einen nichtlinearen, additiven Feedback-Zufallszahlengenerator und setzt einen Standard-Statusvektor mit der Größe von 31 langen Ganzzahlen ein, um aufeinander folgendanderfolgende Pseudo-Zufallszahlen im Bereich von 0 bis 2^{31}-1 zu generieren. Die Periode dieses Zufallszahlengenerators ist sehr groß, und zwar etwa 16 x (2^{31}-1). Die Größe des Statusvektors bestimmt die Periode des Zufallszahlengenerators. Wird ein größerer Statusvektor verwendet, so verlängert sich die Periode.

Bei 256 Byte Status-Information ist die Periode des Zufallszahlengenerators größer als 2^{69}.

Ebenso wie `rand()` erzeugt `random()` standardmäßig eine Folge von Zahlen, die dadurch dupliziert werden können, indem zuvor `srandom()` mit *seed* gleich 1 aufgerufen wird.

`srandom()` initialisiert den aktuellen Statusvektor mit dem Inhalt von *seed*.

Die Funktionen `initstate()` und `setstate()` behandeln den Neustart und die Modifizierung von Zufallszahlen-Generatoren. Mit `initstate()` kann der Statusvektor, auf den das Argument *state* zeigt, zur späteren Verwendung initialisiert werden. Das Argument *size* gibt dabei die Größe des Statusvektors in Byte an. `initstate()` verwendet *size*, um festzustellen, wie anspruchsvoll der eingesetzte Zufallszahlengenerator sein soll - je mehr Statusinformationen, desto besser die generierten Zufallszahlen. Optimale Werte für die Anzahl der Statusinformationen sind 8, 32, 64, 128 und 256 Byte; andere Angaben > 8 werden auf den nächst niedrigeren der vorgenannten Werte abgerundet. Für Werte < 8 verwendet `random()` einen einfachen, linear kongruenten Zufallszahlen-Generator. Das Argument *seed* bestimmt den Startwert für die Initialisierung, durch die ein Anfangspunkt für die Zufallszahlenfolge angegeben wird, der gleichzeitig auch für einen Neustart dient. `initstate()` gibt einen Zeiger auf den vorherigen Vektor mit Statusinformationen zurück.

Wenn `random()` aufgerufen wird, ohne dass zuvor `initstate()` ausgeführt wurde, so verhält sich `random()` so, als ob `initstate()` zuvor mit *seed*=1 und *size*=128 abgelaufen wäre.

Nachdem ein Status initialisiert wurde, ermöglicht die Funktion `setstate()` ein schnelles Wechseln der Statusvektoren. Der Statusvektor, auf den *state* zeigt, wird für die Generie-

rung weiterer Zufallszahlen bis zum nächsten Aufruf von `initstate()` oder `setstate()` verwendet. `setstate()` gibt einen Zeiger auf den vorherigen Statusvektor zurück.

`inistate()` ist nicht threadsicher. Verwenden Sie bei Bedarf die reentrante Funktion `rand_r()`.

Returnwert `random()`:

Pseudo-Zufallszahl
Die Funktion ist immer erfolgreich.

`initstate()` und `setstate()`:

Zeiger auf den vorherigen Statusvektor
bei Erfolg.
Nullzeiger bei Fehler

Hinweis Nachdem ein Statusvektor initialisiert wurde, kann er an anderer Stelle neu gestartet werden:

- indem `initstate()` mit dem gewünschten Startwert, dem Statusvektor und dessen Größe aufgerufen wird.
- indem `setstate()` mit dem Statusvektor und anschließend `srandom()` mit dem gewünschten Startwert aufgerufen wird. Der Vorteil beim Aufrufen dieser beiden Funktionen liegt darin, dass die Größe des Statusvektors nach dessen Initialisierung nicht abgespeichert werden muss.

Beispiel Mit den folgenden Anweisungen wird ein Statusvektor initialisiert, an `initstate()` übergeben und eine mit `random()` erzeugte Zufallszahl ausgegeben:

```
static long state1[32] = { 3, 0x9a319039, 0x32d9c024, 0x9b663182, 0x5da1f342,
0x7449e56b, 0xbeb1dbb0, 0xab5c5918, 0x946554fd, 0x8c2e680f, 0xeb3d799f,
0xb11ee0b7, 0x2d436b86, 0xda672e2a, 0x1588ca88, 0xe369735d, 0x904f35f7,
0xd7158fd6, 0x6fa6f051, 0x616e6b96, 0xac94efdc, 0xde3b81e0, 0xdf0a6fb5,
0xf103bc02, 0x48f340fb, 0x36413f93, 0xc622c298, 0xf5a42ab8, 0x8a88d77b,
0xf5ad9d0e, 0x8999220b, 0x27fb47b9 };

main()
{
   unsigned seed;
   int n;
   seed = 1;
   n = 128;
   initstate(seed, state1, n);
   setstate(state1);
   printf("%d", random());
}
```

Siehe auch `drand48()`, `rand()`, `rand_r()`, `srand()`, `stdlib.h`.

insque, remque - Element in Queue einfügen oder aus Queue entfernen

Syntax

```
#include <search.h>

void insque(void *element, void *pred);

void remque(void *element);
```

Beschreibung

`insque()` und `remque()` ändern Queues, die aus doppelt verketteten Elementen erzeugt werden. Die Queue kann linear oder ringförmig verkettet sein. Um die Funktionen `insque()` und `remque()` benutzen zu können, muss in der Anwendung eine Struktur definiert sein, die zunächst zwei Zeiger auf eben diese Struktur enthält. Die weiteren Komponenten der Struktur sind anwendungsspezifisch. Der erste Zeiger der Struktur verweist auf den nächsten Eintrag in der Queue. Der zweite Zeiger verweist auf den vorherigen Eintrag in der Queue. Wenn die Queue linear ist, wird sie durch Nullzeiger abgeschlossen. Die Namen der Struktur und der darin enthaltenen Zeiger sind frei wählbar.

`insque()` fügt das Element, auf das *element* zeigt, in einer Queue direkt hinter *pred* ein.

`remque()` entfernt das Element, auf das *element* zeigt, aus einer Queue.

Der Aufruf `insque(&`*element*`, NULL)`, wobei *element* das erste Element der Queue ist, dient dazu, eine lineare Liste zu initialisieren. Die beiden Zeiger von *element* werden mit Nullzeigern belegt.

Um eine ringförmig verkettete Liste aufzubauen, muss die Anwendung zunächst die Adresse des Startelements der Queue in die beiden Zeiger des Startelements eintragen.

ioctl - Geräte und STREAMS steuern

Syntax #include <stropts.h>

int ioctl(int *fildes*, int *request*, .../* arg */);

Beschreibung

`ioctl()` führt eine Vielzahl an Steuerfunktionen für Geräte und `STREAMS` aus. Bei Nicht-`STREAMS`-Dateien sind die von diesem Aufruf ausgeführten Funktionen undefiniert. Das Argument *request* und ein optionales drittes Argument mit variierendem Typ werden an die mit *fildes* bezeichnete Datei weitergereicht und vom Gerätetreiber interpretiert.

fildes ist ein offener Dateideskriptor, der sich auf ein Gerät bezieht.

request wählt die auszuführende Steuerfunktion aus und hängt jeweils von den adressierten Geräten ab.

arg beinhaltet zusätzliche Informationen, die von diesen spezifischen Geräten zur Ausführung der angefragten Funktion benötigt werden. Der Datentyp von *arg* hängt von der jeweiligen Steuerfunktion ab, ist jedoch entweder eine ganze Zahl oder ein Zeiger auf eine gerätespezifische Datenstruktur.

Die folgenden `ioctl()`–Kommandos, mit den jeweils angegebenen Fehlernummern, können auf alle `STREAMS`-Dateien angewendet werden:

`I_PUSH` Klinkt das Modul, auf dessen Name *arg* zeigt, in den Anfang des aktuellen Streams ein, direkt unterhalb des Stream-Kopfs. Ist der Stream eine Pipe, dann wird das Modul zwischen den Stream-Köpfen beider Enden der Pipe eingeklinkt. Danach ruft dieses Kommando die `open()`-Funktion des neu eingeklinkten Moduls auf.

`ioctl()` mit dem `I_PUSH`–Kommando schlägt fehl, wenn gilt:

`EINVAL`	Ungültiger Modulname.
`EFAULT`	*arg* zeigt auf einen Punkt außerhalb des reservierten Adressraums.
`ENXIO`	Die `open()`-Funktion des neuen Moduls schlug fehl.
`ENXIO`	Verbindungsabbruch für *fildes* empfangen.

`I_POP` Klinkt das Modul direkt unterhalb des Stream-Kopfs aus dem Stream aus, auf den *fildes* verweist. Damit ein Modul aus einer Pipe ausgeklinkt werden kann, muss das Modul von der Seite her ausgeklinkt werden, von der es eingeklinkt worden ist. *arg* sollte in diesem Fall gleich 0 sein. Im Fehlerfall nimmt `errno` einen der folgenden Werte an:

`ioctl()` mit dem `I_POP`–Kommando schlägt fehl, wenn gilt:

`EINVAL`	Kein Modul Stream vorhanden.
`ENXIO`	Verbindungsabbruch für *fildes* empfangen.

I_LOOK Ermittelt den Namen des Moduls unmittelbar unterhalb des Stream-Kopfs in dem Stream, der durch *fildes* angegeben wird, und legt ihn in einer mit dem Nullbyte abgeschlossenen Zeichenkette ab, auf die *arg* zeigt. Der Puffer, auf den *arg* zeigt, sollte mindestens FMNAMESZ+1 Bytes lang sein. FMNAMESZ ist in stropts.h definiert.

ioctl() mit dem I_LOOK–Kommando schlägt fehl, wenn gilt:

EINVAL Kein Modul im Stream vorhanden.
EFAULT *arg* zeigt auf einen Punkt außerhalb des reservierten Adressraums.

I_FLUSH Leert alle Lese- und/oder Schreib-Queues, je nachdem, welchen Wert *arg* hat. Zulässige Werte für *arg* sind:

FLUSHR Lese-Queues leeren.
FLUSHW Schreib-Queues leeren.
FLUSHRW Lese- und Schreib-Queues leeren.

ioctl() mit dem I_FLUSH–Kommando schlägt fehl, wenn gilt:

EINVAL Ungültiger Wert für *arg*.
EAGAIN oder ENOSR
Es konnte kein Puffer für die flush-Nachricht reserviert werden, da nicht genügend STREAMS-Speicherplatz verfügbar war.
ENXIO Verbindungsabbruch für *fildes* empfangen.

I_FLUSHBAND
Leert ein bestimmtes Band von Nachrichten. *arg* zeigt auf eine bandinfo-Struktur, die folgende Komponenten besitzt:

unsigned char *bi_pri*;
int *bi_flag*;

Die *bi_flag*-Komponente kann gleich FLUSHR, FLUSHW oder FLUSHRW sein (siehe oben). Die *bi_pri*-Komponente bestimmt das Prioritätsband.

I_SETSIG Informiert den Stream-Kopf darüber, dass der Benutzer will, dass der Systemkern das SIGPOLL-Signal auslöst (siehe signal()), wenn ein bestimmtes Ereignis für den dem *fildes* zugeordneten Stream eintritt. I_SETSIG unterstützt die Fähigkeit zur asynchronen Verarbeitung unter STREAMS. Der Wert von *arg* ist eine Bitmaske, die angibt, bei welchen Ereignissen das Signal ausgelöst werden soll. Es handelt sich dabei um das bitweise ODER einer beliebigen Kombination der folgenden Konstanten:

S_RDNORM Eine Nachricht normaler Priorität (Prioritätsband = 0) befindet sich an der Spitze der Lese-Queue des Stream-Kopfs. Ein Signal wird auch dann erzeugt, wenn die Nachricht die Länge 0 hat.

S_RDBAND Eine Nachricht im Prioritätsband > 0 befindet sich an der Spitze der Lese-Queue des Stream-Kopfs. Ein Signal wird auch dann erzeugt, wenn die Nachricht die Länge 0 hat.

S_INPUT Irgendeine Nachricht ungleich M_PCPROTO (hochprior) traf in der Lese-Queue des Stream-Kopfs ein. Dieses Ereignis wird aus Gründen der Kompatibilität zu früheren Versionen von UNIX System V angeboten. Ein Signal wird auch dann erzeugt, wenn die Nachricht die Länge 0 hat.

S_HIPRI Eine Nachricht mit hoher Priorität (M_PCPROTO) befindet sich an der Spitze der Lese-Queue des Stream-Kopfs. Ein Signal wird auch dann erzeugt, wenn die Nachricht die Länge 0 hat.

S_OUTPUT Eine Schreib-Queue für normale Daten (Prioritätsband = 0) gerade unterhalb des Stream-Kopfes ist nicht mehr voll (ohne Flusskontrolle). Dies informiert den Benutzer, dass Platz in der Queue vorhanden ist, normale Daten in Richtung stream-abwärts zu senden oder zu schreiben.

S_WRNORM Genau wie S_OUTPUT.

S_WRBAND
Eine Schreib-Queue für Daten im Prioritätsband ≠ 0 gerade unterhalb des Stream-Kopfes ist nicht mehr voll. Dies informiert einen Benutzer, dass Platz in der Queue vorhanden ist, Daten mit Priorität in Richtung stream-abwärts zu senden oder zu schreiben.

S_MSG Eine M_SIG- oder M_PCSIG-Nachricht, die das Signal SIGPOLL enthält, hat die Spitze der Lese-Queue des Stream-Kopfs erreicht.

S_ERROR Eine M_ERROR-Nachricht hat den Stream-Kopf erreicht.

S_HANGUP
Eine M_HANGUP-Nachricht hat den Stream-Kopf erreicht.

S_BANDURG
Wird dieses Ereignis zusammen mit S_RDBAND verwendet, dann wird SIGURG statt SIGPOLL erzeugt, wenn eine Nachricht hoher Priorität die Spitze der Lese-Queue des Stream-Kopfs erreicht.

Ist der Wert von *arg* gleich 0, dann meldet sich der aufrufende Prozess wieder ab, und er erhält keine weiteren SIGPOLL-Signale mehr.

Ein Benutzerprozess kann entscheiden, nur im Fall von Nachrichten hoher Priorität ein Signal zugestellt zu bekommen, wenn er die Bitmaske *arg* auf den Wert S_HIPRI setzt.

Prozesse, die das Signal SIGPOLL erhalten wollen, müssen sich explizit für den Empfang anmelden, indem sie I_SETSIG verwenden. Wenn sich mehrere Prozesse für dieses Signal anmelden und dabei das gleiche Ereignis für denselben Stream anfordern, dann erhält jeder Prozess das Signal, wenn das Ereignis eintritt.

ioctl() mit dem I_SETSIG–Kommando schlägt fehl, wenn gilt:

EINVAL Der Wert von *arg* ist ungültig oder *arg* ist gleich 0 und der Prozess nicht für den Empfang des SIGPOLL-Signals angemeldet.

EAGAIN Die Reservierung einer Datenstruktur für die Signal-Anforderung schlug fehl.

I_GETSIG Liefert die Ereignisse, für die sich der aufrufende Prozess derzeit angemeldet hat, um ein SIGPOLL-Signal zu erhalten. Die Ereignisse werden in der Bitmaske zurückgeliefert, auf die *arg* zeigt, wobei die Ereignisse die in der Beschreibung von I_SETSIG spezifizierten sind (siehe oben).

ioctl() mit dem I_GETSIG–Kommando schlägt fehl, wenn gilt:

EINVAL Der Prozess ist nicht für den Empfang des SIGPOLL-Signals angemeldet.

EFAULT *arg* zeigt auf einen Punkt außerhalb des reservierten Adressraums.

I_FIND Vergleicht die Namen aller Module, die sich derzeit im Stream befinden, mit dem Namen, auf den *arg* zeigt. Es liefert den Wert 1 zurück, wenn das angegebene Modul im Stream vorhanden ist. Es liefert den Wert 0, wenn das angegebene Modul nicht eingeklinkt ist.

ioctl() mit dem I_FIND–Kommando schlägt fehl, wenn gilt:

EINVAL *arg* enthält keinen gültigen Modulnamen.

EFAULT *arg* zeigt auf einen Punkt außerhalb des reservierten Adressraums.

I_PEEK Erlaubt einem Benutzer, die Informationen in der ersten Nachricht in der Lese-Queue des Stream-Kopfs zu lesen, ohne die Nachricht aus der Queue zu entfernen. I_PEEK arbeitet analog zu getmsg(), außer dass dieses Kommando die Nachricht nicht aus der Queue entfernt.

arg zeigt auf eine strpeek-Struktur, die folgende Komponenten enthält:

```
struct strbuf ctlbuf;
struct strbuf databuf;
long flags;
```

Die maxlen-Komponente in den strbuf-Strukturen *ctlbuf* und *databuf* (siehe getmsg()) muss gleich der Anzahl der Bytes gesetzt sein, die als Steuer- und/oder Daten-Informationen gelesen werden sollen. *flags* kann gleich

RS_HIPRI oder gleich 0 gesetzt sein. Ist RS_HIPRI gesetzt, dann sucht I_PEEK eine Nachricht hoher Priorität in der Lese-Queue des Stream-Kopfs. Andernfalls sucht I_PEEK nach der ersten Nachricht in der Lese-Queue des Stream-Kopfs.

I_PEEK liefert den Wert 1, wenn eine Nachricht gefunden wurde. I_PEEK liefert den Wert 0, wenn keine Nachricht in der Lese-Queue des Stream-Kopfs gefunden werden konnte, bzw. wenn *flags* auf RS_HIPRI gesetzt war und keine Nachricht mit hoher Priorität gefunden wurde. Dieses Kommando wartet nicht darauf, dass eine Nachricht eintrifft. Nach der Rückkehr liefert *ctlbuf* die Informationen aus dem Steuerteil, *databuf* die Informationen aus dem Datenteil und *flags* enthält den Wert RS_HIPRI oder 0.

ioctl() mit dem I_PEEK–Kommando schlägt fehl, wenn gilt:

EFAULT *arg* zeigt auf einen Punkt außerhalb des reservierten Adressraums oder der Pufferbereich, der in *ctlbuf* oder *databuf* angegeben wurde, befindet sich außerhalb des reservierten Adressraums.

EBADMSG Zu lesende eingereihte Nachricht ist für I_PEEK ungültig.

EINVAL *flags* hat einen ungültigen Wert.

I_SRDOPT Setzt die Einstellung für das Lesen (siehe read()) auf den Wert des Arguments *arg*. Zulässige Werte für *arg* sind:

RNORM Betriebsart „Bytestrom“ (Voreinstellung).

RMSGD Betriebsart „Nachricht verwerfen“.

RMSGN Betriebsart „Nachricht nicht verwerfen“.

Ist der Wert für *arg* durch bitweises ODER von RMSGD und RMSGN entstanden, so führt dies zum Fehler EINVAL. Bitweises ODER von RNORM mit RMSGD ergibt RMSGN; bitweises ODER von RNORM mit RMSGN ergibt RMSGD.

Zusätzlich kann die Behandlung von Steuer-Nachrichten durch den Stream-Kopf durch folgende Kennzeichen für *arg* geändert werden:

RPROTNORM
read() schlägt mit EBADMSG fehl, wenn sich eine Steuer-Nachricht am Anfang der Lese-Queue des Stream-Kopfs befindet. Dies ist das Standard-Verhalten.

RPROTDAT Liefert den Steuerteil einer Nachricht als Daten, wenn ein Benutzer read() aufruft.

RPROTDIS Verwirft den Steuerteil einer Nachricht und liefert einen vorhandenen Datenteil aus, wenn ein Benutzer read() aufruft.

ioctl() mit dem I_SRDOPT–Kommando schlägt fehl, wenn gilt:

EINVAL *arg* hat keinen der oben genannten legalen Werte.

I_GRDOPT Liefert die derzeit gültige Einstellung für das Lesen in der int-Variablen, auf die *arg* zeigt. Die Einstellungen für das Lesen werden unter read() beschrieben.

ioctl() mit dem I_GRDOPT–Kommando schlägt fehl, wenn gilt:

EFAULT *arg* zeigt auf einen Punkt außerhalb des reservierten Adressraums.

I_NREAD Zählt die Anzahl der Datenbytes in den Datenblöcken der ersten Nachricht in der Lese-Queue des Stream-Kopfs und legt diese Anzahl in der Variablen ab, auf die *arg* zeigt. Das Ergebnis für dieses Kommando ist die Anzahl der Nachrichten in der Lese-Queue des Stream-Kopfs. Wird z. B. in *arg* der Wert 0 zurückgeliefert, aber der ioctl-Aufruf liefert ein Ergebnis größer als 0, dann zeigt dies an, dass die nächste Nachricht in der Queue die Länge 0 hat.

ioctl() mit dem I_NREAD–Kommando schlägt fehl, wenn gilt:

EFAULT *arg* zeigt auf einen Punkt außerhalb des reservierten Adressraums.

I_FDINSERT

Erzeugt eine Nachricht aus benutzerdefinierten Puffern, fügt Informationen über einen anderen Stream hinzu und sendet die Nachricht stream-abwärts. Die Nachricht enthält einen Steuer- und einen optionalen Datenteil. Die zu sendenden Daten- und Steuerteile werden dadurch unterschieden, dass sie in eigenen Puffern abgelegt werden (siehe unten).

arg zeigt auf eine strfdinsert-Struktur, die folgende Komponenten besitzt:

```
struct strbuf ctlbuf;
struct strbuf databuf;
long flags;
Int fildes;
int offset;
```

Die *len*-Komponente in der strbuf-Struktur *ctlbuf* (siehe putmsg()) muss gleich der Größe eines Zeigers plus der Anzahl von Bytes für die Steuer-Informationen dieser Nachricht sein. *fildes* in der strfdinsert-Struktur gibt den Dateideskriptor des anderen Streams an. *offset* muss auf Wortgrenze ausgerichtet sein und gibt die Anzahl der Bytes an, nach der I_FDINSERT einen Zeiger hinter dem Anfang des Steuerpuffers ablegt. Dieser Zeiger ist die Adresse der Lese-Queue-Struktur des Treibers für den Stream, der *fildes* in der Struktur strfdinsert entspricht. Die *len*-Komponente in der strbuf-Struktur *databuf* muss gleich der Anzahl der Bytes gesetzt sein, die als Daten-Informationen mit der Nachricht gesendet werden sollen, oder 0, wenn kein Datenteil gesendet werden soll.

flags gibt an, welche Art von Nachricht erzeugt werden soll. Eine normale Nachricht wird erzeugt, wenn *flags* gleich 0 ist, eine Nachricht hoher Priorität wird erzeugt, wenn *flags* gleich RS_HIPRI ist. Bei normalen Nachrichten blockiert I_FDINSERT, wenn die Schreib-Queue des Streams auf Grund der internen Flusskontrolle voll ist. Bei Nachrichten hoher Priorität blockiert I_FDINSERT in diesem Fall nicht. Bei normalen Nachrichten blockiert I_FDINSERT dann nicht, wenn die Schreib-Queue voll ist, aber O_NDELAY oder O_NONBLOCK gesetzt ist. Statt dessen schlägt der Aufruf fehl und errno ist dann gleich EAGAIN.

I_FDINSERT blockiert auch, wenn der Aufruf auf die Verfügbarkeit von Nachrichten-Blöcken wartet und nicht durch ein Fehlen interner Betriebsmittel daran gehindert wird. Dabei ist es gleichgültig, welche Priorität gesetzt ist und ob O_NDELAY oder O_NONBLOCK angegeben wurden. Es wird keine Teil-Nachricht gesendet.

ioctl() mit dem I_FDINSERT–Kommando schlägt fehl, wenn gilt:

EAGAIN Es wurde ein Nachricht ohne Priorität angegeben, O_NDELAY oder O_NONBLOCK ist gesetzt und die Schreib-Queue des Streams ist auf Grund der internen Flusskontrolle voll.

EAGAIN oder ENOSR
Es konnten keine Puffer für die zu erzeugende Nachricht reserviert werden, da zu wenig Speicherplatz unter STREAMS verfügbar war.

EINVAL Es liegt einer der folgenden Gründe vor:
- *fildes* in der strfdinsert-Struktur ist kein gültiger offener Dateideskriptor für einen Stream.
- Die Größe eines Zeigers plus *offset* ist größer als die *len*-Komponente des Puffers, der durch *ctlptr* angegeben wurde.
- *offset* gibt keinen korrekt ausgerichteten Ort im Datenpuffer an.
- *flags* hat einen undefinierten Wert.

ENXIO Ein Verbindungsabbruch wurde für *fildes* im ioctl-Aufruf oder für *fildes* in der strfdinsert-Struktur empfangen.

ERANGE Die *len*-Komponente für den durch *databuf* angegebenen Puffer liegt nicht in dem Bereich, der durch die Werte für die maximale und minimale Paketgröße des obersten Moduls im Stream festgelegt ist. Oder die *len*-Komponente für den durch *databuf* angegebenen Puffer ist größer als die konfigurierte maximale Größe des Datenteils einer Nachricht. Oder die *len*-Komponente für den durch *ctlbuf* angegebenen Puffer ist größer als die konfigurierte maximale Größe des Steuerteils einer Nachricht.

EFAULT *arg* zeigt auf einen Punkt außerhalb des reservierten Adressraums oder der Pufferbereich, der in *ctlbuf* oder *databuf* angegeben wurde, befindet sich außerhalb des reservierten Adressraums.

I_FDINSERT kann auch dann fehlschlagen, wenn eine Fehler-Nachricht vom Stream-Kopf des Streams empfangen wird, der zu *fildes* in der strfdinsert-Struktur gehört. In diesem Fall besitzt errno den Wert aus der Nachricht.

I_STR Erzeugt eine interne ioctl-Nachricht aus den Daten, auf die *arg* zeigt und sendet diese Nachricht stream-abwärts.

Dieser Mechanismus wird angeboten, um benutzerdefinierte ioctl()-Anforderungen stream-abwärts an Module und Treiber zu senden. Er erlaubt, dass Informationen mit dem ioctl() gesendet werden und liefert dem Benutzer alle Informationen zurück, die vom stream-abwärtigen Empfänger stream-aufwärts gesendet werden. I_STR blockiert, bis das System mit einer positiven oder negativen Bestätigung antwortet, oder bis nach einer bestimmten Zeitspanne ein Timeout erfolgt. Wenn ein Timeout erfolgt, dann schlägt der Aufruf mit errno gleich ETIME fehl.

Es kann immer höchstens ein I_STR-Aufruf in einem Stream aktiv sein. Weitere I_STR -Aufrufe blockieren, bis der aktive I_STR-Aufruf sich am Stream-Kopf beendet. Die Voreinstellung für einen Timeout bei diesen Anforderungen beträgt 15 Sekunden. O_NDELAY und O_NONBLOCK (siehe open()) haben keinen Einfluss auf diesen Aufruf.

Um Anforderungen stream-abwärts zu senden, muss *arg* auf eine strioctl-Struktur zeigen, die folgende Komponenten enthält:

```
int ic_cmd;
int ic_timout;
int ic_len;
char *ic_dp;
```

ic_cmd ist das interne ioctl()-Kommando, das an ein stream-abwärts liegendes Modul oder einen Treiber gesendet werden soll und *ic_timout* ist die Zahl der Sekunden für ein Timeout (-1 = unendlich, 0 = Voreinstellung, > 0 = wie angegeben). *ic_len* ist die Anzahl der Bytes im Daten-Argument und *ic_dp* ist ein Zeiger auf das Daten-Argument. Die *ic_len*-Komponente besitzt zwei Verwendungen: bei der Eingabe enthält sie die Länge des übergebenen Daten-Arguments, bei der Rückkehr vom Kommando enthält sie die Anzahl der an den Benutzer zurückgelieferten Bytes (der Puffer, auf den *ic_dp* zeigt, sollte groß genug sein, die maximale Länge der von einem Modul oder Treiber zurückzu-liefernden Daten aufnehmen zu können).

Der Stream-Kopf wandelt die Informationen in der strioctl-Struktur in eine interne ioctl()-Nachricht um und sendet diese stream-abwärts.

`ioctl()` mit dem `I_STR`–Kommando schlägt fehl, wenn gilt:

`EAGAIN` oder `ENOSR`
: Auf Grund fehlenden Speicherplatzes konnte kein Puffer für die `ioctl()`-Nachricht reserviert werden.

`EINVAL`
: *ic_len* ist kleiner als 0, oder *ic_len* ist größer als die konfigurierte maximale Größe des Datenteils einer Nachricht, oder *ic_timout* ist kleiner als -1.

`ENXIO`
: Verbindungsabbruch wurde für *fildes* empfangen.

`ETIME`
: Ein stream-abwärtiger `ioctl()`-Aufruf erhielt ein Timeout, bevor eine Bestätigung empfangen wurde.

`EFAULT`
: *arg* zeigt auf einen Punkt außerhalb des reservierten Adressraums oder der Pufferbereich, der von *ic_dp* und *ic_len* angegeben wurde (getrennt für gesendete und empfangene Daten), befindet sich außerhalb des reservierten Adressraums.

Ein `I_STR`-Aufruf kann auch dann fehlschlagen, wenn eine Fehler- oder Verbindungsabbruch-Nachricht vom Stream-Kopf des Streams empfangen wird, während dieser auf eine Bestätigung wartet. Zusätzlich kann eine Fehlernummer in der positiven oder negativen Nachricht zurückgeliefert werden, in dem Fall, dass das `ioctl`-Kommando weiter stream-abwärts fehlschlägt. In diesem Fall besitzt `errno` den Wert aus der Nachricht.

`I_SWROPT` Legt die Einstellungen für das Schreiben fest, wobei der Wert des Arguments *arg* benutzt wird. Zulässige Werte für *arg* sind:

`SNDZERO`
: Sendet eine Nachricht der Länge 0 stream-abwärts, wenn ein Aufruf von `write()` mit 0 Bytes erfolgt.
Soll in diesem Fall keine Nachricht der Länge 0 gesendet werden, dann darf dieses Bit in *arg* nicht gesetzt sein.

`ioctl()` mit dem `I_SWROPT`–Kommando schlägt fehl, wenn gilt:

`EINVAL`
: *arg* enthält nicht den oben angegebenen Wert.

`I_GWROPT` Liefert die aktuell gültige Einstellung für das Schreiben in der `int`-Variablen zurück, auf die *arg* zeigt (siehe unter `I_SWROPT`).

`I_SENDFD` Fordert den Stream, der *fildes* zugeordnet ist, auf, eine Nachricht, die einen Dateizeiger enthält, an den Stream-Kopf am anderen Ende der Pipe zu senden. Der Dateizeiger entspricht dem Argument *arg*, das ein offener Dateideskriptor sein muss.

I_SENDFD wandelt *arg* in den entsprechenden Dateizeiger um. Das Kommando reserviert einen Nachrichten-Block und fügt den Dateizeiger in diesen Block ein. Benutzernummer und Gruppennummer des sendenden Prozesses werden ebenfalls eingefügt. Diese Nachricht wird direkt in die Lese-Queue des Stream-Kopfs am anderen Ende der Pipe eingetragen.

ioctl() mit dem I_SENDFD–Kommando schlägt fehl, wenn gilt:

EAGAIN Der sendende Stream ist nicht in der Lage, einen Nachrichten-Block zu reservieren, der den Dateizeiger aufnehmen kann, oder die Lese-Queue des empfangenden Stream-Kopfs ist voll und kann die von I_SENDFD gesendete Nachricht nicht aufnehmen.

EBADF *arg* ist kein gültiger, offener Dateideskriptor.

EINVAL *fildes* ist nicht mit einer Pipe verbunden.

ENXIO Verbindungsabbruch für *fildes* empfangen.

I_RECVFD Ermittelt die Zuordnung zu einer offenen Dateibeschreibung einer Nachricht , die mit dem Kommando I_SENDFD für ioctl() über eine Pipe gesendet wurde und reserviert einen neuen Dateideskriptor im aufrufenden Prozess, der sich auf diese offene Dateibeschreibung bezieht. *arg* ist ein Zeiger auf einen Datenpuffer, der groß genug ist, eine strrecvfd-Datenstruktur aufzunehmen. Die Struktur strrecvfd ist in stropts.h definiert und enthält die folgenden Komponenten:

```
int fd;
uid_t uid;
gid_t gid;
char fill[8];
```

fd ist ein Dateideskriptor. *uid* und *gid* sind die Benutzer- und die Gruppennummer des sendenden Streams.

Wenn O_NDELAY und O_NONBLOCK nicht gesetzt ist (siehe open()), dann blockiert I_RECVFD, bis eine Nachricht am Stream-Kopf vorhanden ist. Ist O_NDELAY oder O_NONBLOCK gesetzt, so schlägt I_RECVFD fehl, wobei errno gleich EAGAIN ist, wenn keine Nachricht am Stream-Kopf vorhanden ist.

Wenn die Nachricht am Stream-Kopf eine Nachricht ist, die von I_SENDFD gesendet wurde, dann wird ein neuer Benutzer-Dateideskriptor für den in der Nachricht enthaltenen Dateizeiger reserviert. Der neue Dateideskriptor wird in der *fd*-Komponente der strrecvfd-Struktur abgelegt. Die Struktur wird in den Datenpuffer des Benutzers kopiert, auf den *arg* zeigt.

ioctl() mit dem I_RECVFD–Kommando schlägt fehl, wenn gilt:

EAGAIN Es befindet sich keine Nachricht in der Lese-Queue des Stream-Kopfs und O_NDELAY oder O_NONBLOCK ist gesetzt.

EBADMSG Die Nachricht in der Lese-Queue des Stream-Kopfs ist keine Nachricht, die einen übergebenen Dateideskriptor enthält.

EMFILE NOFILES Dateideskriptoren sind bereits geöffnet.

ENXIO Verbindungsabbruch für *fildes* empfangen.

EOVERFLOW
uid oder *gid* ist zu groß, um in der Struktur abgelegt werden zu können, auf die *arg* zeigt.

EFAULT *arg* zeigt auf einen Punkt außerhalb des reservierten Adressraums.

I_LIST Erlaubt es einem Benutzer, alle Modulnamen im Stream auszugeben einschließlich des obersten Treibers. Ist *arg* gleich NULL, so ist das Ergebnis des Aufrufs die Anzahl der Module (einschließlich Treiber), die sich in dem Stream befinden, auf den *fildes* verweist. Dies erlaubt dem Benutzer, genügend Platz für die Modulnamen zu reservieren. Ist *arg* ungleich NULL, dann sollte dieses Argument auf eine str_list-Struktur zeigen, die folgende Komponenten besitzt:

int *sl_nmods*;
struct str_mlist **sl_modlist*;

Die str_mlist-Struktur besitzt folgende Komponenten:

char *l_name[FMNAMESZ+1]*;

sl_nmods gibt die Anzahl der Einträge an, die der Benutzer im Vektor reserviert hat. Nach der Rückkehr enthält *sl_modlist* die Liste der Modulnamen und *sl_nmods* enthält die Anzahl der Einträge im Vektor *sl_modlist*, dies ist die Anzahl aller Module einschließlich des Treibers. Der Rückgabewert von ioctl() ist 0. Mit dem Schreiben der Einträge wird bei der Spitze des Streams begonnen und dann stream-abwärts fortgefahren, bis entweder das Ende des Streams oder die Anzahl der gewünschten Module (*sl_nmods*) erreicht ist.

ioctl() mit dem I_LIST–Kommando schlägt fehl, wenn gilt:

EINVAL Die Komponente *sl_nmods* ist kleiner als 1.

EAGAIN oder ENOSR
Puffer konnte nicht reserviert werden.

I_ATMARK Erlaubt es dem Benutzer zu überprüfen, ob die aktuelle Nachricht in der Lese-Queue des Stream-Kopfs von einem Modul weiter stream-abwärts „markiert“ wurde. *arg* legt fest, wie die Überprüfung durchgeführt wird, wenn es mehrfach „markierte“ Nachrichten in der Lese-Queue des Stream-Kopfs geben kann. Es kann folgende Werte annehmen:

ANYMARK Prüfen, ob die Nachricht markiert ist.

LASTMARK Prüfen, ob die Nachricht die letzte markierte in der Queue ist.

Das Ergebnis hat den Wert 1, wenn die entsprechende Markierungs-Bedingung erfüllt ist. Andernfalls hat es den Wert 0. Im Fehlerfall kann `errno` den folgenden Wert annehmen:

`ioctl()` mit dem `I_ATMARK`–Kommando schlägt fehl, wenn gilt:

`EINVAL` Der Wert von *arg* ist ungültig.

`I_CKBAND`

Prüft, ob eine Nachricht in einem gegebenen Prioritätsband in der Lese-Queue des Stream-Kopfs existiert. Das Ergebnis ist 1, wenn eine solche Nachricht existiert, oder -1 bei einem Fehler. *arg* sollte eine ganze Zahl sein, die den Wert des zu überprüfenden Prioritätsbands enthält.

`ioctl()` mit dem `I_CKBAND`–Kommando schlägt fehl, wenn gilt:

`EINVAL` Der Wert von *arg* ist ungültig.

`I_GETBAND`

Liefert das Prioritätsband der ersten Nachricht in der Lese-Queue des Stream-Kopfs in dem ganzzahligen Wert zurück, auf den *arg* zeigt.

`ioctl()` mit dem `I_GETBAND`–Kommando schlägt fehl, wenn gilt:

`ENODATA` Es befindet sich keine Nachricht in der Lese-Queue des Stream-Kopfs.

`I_CANPUT` Prüft, ob ein bestimmtes Band beschreibbar ist. *arg* ist gleich dem zu überprüfenden Prioritätsband. Das Ergebnis ist 0, wenn das Prioritätsband *arg* der Flusskontrolle unterliegt, 1, wenn das Band beschreibbar ist oder -1 bei einem Fehler.

`ioctl()` mit dem `I_CANPUT`–Kommando schlägt fehl, wenn gilt:

`EINVAL` Der Wert von *arg* ist ungültig.

`I_SETCLTIME`

Erlaubt es einem Benutzer, festzulegen, wie lange der Stream-Kopf wartet, wenn ein Stream geschlossen wird und sich noch Daten in den Schreib-Queues befinden. Bevor er jedes Modul und jeden Treiber schließt, wartet der Stream-Kopf die angegebene Zeit, damit die Daten noch übertragen werden können. Sind nach der Wartezeit immer noch Daten vorhanden, so werden diese verworfen. *arg* ist ein Zeiger auf die Anzahl der Millisekunden, die gewartet werden sollen, jeweils auf den nächsthöheren gültigen Wert im System gerundet. Die Voreinstellung ist 15 Sekunden.

`ioctl()` mit dem `I_SETCLTIME`–Kommando schlägt fehl, wenn gilt:

`EINVAL` Der Wert von *arg* ist ungültig.

I_GETCLTIME
: Liefert die Wartezeit beim Schließen in der `long`-Variablen zurück, auf die *arg* zeigt.

Multiplex-Konfigurationen unter STREAMS

I_LINK
: Verbindet zwei Datenströme, wobei *fildes* der Dateideskriptor des Streams ist, der mit dem Multiplex-Treiber verbunden ist und *arg* ist der Dateideskriptor des Streams, der mit einem anderen Treiber verbunden ist. Der Stream, der mit *arg* angegeben wird, wird unterhalb des Multiplex-Treibers verbunden. I_LINK erwartet, dass der Multiplex-Treiber eine Bestätigung an den Stream-Kopf sendet. Dieser Aufruf liefert eine Multiplexer-Kennzahl (diese Kennzahl wird für den Abbau des Multiplexers benötigt; siehe I_UNLINK) bei Erfolg und -1 bei einem Fehler.

  `ioctl()` mit dem I_LINK–Kommando schlägt fehl, wenn gilt:

  ENXIO
  : Verbindungsabbruch für *fildes* empfangen.

  ETIME
  : Timeout erfolgte, bevor die Bestätigung vom Stream-Kopf empfangen wurde.

  EAGAIN oder ENOSR
  : Nicht genügend Speicher unter STREAMS verfügbar, um I_LINK ausführen zu können.

  EBADF
  : *arg* ist kein gültiger, offener Dateideskriptor.

  EINVAL
  : Einer der folgenden Fehler ist aufgetreten:

    – Der *fildes* zugeordnete Stream unterstützt das Multiplexen nicht.
    – *arg* ist kein Stream, oder bereits unter einem Multiplexer verbunden.
    – Die angegebene Verbindung würde eine Schleife in der sich ergebenden Konfiguration erzeugen, d. h. ein gegebener Treiber ist in einer Multiplex-Konfiguration an mehr als einer Stelle vorhanden.
    – *fildes* ist der Dateideskriptor einer Pipe oder FIFO-Datei.

  Die Operation I_LINK kann auch fehlschlagen, wenn sie darauf wartet, dass der Multiplex-Treiber die Verbindungsanforderung bestätigt. Dies kann dann geschehen, wenn eine Nachricht am Stream-Kopf von *fildes* eintrifft, die einen Fehler oder einen Verbindungsabbruch anzeigt. Zusätzlich kann eine Fehlernummer in der positiven oder negativen Bestätigung enthalten sein. In diesen Fällen schlägt I_LINK fehl, wobei `errno` gleich dem Wert in der Nachricht ist.

I_UNLINK Löst die Verbindung zwischen den beiden durch *fildes* und *arg* angegebenen Datenströmen. *fildes* ist der Dateideskriptor des mit dem Multiplex-Treiber verbundenen Streams. *arg* ist die Multiplexer-Kennzahl, die von I_LINK zurückgeliefert wurde. Ist *arg* gleich MUXID_ALL, dann werden alle Datenströme abgehängt, die mit *fildes* verbunden waren. Ebenso wie I_LINK erwartet auch dieses Kommando, dass der Multiplex-Treiber die Auflösung der Verbindung bestätigt.

ioctl() mit dem I_UNLINK–Kommando schlägt fehl, wenn gilt:

ENXIO Verbindungsabbruch für *fildes* empfangen.

ETIME Timeout erfolgte, bevor eine Bestätigung vom Stream-Kopf empfangen wurde.

EAGAIN oder ENOSR
Es kann nicht genügend Speicherplatz für die Bestätigung reserviert werden.

EINVAL *arg* ist keine gültige Multiplexer-Kennzahl oder *fildes* ist nicht der Stream, für den die I_LINK-Operation ausgeführt wurde, die *arg* geliefert hat.

EINVAL *fildes* ist der Dateideskriptor einer Pipe oder FIFO-Datei.

Die Operation I_UNLINK kann auch fehlschlagen, wenn sie darauf wartet, dass der Multiplex-Treiber die Verbindungsanforderung bestätigt. Dies kann dann geschehen, wenn eine Nachricht am Stream-Kopf von *fildes* eintrifft, die einen Fehler oder einen Verbindungsabbruch anzeigt. Zusätzlich kann eine Fehlernummer in der positiven oder negativen Bestätigung enthalten sein. In diesen Fällen schlägt I_UNLINK fehl, wobei errno gleich dem Wert in der Nachricht ist.

I_PLINK Verbindet zwei Datenströme, wobei *fildes* der Dateideskriptor des Streams ist, der mit dem Multiplex-Treiber verbunden ist und *arg* ist der Dateideskriptor des Streams, der mit einem anderen Treiber verbunden ist. Der Stream, der mit *arg* angegeben wird, wird unterhalb des Multiplex-Treibers über einen ständigen Verweis verbunden. Dieser Aufruf erzeugt einen ständigen Verweis, der auch dann existieren kann, wenn der Dateideskriptor *fildes*, der dem oberen Stream zum Multiplex-Treiber zugeordnet ist, geschlossen wird. I_PLINK erwartet, dass der Multiplex-Treiber eine Bestätigung an den Stream-Kopf sendet. Dieser Aufruf liefert eine Multiplexer-Kennzahl (diese Kennzahl wird für den Abbau des Multiplexers benötigt; siehe I_PUNLINK) bei Erfolg und -1 bei Fehler.

ioctl() mit dem I_PLINK–Kommando schlägt fehl, wenn gilt:

ENXIO Verbindungsabbruch für *fildes* empfangen.

ETIME Timeout erfolgte, bevor eine Bestätigung vom Stream-Kopf empfangen wurde.

EAGAIN oder ENOSR
: Nicht genügend Speicher unter STREAMS verfügbar, um I_PLINK ausführen zu können.

EBADF
: *arg* ist kein gültiger, offener Dateideskriptor.

EINVAL
: Einer der folgenden Fehler ist aufgetreten:

  – Der *fildes* zugeordnete Stream unterstützt das Multiplexen nicht.
  – *arg* ist kein Stream oder bereits unter einem Multiplexer angehängt.
  – Die angegebene Verbindung würde eine Schleife in der sich ergebenden Konfiguration erzeugen, d. h. ein gegebener Treiber ist in einer Multiplex-Konfiguration an mehr als einer Stelle vorhanden.
  – *fildes* ist der Dateideskriptor einer Pipe oder FIFO-Datei.

Die Operation I_PLINK kann auch fehlschlagen, wenn sie darauf wartet, dass der Multiplex-Treiber die Verbindungsanforderung bestätigt. Dies kann dann geschehen, wenn eine Nachricht am Stream-Kopf von *fildes* eintrifft, die einen Fehler oder einen Verbindungsabbruch anzeigt. Zusätzlich kann eine Fehlernummer in der positiven oder negativen Bestätigung enthalten sein. In diesen Fällen schlägt I_PLINK fehl, wobei errno gleich dem Wert in der Nachricht ist.

I_PUNLINK
: Löst die ständige Verbindung zwischen den beiden durch *fildes* und *arg* angegebenen Datenströmen. *fildes* ist der Dateideskriptor des mit dem Multiplex-Treiber verbundenen Streams. *arg* ist die Multiplexer-Kennzahl, die von I_PLINK zurückgeliefert worden ist, als ein Stream unter dem Multiplex-Treiber eingehängt wurde. Ist *arg* gleich MUXID_ALL, dann werden alle Datenströme abgehängt, die mit *fildes* über ständige Verweise verbunden waren. Ebenso wie I_PLINK erwartet auch dieses Kommando, dass der Multiplex-Treiber die Auflösung der Verbindung bestätigt.

  ioctl() mit dem I_PUNLINK–Kommando schlägt fehl, wenn gilt:

  ENXIO
  : Verbindungsabbruch für *fildes* empfangen.

  ETIME
  : Timeout erfolgte, bevor eine Bestätigung vom Stream-Kopf empfangen wurde.

  EAGAIN oder ENOSR
  : Puffer für die Bestätigung konnte nicht reserviert werden.

  EINVAL
  : Ungültige Multiplexer-Kennzahl.

  EINVAL
  : *fildes* ist der Dateideskriptor einer Pipe oder FIFO-Datei.

Die Operation I_PUNLINK kann auch fehlschlagen, wenn sie darauf wartet, dass der Multiplex-Treiber die Verbindungs-Anforderung bestätigt. Dies kann dann geschehen, wenn eine Nachricht am Stream-Kopf von *fildes* eintrifft, die einen Fehler oder einen Verbindungsabbruch anzeigt. Zusätzlich kann eine Fehlernummer in der positiven oder negativen Bestätigung enthalten sein. In diesen Fällen schlägt I_PUNLINK fehl, wobei errno gleich dem Wert in der Nachricht ist.

Returnwert nicht negative ganze Zahl
bei Erfolg. Der zurückgegebene Wert hängt jeweils von der Geräte-Steuerfunktion ab.

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

Fehler ioctl() ist bei jedem Dateityp erfolglos, wenn einer oder mehrere der nachstehenden Punkte zutreffen:

EBADF *fildes* ist kein gültiger offener Dateideskriptor.

EINTR Ein Signal wurde während des Systemaufrufs ioctl() abgefangen.

EINVAL Der mit *fildes* bezeichnete Stream oder Multiplexer ist (direkt oder indirekt) unter einem Multiplexer eingehängt.

ioctl() ist außerdem erfolglos, wenn der Gerätetreiber einen Fehler feststellt. In diesem Fall wird der Fehler durch ioctl() ohne Änderung an den Aufrufer weitergeleitet. Nicht alle nachstehenden Fehlerfälle können bei jedem Treiber auftreten:

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

EIO Ein physikalischer E/A-Fehler ist aufgetreten.

ENOTTY *fildes* bezeichnet keinen Gerätetreiber/keine STREAMS–Datei, der/die Steuerfunktionen akzeptiert.

ENXIO *request* und *arg* sind für diesen Gerätetreiber gültig, jedoch kann der angeforderte Dienst nicht auf diesem Gerät ausgeführt werden.

ENODEV *fildes* bezeichnet eine gültige STREAMS-Datei, aber der zugehörige Gerätetreiber unterstützt die Funktion ioctl() nicht.

ENOLINK *fildes* befindet sich auf einem fernen Rechner, und die Verbindung zu diesem Rechner ist nicht mehr aktiv.

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

Wenn ein Stream unter einem Multiplexer eingehängt ist, führt jedes ioctl()-Kommando außer I_UNLINK und I_PUNLINK zum Fehler EINVAL.

Siehe auch `streamio()` in „Leitfaden für Programmierer: STREAMS“, `termio()` in „Referenzhandbuch für Systemverwalter“, `close()`, `fcntl()`, `getmsg()`, `open()`, `pipe()`, `poll()`, `putmsg()`, `read()`, `sigaction()`, `write()`, `stropts.h`.

isalnum - auf alphanumerisches Zeichen prüfen

Syntax #include <ctype.h>

int isalnum(int *c*);

Beschreibung

`isalnum()` überprüft, ob das Zeichen *c* ein Buchstabe oder eine Ziffer ist.

In allen Fällen ist das Argument *c* vom Typ *int*. Der Wert von *c* muss als *unsigned char* darstellbar oder gleich dem Wert des Makros `EOF` sein . Wenn *c* irgendeinen anderen Wert besitzt, ist das Verhalten undefiniert.

Returnwert

≠ 0	alphanumerisch
0	nicht alphanumerisch

Hinweis `isalnum()` ist sowohl als Makro als auch als Funktion realisiert. Um einen Funktionsaufruf zu erzeugen, muss die Definition des Makronamens rückgängig gemacht werden (`#undef isalnum`).

Das Verhalten von `isalnum()` wird von den Klassen `alpha` und `digit` der aktuellen Lokalität bestimmt. Die aktuelle Lokalität ist die C-Lokalität, wenn nicht explizit mit `setlocale()` umgeschaltet wurde.

Siehe auch `isalpha()`, `iscntrl()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `isxdigit()`, `setlocale()`, `ctype.h`, `stdio.h`.

isalpha - auf alphabetisches Zeichen prüfen

Syntax

```
#include <ctype.h>

int isalpha(int c);
```

Beschreibung

`isalpha()` überprüft, ob das Zeichen *c* ein Buchstabe ist.

In allen Fällen ist das Argument *c* vom Typ *int*. Der Wert von *c* muss als *unsigned char* darstellbar oder gleich dem Wert des Makros `EOF` sein. Wenn *c* irgendeinen anderen Wert besitzt, ist das Verhalten undefiniert.

Returnwert

≠ 0	Buchstabe
0	kein Buchstabe

Hinweis

`isalpha()` ist sowohl als Makro als auch als Funktion realisiert. Um einen Funktionsaufruf zu erzeugen, muss die Definition des Makronamens rückgängig gemacht werden (`#undef isalpha`).

Das Verhalten von `isalpha()` wird von der Klasse `alpha` der aktuellen Lokalität bestimmt. Die aktuelle Lokalität ist die C-Lokalität, wenn nicht explizit mit `setlocale()` umgeschaltet wurde.

Siehe auch `isalnum()`, `iscntrl()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `isxdigit()`, `setlocale()`, `ctype.h`, `stdio.h`.

isascii - auf 7-Bit ASCII-Zeichen prüfen

Syntax #include <ctype.h>

int isascii (int *c*)

Beschreibung

`isascii()` prüft, ob *c* kleiner als 128 ist.
(Der US-ASCII Zeichencode ist für die Werte von 0 bis 127 definiert.)

`isascii()` ist für alle ganzzahligen Werte definiert.

BS2000
`isascii()` ist ein Synonym für `isebcdic()`. `isascii()` prüft, ob der Wert des Zeichens *c* ein EBCDIC-Zeichen repräsentiert (Werte 0 - 255). ❑

Returnwert

≠ 0	der Wert von *c* liegt zwischen 0 und 127 (ASCII-Zeichen).
0	kein ASCII-Zeichen (Werte ≠ 0 - 127).

BS2000

≠ 0	der Wert von *c* liegt zwischen 0 und 255 (EBCDIC-Zeichen).
0	kein EBCDIC-Zeichen (Werte ≠ 0 - 255). ❑

Hinweis `isascii()` ist sowohl als Makro als auch als Funktion realisiert. Um einen Funktionsaufruf zu erzeugen, muss die Definition des Makronamens rückgängig gemacht werden (`#undef isascii`).

Siehe auch `isalnum()`, `isalpha()`, `iscntrl()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `isxdigit()`, `ctype.h`, `ascii_to_ebcdic()`, `ebcdic_to_ascii()`.

isastream - Dateideskriptor testen

Syntax #include <stropts.h>

int isastream(int *fildes*);

Beschreibung

Die Funktion `isastream()` prüft, ob ein Dateideskriptor eine `STREAMS`-Datei repräsentiert. *fildes* verweist auf eine offene Datei.

Returnwert

1	wenn *fildes* eine `STREAMS`-Datei repräsentiert.
0	wenn *fildes* keine `STREAMS`-Datei repräsentiert.
-1	bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `isastream()` schlägt fehl, wenn gilt:

`EBADF`	*fildes* ist kein gültiger, offener Dateideskriptor.

Siehe auch `stropts.h`.

isatty - auf Verbindung zu einem Terminal prüfen

Syntax #include <unistd.h>

int isatty(int *fildes*);

Beschreibung

`isatty()` prüft, ob der mit *fildes* angegebene Dateideskriptor einem Terminal zugeordnet ist.

Returnwert

1	bei Erfolg. *fildes* ist einem Terminal zugeordnet.
0	bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `isatty()` schlägt fehl, wenn gilt:

`EBADF`	*fildes* ist kein gültiger Dateideskriptor.
`ENOTTY`	*fildes* ist keinem Terminal zugeordnet.

Siehe auch `unistd.h`.

iscntrl - auf Steuerzeichen prüfen

Syntax #include <ctype.h>

int iscntrl(int *c*);

Beschreibung

`iscntrl()` überprüft, ob das Zeichen *c* ein Steuerzeichen ist. Steuerzeichen sind nicht abdruckbare Zeichen, z. B. für die Druckersteuerung.

In allen Fällen ist das Argument *c* vom Typ *int*. Der Wert von *c* muss als *unsigned char* darstellbar oder gleich dem Wert des Makros `EOF` sein. Wenn *c* irgendeinen anderen Wert besitzt, ist das Verhalten undefiniert.

Returnwert

≠ 0	Steuerzeichen
0	kein Steuerzeichen

Hinweis `iscntrl()` ist sowohl als Makro als auch als Funktion realisiert. Um einen Funktionsaufruf zu erzeugen, muss die Definition des Makronamens rückgängig gemacht werden (`#undef iscntrl`).

Das Verhalten von `iscntrl()` wird von der Klasse `cntrl` der aktuellen Lokalität bestimmt. Die aktuelle Lokalität ist die C-Lokalität, wenn nicht explizit mit `setlocale()` umgeschaltet wurde.

Siehe auch `isalnum()`, `isalpha()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `isxdigit()`, `setlocale()`, `ctype.h`.

isdigit - auf Dezimalziffer prüfen

Syntax #include <ctype.h>

int isdigit(int *c*);

Beschreibung

`isdigit()` überprüft, ob das Zeichen *c* eine Dezimalziffer ist.

In allen Fällen ist das Argument *c* vom Typ *int*. Der Wert von *c* muss als *unsigned char* darstellbar oder gleich dem Wert des Makros `EOF` sein . Wenn *c* irgendeinen anderen Wert besitzt, ist das Verhalten undefiniert.

Returnwert

≠ 0	Dezimalziffer
0	keine Dezimalziffer

Hinweis `isdigit()` ist sowohl als Makro als auch als Funktion realisiert. Um einen Funktionsaufruf zu erzeugen, muss die Definition des Makronamens rückgängig gemacht werden (`#undef isdigit`).

Das Verhalten von `isdigit()` wird von der Klasse `digit` der aktuellen Lokalität bestimmt. Die aktuelle Lokalität ist die C-Lokalität, wenn nicht explizit mit `setlocale()` umgeschaltet wurde.

Siehe auch `isalnum()`, `isalpha()`, `iscntrl()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `isxdigit()`, `ctype.h`.

isebcdic - auf EBCDIC-Zeichen prüfen *(BS2000)*

Syntax #include <ctype.h>

int isebcdic(int *c*);

Beschreibung

`isebcdic` prüft, ob der Wert des Zeichens *c* ein EBCDIC-Zeichen repräsentiert (Werte 0 - 255).

Returnwert

≠ 0	der Wert von *c* repräsentiert ein EBCDIC-Zeichen (Werte 0 - 255).
0	kein EBCDIC-Zeichen (Werte ≠ 0 - 255).

Hinweis `isebcdic` ist sowohl als Makro als auch als Funktion realisiert. Um einen Funktionsaufruf zu erzeugen, muss die Definition des Makronamens rückgängig gemacht werden (`#undef isebcdic`).

`isebcdic` ist ein Synonym für `isascii`.

Siehe auch `isalpha()`, `isalnum()`, `isascii()`, `iscntrl()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `isxdigit()`.

isgraph - auf darstellbares Zeichen prüfen

Syntax #include <ctype.h>

int isgraph(int *c*);

Beschreibung

`isgraph()` überprüft, ob *c* ein darstellbares Zeichen ist. Darstellbare Zeichen sind: alphanumerische Zeichen und Sonderzeichen. Leerzeichen gelten als nicht darstellbar.

In allen Fällen ist das Argument *c* vom Typ *int*. Der Wert von *c* muss als *unsigned char* darstellbar oder gleich dem Wert des Makros `EOF` sein. Wenn *c* irgendeinen anderen Wert besitzt, ist das Verhalten undefiniert.

Returnwert

≠ 0	darstellbar
0	nicht darstellbar

Hinweis `isgraph()` ist sowohl als Makro als auch als Funktion realisiert. Um einen Funktionsaufruf zu erzeugen, muss die Definition des Makronamens rückgängig gemacht werden (`#undef isgraph`).

Das Verhalten von `isgraph()` wird von der Klasse `graph` der aktuellen Lokalität bestimmt. Die aktuelle Lokalität ist die C-Lokalität, wenn nicht explizit mit `setlocale()` umgeschaltet wurde.

Siehe auch `isalnum()`, `isalpha()`, `iscntrl()`, `isdigit()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `isxdigit()`, `setlocale()`, `ctype.h`.

islower - auf Kleinbuchstaben prüfen

Syntax #include <ctype.h>

int islower(int *c*);

Beschreibung

`islower()` überprüft, ob das Zeichen *c* ein Kleinbuchstabe ist.

In allen Fällen ist das Argument *c* vom Typ *int*. Der Wert von *c* muss als *unsigned char* darstellbar oder gleich dem Wert des Makros `EOF` sein. Wenn *c* irgendeinen anderen Wert besitzt, ist das Verhalten undefiniert.

Returnwert

≠ 0	Kleinbuchstabe
0	kein Kleinbuchstabe

Hinweis `islower()` ist sowohl als Makro als auch als Funktion realisiert. Um einen Funktionsaufruf zu erzeugen, muss die Definition des Makronamens rückgängig gemacht werden (`#undef islower`).

Das Verhalten von `islower()` wird von der Klasse `lower` der aktuellen Lokalität bestimmt. Die aktuelle Lokalität ist die C-Lokalität, wenn nicht explizit mit `setlocale()` umgeschaltet wurde.

Siehe auch `isalnum()`, `isalpha()`, `iscntrl()`, `isdigit()`, `isgraph()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `isxdigit()`, `setlocale()`, `ctype.h`.

isnan - auf NaN (not a number) prüfen

Syntax #include <math.h>

int isnan(double *x*);

Beschreibung

`isnan()` überprüft, ob *x* kein `NaN` ist.
Kein `NaN` bedeutet, *x* ist ein gültiges Bitmuster einer Gleitpunktzahl.

Returnwert

0	wenn *x* kein `NaN` ist.

Hinweis In dieser Implementierung liefert `isnan()` immer den Wert 0, d.h. alle Bitmuster für Gleitpunktzahlen sind gültig.

Siehe auch `math.h`.

isprint - auf druckbares Zeichen prüfen

Syntax #include <ctype.h>

int isprint(int *c*);

Beschreibung

`isprint()` überprüft, ob *c* ein druckbares Zeichen ist. Druckbare Zeichen sind: alphanumerische Zeichen, Sonderzeichen und Leerzeichen.

In allen Fällen ist das Argument *c* vom Typ *int*. Der Wert von *c* muss als *unsigned char* darstellbar oder gleich dem Wert des Makros `EOF` sein. Wenn *c* irgendeinen anderen Wert besitzt, ist das Verhalten undefiniert.

Returnwert

$\neq 0$	druckbar (alphanumerisches Zeichen, Sonderzeichen oder Leerzeichen).
0	nicht druckbar

Hinweis `isprint()` ist sowohl als Makro als auch als Funktion realisiert. Um einen Funktionsaufruf zu erzeugen, muss die Definition des Makronamens rückgängig gemacht werden (`#undef isprint`).

Das Verhalten von `isprint()` wird von der Klasse `print` der aktuellen Lokalität bestimmt. Die aktuelle Lokalität ist die C-Lokalität, wenn nicht explizit mit `setlocale()` umgeschaltet wurde.

Siehe auch `isalnum()`, `isalpha()`, `iscntrl()`, `isdigit()`, `isgraph()`, `islower()`, `ispunct()`, `isspace()`, `isupper()`, `isxdigit()`, `setlocale()`, `ctype.h`.

ispunct - auf Sonderzeichen prüfen

Syntax

```
#include <ctype.h>

int ispunct(int c);
```

Beschreibung

`ispunct()` überprüft, ob *c* ein Sonderzeichen ist, d.h. weder ein Steuerzeichen noch ein alphanumerisches Zeichen, noch ein Zeichen für Zwischenraum (siehe `isspace`).

In allen Fällen ist das Argument *c* vom Typ *int*. Der Wert von *c* muss als *unsigned char* darstellbar oder gleich dem Wert des Makros `EOF` sein. Wenn *c* irgendeinen anderen Wert besitzt, ist das Verhalten undefiniert.

Returnwert

≠ 0	Sonderzeichen
0	kein Sonderzeichen

Hinweis

`ispunct()` ist sowohl als Makro als auch als Funktion realisiert. Um einen Funktionsaufruf zu erzeugen, muss die Definition des Makronamens rückgängig gemacht werden (`#undef ispunct`).

Das Verhalten von `ispunct()` wird von der Klasse `punct` der aktuellen Lokalität bestimmt. Die aktuelle Lokalität ist die C-Lokalität, wenn nicht explizit mit `setlocale()` umgeschaltet wurde.

Siehe auch `isalnum()`, `isalpha()`, `iscntrl()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `isspace()`, `isupper()`, `isxdigit()`, `setlocale()`, `ctype.h`.

isspace - auf Zwischenraumzeichen prüfen

Syntax

```
#include <ctype.h>

int isspace(int c);
```

Beschreibung

`isspace()` überprüft, ob *c* ein Zwischenraumzeichen ist. Zwischenraumzeichen sind: Leerzeichen, horizontaler Tabulator, Wagenrücklauf, Zeilenvorschub, Seitenvorschub oder vertikaler Tabulator.

In allen Fällen ist das Argument *c* vom Typ *int*. Der Wert von *c* muss als *unsigned char* darstellbar oder gleich dem Wert des Makros `EOF` sein. Wenn *c* irgendeinen anderen Wert besitzt, ist das Verhalten undefiniert.

Returnwert

≠ 0	Zwischenraumzeichen
0	kein Zwischenraumzeichen

Hinweis

`isspace()` ist sowohl als Makro als auch als Funktion realisiert. Um einen Funktionsaufruf zu erzeugen, muss die Definition des Makronamens rückgängig gemacht werden (`#undef isspace`).

Das Verhalten von `isspace()` wird von der Klasse `space` der aktuellen Lokalität bestimmt. Die aktuelle Lokalität ist die C-Lokalität, wenn nicht explizit mit `setlocale()` umgeschaltet wurde.

Siehe auch `isalnum()`, `isalpha()`, `iscntrl()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isupper()`, `isxdigit()`, `setlocale()`, `ctype.h`.

isupper - auf Großbuchstaben prüfen

Syntax #include <ctype.h>

int isupper(int *c*);

Beschreibung

`isupper()` überprüft, ob das Zeichen *c* ein Großbuchstabe ist.

In allen Fällen ist das Argument *c* vom Typ *int*. Der Wert von *c* muss als *unsigned char* darstellbar oder gleich dem Wert des Makros `EOF` sein. Wenn *c* irgendeinen anderen Wert besitzt, ist das Verhalten undefiniert.

Returnwert

≠ 0	Großbuchstabe
0	kein Großbuchstabe

Hinweis `isupper()` ist sowohl als Makro als auch als Funktion realisiert. Um einen Funktionsaufruf zu erzeugen, muss die Definition des Makronamens rückgängig gemacht werden (`#undef isupper`).

Das Verhalten von `isupper()` wird von der Klasse `upper` der aktuellen Lokalität bestimmt. Die aktuelle Lokalität ist die C-Lokalität, wenn nicht explizit mit `setlocale()` umgeschaltet wurde.

Siehe auch `isalnum()`, `isalpha()`, `iscntrl()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isxdigit()`, `setlocale()`, `ctype.h`.

iswalnum - auf alphanumerisches Langzeichen prüfen

Syntax

```
#include <wchar.h>

int iswalnum(wint_t wc);
```

Beschreibung

`iswalnum()` überprüft, ob das Langzeichen *wc* alphanumerisch ist.

In allen Fällen ist das Argument *wc* vom Typ *wint_t*. Der Wert von *wc* muss ein Langzeichenwert sein, der einem gültigen Zeichen in der aktuellen Lokalität entspricht, oder er muss gleich dem Wert des Makros `WEOF` sein. Wenn *wc* irgendeinen anderen Wert besitzt, ist das Verhalten undefiniert.

Returnwert

≠ 0	alphanumerisch
0	nicht alphanumerisch

Hinweis

`iswalnum()` ist sowohl als Makro als auch als Funktion realisiert. Um einen Funktionsaufruf zu erzeugen, muss die Definition des Makronamens rückgängig gemacht werden (`#undef iswalnum`).

Das Verhalten von `iswalnum()` wird von den Klassen `alpha` und `digit` der aktuellen Lokalität bestimmt. Die aktuelle Lokalität ist die C-Lokalität, wenn nicht explizit mit `setlocale()` umgeschaltet wurde.

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Siehe auch

`iswalpha()`, `iswcntrl()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswprint()`, `iswpunct()`, `iswspace()`, `iswupper()`, `iswxdigit()`, `setlocale()`, `wchar.h`, `stdio.h`.

iswalpha - auf alphabetisches Langzeichen prüfen

Syntax #include <wchar.h>

int iswalpha(wint_t *wc*);

Beschreibung

`iswalpha` überprüft, ob das Langzeichen *wc* ein Buchstabe ist.

In allen Fällen ist das Argument *wc* vom Typ *wint_t*. Der Wert von *wc* muss ein Langzeichenwert sein, der einem gültigen Zeichen in der aktuellen Lokalität entspricht, oder er muss gleich dem Wert des Makros `WEOF` sein. Wenn *wc* irgendeinen anderen Wert besitzt, ist das Verhalten undefiniert.

Returnwert

$\neq 0$	Buchstabe
0	kein Buchstabe

Hinweis `iswalpha()` ist sowohl als Makro als auch als Funktion realisiert. Um einen Funktionsaufruf zu erzeugen, muss die Definition des Makronamens rückgängig gemacht werden (#`undef iswalpha`).

Das Verhalten von `iswalpha()` wird von der Klasse `alpha` der aktuellen Lokalität bestimmt. Die aktuelle Lokalität ist die C-Lokalität, wenn nicht explizit mit `setlocale()` umgeschaltet wurde.

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Siehe auch `iswalnum()`, `iswcntrl()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswprint()`, `iswpunct()`, `iswspace()`, `iswupper()`, `iswxdigit()`, `setlocale()`, `wchar.h`, `stdio.h`.

iswcntrl - auf Steuerlangzeichen prüfen

Syntax

```
#include <wchar.h>

int iswcntrl(wint_t wc);
```

Beschreibung

`iswcntrl()` überprüft, ob das Langzeichen *wc* ein Steuerzeichen ist. Steuerzeichen sind nicht abdruckbare Zeichen, z. B. für die Druckersteuerung.

In allen Fällen ist das Argument *wc* vom Typ *wint_t*. Der Wert von *wc* muss ein Langzeichenwert sein, der einem gültigen Zeichen in der aktuellen Lokalität entspricht, oder er muss gleich dem Wert des Makros `WEOF` sein. Wenn *wc* irgendeinen anderen Wert besitzt, ist das Verhalten undefiniert.

Returnwert

≠ 0	Steuerzeichen
0	kein Steuerzeichen

Hinweis

`iswcntrl()` ist sowohl als Makro als auch als Funktion realisiert. Um einen Funktionsaufruf zu erzeugen, muss die Definition des Makronamens rückgängig gemacht werden (`#undef iswcntrl`).

Das Verhalten von `iswcntrl()` wird von der Klasse `cntrl` der aktuellen Lokalität bestimmt. Die aktuelle Lokalität ist die C-Lokalität, wenn nicht explizit mit `setlocale()` umgeschaltet wurde.

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Siehe auch `iswalnum()`, `iswalpha()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswprint()`, `iswpunct()`, `iswspace()`, `iswupper()`, `iswxdigit()`, `setlocale()`, `wchar.h`.

iswctype - Langzeichen auf Klasse prüfen

Syntax #include <wchar.h>

int iswctype(wint_t *wc*, wctype_t *charclass*);

Beschreibung

`iswctype()` überprüft, ob das Langzeichen *wc* zur Zeichenklasse *charclass* gehört.

In allen Fällen ist das Argument *wc* vom Typ *wint_t*. Der Wert von *wc* muss ein Langzeichenwert sein, der einem gültigen Zeichen in der aktuellen Lokalität entspricht, oder er muss gleich dem Wert des Makros `WEOF` sein. Wenn *wc* irgendeinen anderen Wert besitzt, ist das Verhalten undefiniert.

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Returnwert

≠ 0	Langzeichen in Zeichenklasse *charclass*
0	Langzeichen nicht in Zeichenklasse *charclass*

Hinweis Die zwölf Zeichenketten `"alnum"`, `"alpha"`, `"blank"`, `"cntrl"`, `"digit"`, `"graph"`, `"lower"`, `"print"`, `"punct"`, `"space"`, `"upper"` und `"xdigit"` sind für die Standard-Zeichenklassen reserviert. In der folgenden Tabelle sind die Funktionen der linken Spalte mit denen der rechten Spalte jeweils gleichwertig.

```
iswalnum(wc)      iswcty-     wctype("alnum"))
                  pe(wc,
iswalpha(wc)      iswcty-     wctype("alpha"))
                  pe(wc,
iswcntrl(wc)      iswcty-     wctype("cntrl"))
                  pe(wc,
iswdigit(wc)      iswcty-     wctype("digit"))
                  pe(wc,
iswgraph(wc)      iswcty-     wctype("graph"))
                  pe(wc,
iswlower(wc)      iswcty-     wctype("lower"))
                  pe(wc,
iswprint(wc)      iswcty-     wctype("print"))
                  pe(wc,
iswpunct(wc)      iswcty-     wctype("punct"))
                  pe(wc,
```

`iswspace(wc)`	`iswcty-` `pe(wc,`	`wctype("space"))`
`iswupper(wc)`	`iswcty-` `pe(wc,`	`wctype("upper"))`
`iswxdigit(wc)`	`iswcty-` `pe(wc,`	`wctype("xdigit"))`

Der Aufruf `iswctype(wc, wctype("blank"))` hat keine gleichwertige `isw*`–Funktion.

Siehe auch `wctype()`, `iswalnum()`, `iswalpha()`, `iswcntrl()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswprint()`, `iswpunct()`, `iswspace()`, `iswupper()`, `iswxdigit()`, `wchar.h`.

iswdigit - auf dezimales Langzeichen prüfen

Syntax

```
#include <wchar.h>

int iswdigit(wint_t wc);
```

Beschreibung

`iswdigit()` überprüft, ob das Langzeichen *wc* eine Dezimalziffer ist.

In allen Fällen ist das Argument *wc* vom Typ *wint_t*. Der Wert von *wc* muss ein Langzeichenwert sein, der einem gültigen Zeichen in der aktuellen Lokalität entspricht, oder er muss gleich dem Wert des Makros `WEOF` sein. Wenn *wc* irgendeinen anderen Wert besitzt, ist das Verhalten undefiniert.

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Returnwert

≠ 0	Dezimalziffer
0	keine Dezimalziffer

Hinweis

`iswdigit()` ist sowohl als Makro als auch als Funktion realisiert. Um einen Funktionsaufruf zu erzeugen, muss die Definition des Makronamens rückgängig gemacht werden (`#undef iswdigit`).

Das Verhalten von `iswdigit()` wird von der Klasse `digit` der aktuellen Lokalität bestimmt. Die aktuelle Lokalität ist die C-Lokalität, wenn nicht explizit mit `setlocale()` umgeschaltet wurde.

Siehe auch `iswalnum()`, `iswalpha()`, `iswcntrl()`, `iswgraph()`, `iswlower()`, `iswprint()`, `iswpunct()`, `iswspace()`, `iswupper()`, `iswxdigit()`, `wchar.h`.

iswgraph - auf darstellbares Langzeichen prüfen

Syntax #include <wchar.h>

int iswgraph(wint_t *wc*);

Beschreibung

`iswgraph()` überprüft, ob das mit *wc* angegebene Langzeichen ein darstellbares Zeichen ist. Darstellbare Zeichen sind: alphanumerische Zeichen und Sonderzeichen. Leerzeichen gelten als nicht darstellbar.

In allen Fällen ist das Argument *wc* vom Typ *wint_t*. Der Wert von *wc* muss ein Langzeichenwert sein, der einem gültigen Zeichen in der aktuellen Lokalität entspricht, oder er muss gleich dem Wert des Makros `WEOF` sein. Wenn *wc* irgendeinen anderen Wert besitzt, ist das Verhalten undefiniert.

Returnwert

≠ 0	darstellbar
0	nicht darstellbar

Hinweis `iswgraph()` ist sowohl als Makro als auch als Funktion realisiert. Um einen Funktionsaufruf zu erzeugen, muss die Definition des Makronamens rückgängig gemacht werden (`#undef iswgraph`).

Das Verhalten von `iswgraph()` wird von der Klasse `graph` der aktuellen Lokalität bestimmt. Die aktuelle Lokalität ist die C-Lokalität, wenn nicht explizit mit `setlocale()` umgeschaltet wurde.

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Siehe auch `iswalnum()`, `iswalpha()`, `iswcntrl()`, `iswdigit()`, `iswlower()`, `iswprint()`, `iswpunct()`, `iswspace()`, `iswupper()`, `iswxdigit()`, `setlocale()`, `wchar.h`.

iswlower - auf Kleinbuchstaben-Langzeichen prüfen

Syntax

```
#include <wchar.h>

int iswlower(wint_t wc);
```

Beschreibung

`iswlower()` überprüft, ob das Langzeichen *wc* ein Kleinbuchstabe ist.

In allen Fällen ist das Argument *wc* vom Typ *wint_t*. Der Wert von *wc* muss ein Langzeichenwert sein, der einem gültigen Zeichen in der aktuellen Lokalität entspricht, oder er muss gleich dem Wert des Makros `WEOF` sein. Wenn *wc* irgendeinen anderen Wert besitzt, ist das Verhalten undefiniert.

Returnwert

≠ 0	Kleinbuchstabe
0	kein Kleinbuchstabe

Hinweis

`iswlower()` ist sowohl als Makro als auch als Funktion realisiert. Um einen Funktionsaufruf zu erzeugen, muss die Definition des Makronamens rückgängig gemacht werden (`#undef iswlower`).

Das Verhalten von `iswlower()` wird von der Klasse `lower` der aktuellen Lokalität bestimmt. Die aktuelle Lokalität ist die C-Lokalität, wenn nicht explizit mit `setlocale()` umgeschaltet wurde.

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Siehe auch `iswalnum()`, `iswalpha()`, `iswcntrl()`, `iswdigit()`, `iswgraph()`, `iswprint()`, `iswpunct()`, `iswspace()`, `iswupper()`, `iswxdigit()`, `setlocale()`, `wchar.h`.

iswprint - auf druckbares Langzeichen prüfen

Syntax #include <wchar.h>

int iswprint(wint_t *wc*);

Beschreibung

`iswprint()` überprüft, ob *wc* ein druckbares Langzeichen ist. Druckbare Langzeichen sind: alphanumerische Zeichen, Sonderzeichen und Leerzeichen .

In allen Fällen ist das Argument *wc* vom Typ *wint_t*. Der Wert von *wc* muss ein Langzeichenwert sein, der einem gültigen Zeichen in der aktuellen Lokalität entspricht, oder er muss gleich dem Wert des Makros `WEOF` sein. Wenn *wc* irgendeinen anderen Wert besitzt, ist das Verhalten undefiniert.

Returnwert

$\neq 0$	druckbar (alphanumerisches Zeichen, Sonderzeichen oder Leerzeichen).
0	nicht druckbar

Hinweis `iswprint()` ist sowohl als Makro als auch als Funktion realisiert. Um einen Funktionsaufruf zu erzeugen, muss die Definition des Makronamens rückgängig gemacht werden (`#undef iswprint`).

Das Verhalten von `iswprint()` wird von der Klasse `print` der aktuellen Lokalität bestimmt. Die aktuelle Lokalität ist die C-Lokalität, wenn nicht explizit mit `setlocale()` umgeschaltet wurde.

Siehe auch `iswalnum()`, `iswalpha()`, `iswcntrl()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswpunct()`, `iswspace()`, `iswupper()`, `iswxdigit()`, `setlocale()`, `wchar.h`.

iswpunct - auf Sonderlangzeichen prüfen

Syntax #include <wchar.h>

int iswpunct(wint_t *wc*);

Beschreibung

`iswpunct()` überprüft, ob *wc* ein Sonderlangzeichen ist, d.h. weder ein Steuerlangzeichen noch ein alphanumerisches Langzeichen, noch ein Langzeichen für Zwischenraum (siehe `iswspace`).

In allen Fällen ist das Argument *wc* vom Typ *wint_t*. Der Wert von *wc* muss ein Langzeichenwert sein, der einem gültigen Zeichen in der aktuellen Lokalität entspricht, oder er muss gleich dem Wert des Makros `WEOF` sein. Wenn *wc* irgendeinen anderen Wert besitzt, ist das Verhalten undefiniert.

Returnwert

≠ 0	Sonderlangzeichen
0	kein Sonderlangzeichen

Hinweis `iswpunct()` ist sowohl als Makro als auch als Funktion realisiert. Um einen Funktionsaufruf zu erzeugen, muss die Definition des Makronamens rückgängig gemacht werden (`#undef iswpunct`).

Das Verhalten von `iswpunct()` wird von der Klasse `punct` der aktuellen Lokalität bestimmt. Die aktuelle Lokalität ist die C-Lokalität, wenn nicht explizit mit `setlocale()` umgeschaltet wurde.

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Siehe auch `iswalnum()`, `iswalpha()`, `iswcntrl()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswprint()`, `iswspace()`, `iswupper()`, `iswxdigit()`, `setlocale()`, `wchar.h`.

iswspace - auf Zwischenraum-Langzeichen prüfen

Syntax

```
#include <wchar.h>

int iswspace(wint_t wc);
```

Beschreibung

`iswspace()` überprüft, ob *wc* ein Zwischenraum-Langzeichen ist. Zwischenraum-Langzeichen sind: Leerzeichen, horizontaler Tabulator, Wagenrücklauf, Zeilenvorschub, Seitenvorschub oder vertikaler Tabulator.

In allen Fällen ist das Argument *wc* vom Typ *wint_t*. Der Wert von *wc* muss ein Langzeichenwert sein, der einem gültigen Zeichen in der aktuellen Lokalität entspricht, oder er muss gleich dem Wert des Makros `WEOF` sein. Wenn *wc* irgendeinen anderen Wert besitzt, ist das Verhalten undefiniert.

Returnwert

≠ 0	Zwischenraum-Langzeichen
0	kein Zwischenraum-Langzeichen

Hinweis

`iswspace()` ist sowohl als Makro als auch als Funktion realisiert. Um einen Funktionsaufruf zu erzeugen, muss die Definition des Makronamens rückgängig gemacht werden (`#undef iswspace`).

Das Verhalten von `iswspace()` wird von der Klasse `space` der aktuellen Lokalität bestimmt. Die aktuelle Lokalität ist die C-Lokalität, wenn nicht explizit mit `setlocale()` umgeschaltet wurde.

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Siehe auch `iswalnum()`, `iswalpha()`, `iswcntrl()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswprint()`, `iswpunct()`, `iswupper()`, `iswxdigit()`, `setlocale()`, `wchar.h`.

iswupper - auf Großbuchstaben-Langzeichen prüfen

Syntax

```
#include <wchar.h>

int iswupper(wint_t wc);
```

Beschreibung

`iswupper()` überprüft, ob das Langzeichen *wc* ein Großbuchstabe ist.

In allen Fällen ist das Argument *wc* vom Typ *wint_t*. Der Wert von *wc* muss ein Langzeichenwert sein, der einem gültigen Zeichen in der aktuellen Lokalität entspricht, oder er muss gleich dem Wert des Makros `WEOF` sein. Wenn *wc* irgendeinen anderen Wert besitzt, ist das Verhalten undefiniert.

Returnwert

$\neq 0$	Großbuchstabe
0	kein Großbuchstabe

Hinweis

`iswupper()` ist sowohl als Makro als auch als Funktion realisiert. Um einen Funktionsaufruf zu erzeugen, muss die Definition des Makronamens rückgängig gemacht werden (`#undef iswupper`).

Das Verhalten von `iswupper()` wird von der Klasse `upper` der aktuellen Lokalität bestimmt. Die aktuelle Lokalität ist die C-Lokalität, wenn nicht explizit mit `setlocale()` umgeschaltet wurde.

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Siehe auch `iswalnum()`, `iswalpha()`, `iswcntrl()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswprint()`, `iswpunct()`, `iswspace()`, `iswxdigit()`, `setlocale()`, `wchar.h`.

iswxdigit - auf Hexadezimal-Langzeichen prüfen

Syntax

```
#include <wchar.h>

int iswxdigit(wint_t wc);
```

Beschreibung

`iswxdigit` überprüft, ob das Langzeichen *wc* ein hexadezimales Ziffernzeichen (0-9, A-F bzw. a-f) ist.

In allen Fällen ist das Argument *wc* vom Typ *wint_t*. Der Wert von *wc* muss ein Langzeichenwert sein, der einem gültigen Zeichen in der aktuellen Lokalität entspricht, oder er muss gleich dem Wert des Makros `WEOF` sein. Wenn *wc* irgendeinen anderen Wert besitzt, ist das Verhalten undefiniert.

Returnwert

≠ 0	hexadezimales Ziffernzeichen
0	kein hexadezimales Ziffernzeichen

Hinweis

`iswxdigit()` ist sowohl als Makro als auch als Funktion realisiert. Um einen Funktionsaufruf zu erzeugen, muss die Definition des Makronamens rückgängig gemacht werden (`#undef iswxdigit`).

Das Verhalten von `iswxdigit()` wird von der Klasse `xdigit` der aktuellen Lokalität bestimmt. Die aktuelle Lokalität ist die C-Lokalität, wenn nicht explizit mit `setlocale()` umgeschaltet wurde.

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Siehe auch `iswalnum()`, `iswalpha()`, `iswcntrl()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswprint()`, `iswpunct()`, `iswspace()`, `iswupper()`, `wchar.h`.

isxdigit - auf Hexadezimal-Ziffer prüfen

Syntax #include <ctype.h>

int isxdigit(int *c*);

Beschreibung

`isxdigit` überprüft, ob das Zeichen *c* ein hexadezimales Ziffernzeichen (0-9, A-F bzw. a-f) ist.

In allen Fällen ist das Argument *c* vom Typ *int*. Der Wert von *c* muss als *unsigned char* darstellbar oder gleich dem Wert des Makros `EOF` sein. Wenn *c* irgendeinen anderen Wert besitzt, ist das Verhalten undefiniert.

Returnwert

≠ 0	hexadezimale Ziffer
0	keine hexadezimale Ziffer

Hinweis `isxdigit()` ist sowohl als Makro als auch als Funktion realisiert. Um einen Funktionsaufruf zu erzeugen, muss die Definition des Makronamens rückgängig gemacht werden (`#undef isxdigit`).

Das Verhalten von `isxdigit()` wird von der Klasse `xdigit` der aktuellen Lokalität bestimmt. Die aktuelle Lokalität ist die C-Lokalität, wenn nicht explizit mit `setlocale()` umgeschaltet wurde.

Siehe auch `isalnum()`, `isalpha()`, `iscntrl()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `ctype.h`.

j0, j1, jn - Besselfunktionen der ersten Art anwenden

Syntax #include <math.h>

double j0(double *x*);
double j1(double *x*);
double jn(int *n*, double *x*);

Beschreibung

`j0()`, `j1()` und `jn()` berechnen die Besselfunktionen der ersten Art für Gleitpunktwerte *x* und die ganzzahligen Ordnungen 0, 1 bzw. *n*.

Returnwert Besselfunktion für *x* bei Erfolg.

Siehe auch `y0()`, `y1()`, `yn()`, `math.h`.

jrand48 - Pseudo-Zufallszahlen zwischen -2^{31} und 2^{31} mit Startwert generieren

Syntax #include <stdlib.h>

long int jrand48 (unsigned short int *xsubi*[3]);

Beschreibung

Siehe `drand48()`.

kill - Signal an Prozess oder Prozessgruppe senden

Syntax

#include <signal.h>

Optional
#include <sys/types.h> ❑

int kill(pid_t *pid*, int *sig*);

Beschreibung

Wenn die Funktion mit POSIX-Funktionalität aufgerufen wird, verhält sie sich XPG5-konform, wie folgt:

- `kill()` sendet ein Signal *sig* an den Prozess oder die Prozessgruppe, der bzw. die durch *pid* angegeben wird. *sig* ist entweder eines der in der Datei `signal.h` angegebenen Signale oder gleich 0. Wenn *sig* gleich 0 ist (Nullsignal), wird eine Fehlerüberprüfung durchgeführt, ohne dass ein Signal gesendet wird. Das Nullsignal kann verwendet werden, um die Gültigkeit von *pid* zu überprüfen.
- `{_POSIX_SAVED_IDS}` ist auf allen X/Open-konformen Systemen definiert. Damit ein Prozess ein Signal an den durch *pid* bezeichneten Prozess senden kann, muss die reale oder effektive Benutzernummer des sendenden Prozesses mit der realen oder gesicherten Benutzernummer des empfangenden Prozesses übereinstimmen, vorausgesetzt der sendende Prozess hat geeignete Zugriffsrechte.
- Wenn *pid* größer als 0 ist, wird *sig* an den Prozess gesendet, dessen Prozessnummer gleich *pid* ist.
- Wenn *pid* gleich 0 ist, wird *sig* an alle Prozesse (außer einer Anzahl von Systemprozessen) gesendet, deren Prozessgruppennummer gleich der Prozessgruppennummer des Senders ist und für die der Prozess die Erlaubnis hat, ein Signal zu senden.
- Wenn *pid* gleich -1 ist, wird *sig* an alle Prozesse gesendet, für die der Prozess die Erlaubnis hat, ein Signal zu senden, außer den Systemprozessen.
- Wenn *pid* negativ, aber ungleich -1 ist, wird *sig* an alle Prozesse gesendet, deren Prozessgruppenummer gleich dem Absolutbetrag von *pid* ist, und für die der Prozess die Erlaubnis hat, ein Signal zu senden.
- Wenn durch den Wert von *pid* für den sendenden Prozess *sig* generiert wird und wenn *sig* nicht blockiert ist, wird, bevor `kill()` zurückkehrt, entweder *sig* oder zumindest ein anstehendes, nicht blockiertes Signal an den sendenden Prozess zugestellt.
- Die Benutzernummer wird nicht überprüft, wenn das Signal `SIGCONT` an einen Prozess gesendet wird, der Mitglied derselben Sitzung ist, wie der sendende Prozess.
- `kill()` ist erfolgreich, wenn der Prozess die Erlaubnis hat, *sig* an einen der durch *pid* angegebenen Prozesse zu senden. Wenn `kill()` fehlschlägt, wird kein Signal gesendet.

Werden Threads verwendet, so wirkt sich die Funktion auf den Prozess oder auf einen Thread wie folgt aust:

– Senden eines Signals an einen Prozess oder eine Prozessgruppe;
Für den (Spezial-)Fall, dass der Wert von *pid* bewirkt, dass *sig* für den sendenden Prozess generiert wird, gilt: Wenn das Signal für den aufrufenden Thread nicht blockiert ist und wenn alle anderen Threads des Prozesses das Signal blockieren bzw. nicht in einer `sigwait()`-Funktion auf das Signal warten, wird entweder *sig* oder wenigstens ein anhängiges nicht blockiertes Signal dem sendenden Thread zugestellt, bevor `kill()` zurückkehrt.

– *BS2000*
Wenn die Funktion mit BS2000-Funktionalität aufgerufen wird, verhält sie sich abweichend, wie folgt:

– *pid* muss gleich 0 sein. Dadurch wird das Signal an den aufrufenden Prozess gesendet.

– Für *sig* kann folgende Untermenge der Signale, die in `signal.h` definiert sind, eingesetzt werden:

Signal	STXIT-Klasse	Bedeutung
SIGHUP	ABEND	Abbruch der Dialogstationsleitung
SIGINT	ESCPBRK	Unterbrechung von der Dialogstation mit K2
SIGILL	PROCHK	Ausführung einer ungültigen Instruktion
SIGABRT	–	raise-Signal für Programmbeendigung mit `_exit(-1)`
SIGFPE	PROCHK	fehlerhafte Gleitpunktoperation
SIGKILL	–	raise-Signal für Programmbeendigung mit `exit(-1)`
SIGSEGV	ERROR	Speicherzugriff mit unerlaubtem Segmentzugriff
SIGALRM	RTIMER	ein Zeitintervall ist abgelaufen (Realzeit)
SIGTERM	TERM	Signal bei Programmbeendigung
SIGUSR1	–	vom Benutzer definiert
SIGUSR2	–	vom Benutzer definiert
SIGDVZ	PROCHK	Division durch 0
SIGXCPU	RUNOUT	CPU-Zeit ist aufgebraucht
SIGTIM	TIMER	ein Zeit-Intervall ist abgelaufen (CPU-Zeit)
SIGINTR	INTR	SEND-MESSAGE-Kommando

❑

Returnwert 0 bei erfolgreicher Beendigung.

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

Fehler `kill()` schlägt fehl, wenn gilt:

`EINVAL` Der Wert des Arguments *sig* ist eine ungültige oder nicht unterstützte Signalnummer.

EPERM	Der Prozess besitzt keine Erlaubnis, das Signal an einen empfangenden Prozess zu senden. *BS2000* EPERM wird nicht unterstützt. ❑
ESRCH	Es kann kein Prozess oder keine Prozessgruppe gefunden werden, die der durch *pid* angegebenen entspricht.

Siehe auch getpid(), raise(), setsid(), sigaction(), signal.h, sys/types.h.

killpg - Signal an Prozessgruppe senden

Syntax #include <signal.h>

int killpg(pid_t *pgrp*, int *sig*);

Beschreibung

`killpg()` sendet das Signal *sig* an die Prozessgruppe *pgrp*.

Die reale oder effektive Benutzer-ID des sendenden Prozesses muss mit der realen oder gesicherten „set-user-ID“ des empfangenden Prozesses übereinstimmen, sofern die effektive Benutzer-ID des sendenden Prozesses nicht von einem Benutzer mit entsprechender Berechtigung stammt. Die einzige Ausnahme bildet das Signal `SIGCONT`, das immer an jeden Nachfolger des aktuellen Prozesses gesendet werden kann.

Ist *pgrp* größer als 1, so entspricht `killpg(`*pgrp*, *sig*`)` dem Aufruf von `kill(-`*pgrp*,*sig*`)`.
Ist *pgrp* kleiner als oder gleich 1, so ist das Verhalten von `killpg()` undefiniert.

Returnwert Siehe `kill()`.

Fehler Siehe `kill()`.

Siehe auch `getpgid()`, `getpid()`, `kill()`, `raise()`, `signal.h`.

l64a - 32-Bit-Integerzahl in Zeichenkette umwandeln

Syntax #include <stdlib.h>

char *l64a (long *value*);

Beschreibung

Siehe `a64l()`.

labs - ganzzahligen Absolutwert (long) berechnen

Syntax #include <stdlib.h>

long int labs(long int *j*);

Beschreibung

`labs()` berechnet den Absolutwert einer ganzen Zahl *j* vom Typ `long`.

Returnwert Absolutwert für einen ganzzahligen Wert *j* bei Erfolg.

Hinweis Der Absolutwert der betragsmäßig größten darstellbaren negativen Zahl ist nicht darstellbar. Wird als Argument *i* die betragsmäßig größte negative Zahl (-2^{31}) als Parameter angegeben, wird das Programm mit Fehler beendet.

Siehe auch `abs()`, `cabs()`, `stdlib.h`.

lchown - Eigentümer/Gruppe einer Datei ändern

Syntax

#include <unistd.h>

int lchown(const char **path*, uid_t *owner*, gid_t *group*);

Beschreibung

Die Funktion `lchown()` setzt den Eigentümer und die Gruppenzugehörigkeit der angegebenen Datei genau wie `chown()`, es sei denn, die Datei besteht aus einem symbolischen Verweis. In diesem Fall ändert `lchown()` die Zugehörigkeit der Verweisdatei, wohingegen `chown()` die Zugehörigkeit der Datei oder des Verzeichnisses ändert, auf das sich der Verweis bezieht.

Wenn `chown()`, `lchown()` oder `fchown()` von einem Prozess aufgerufen wird, der nicht den Systemverwalterstatus hat, dann wird das Bit zum Setzen der Benutzer- und Gruppennummer bei Ausführung, beziehungsweise `S_ISUID` und `S_ISGID`, gelöscht (siehe `chmod()`).

Das Betriebssystem hat die Konfigurationsoption `_POSIX_CHOWN_RESTRICTED`, um Zugehörigkeitsänderungen für `chown()`-, `lchown()`- und `fchown()`-Systemaufrufe zu verhindern. In POSIX ist `_POSIX_CHOWN_RESTRICTED` aktiv, daher bewahren die `chown()`-, `lchown()`- und `fchown()`-Systemaufrufe den Eigentümer einer Datei davor, dass die Eigentümernummern seiner Dateien geändert werden und beschränken den Gruppenwechsel der Datei auf die Liste der ergänzenden Gruppennummern.

Nach erfolgreichem Abschluss markieren `chown()`,`lchown()` und `fchown()` das `ST_CTIME`-Feld der Datei zum Aktualisieren.

Returnwert

0	bei Erfolg.
-1	bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen. Bei Rückgabe von -1 werden Benutzernummer und Gruppennummer der Datei nicht verändert.

Fehler

`lchown()` schlägt fehl, wenn gilt:

`EACCES`	Eine Komponente von *path* darf nicht durchsucht werden.
`EINVAL`	Der Wert der angegebenen Benutzer- oder Gruppennummer wird nicht unterstützt, z.B. wenn der Wert kleiner als 0 ist, oder es wurde versucht, auf eine BS2000-Datei zuzugreifen.

ENAMETOOLONG
: Die Länge des Pfadnamens überschreitet {PATH_MAX}, oder die Länge einer Komponente des Pfadnamens überschreitet {NAME_MAX}.

ENOENT
: Eine Komponente des Pfadnamens existiert nicht, oder *path* zeigt auf eine leere Zeichenkette.

ENOTDIR
: Eine Komponente des Pfadnamen-Präfix ist kein Dateiverzeichnis.

EOPNOTSUPP
: Das Argument *path* bezeichnet einen symbolischen Verweis und die Implementierung unterstützt nicht, den Eigentümer oder die Gruppenzugehörigkeit eines symbolischen Verweises zu ändern.

ELOOP
: Während der Übersetzung von *path* wurden zu viele symbolische Verweise angetroffen.

EPERM
: Die effektive Benutzernummer entspricht nicht dem Eigentümer der Datei, und der aufrufende Prozess hat nicht die passenden Zugriffsrechte.

EROFS
: Die Datei steht in einem schreibgeschützten Dateisystem.

EIO
: Es trat während des Lesens oder Schreibens vom Dateisystem ein Ein- oder Ausgabefehler auf.

EINTR
: Ein Signal wurde während der Ausführung der Funktion abgefangen.

Erweiterung
ENAMETOOLONG
: Die Auflösung symbolischer Verweise im Pfadnamen führt zu einem Zwischenergebnis, dessen Länge {PATH_MAX} überschreitet.

Siehe auch chmod(), chown(), symlink(), unistd.h.

lcong48 - Pseudo-Zufallszahlen (signed long int) generieren

Syntax #include <stdlib.h>

void lcong48 (unsigned short int *param*[7]);

Beschreibung

Siehe `drand48()`.

ldexp - Exponent einer Gleitpunktzahl laden

Syntax #include <math.h>

double ldexp(double *x*, int *exp*);

Beschreibung

`ldexp()` berechnet aus der Mantisse *x* und dem Exponenten *exp* die Größe:
x * 2*exp* .

`ldexp()` ist die Umkehrfunktion zu `frexp()`.

Returnwert Wert der Größe *x** 2*exp*
bei Erfolg.

`+/-HUGE_VAL` (je nach Vorzeichen von *x*), bei Überlauf. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `ldexp()` schlägt fehl, wenn gilt:

`ERANGE` Überlauf.

Siehe auch `frexp()`, `modf()`, `math.h`.

ldiv - ganze Zahl (long) dividieren

Syntax #include <stdlib.h>

ldiv_t ldiv(long int *numer*, long int *denom*);

Beschreibung

`ldiv()` berechnet den Quotienten und den Rest der Division *numer* / *denom*.

Sowohl die Argumente als auch das Ergebnis sind vom Typ `long int`.

Das Vorzeichen des Quotienten ist gleich dem Vorzeichen des algebraischen Quotienten. Die Größe des Quotienten ist die größte ganze Zahl kleiner oder gleich dem absoluten Wert des algebraischen Quotienten.

Der Rest ergibt sich aus der Gleichung:

Quotient * Divisor + Rest = Dividend

Returnwert Struktur vom Typ `ldiv_t`
bei Erfolg. Die Struktur enthält sowohl den Quotienten `quot` als auch den Rest `rem` als long-Werte.

Siehe auch `div()`, `stdlib.h`.

lfind - Eintrag in linearer Datentabelle finden

Syntax #include <search.h>

void *lfind(const void **key*, const void **base*, size_t **nelp*, size_t *width*
int (**compar*) (const void *, const void *))

Beschreibung

Siehe `lsearch()`.

lgamma - Logarithmus der Gamma-Funktion berechnen

Syntax

```
#include <math.h>

double lgamma(double x);

extern int signgam;
```

Beschreibung

`lgamma()` berechnet die mathematische Gammafunktion für die Gleitpunktzahl x: a

$$\int_0^{\infty} e^{-t} \; t^{x-1} \; dt$$

Das Vorzeichen dieses Wertes wird in der C-internen Variablen `signgam` als +1 oder -1 abgelegt. `signgam` darf nicht vom Anwender definiert werden.

Returnwert

`lgamma(x)`	bei Erfolg.
`HUGE_VAL`	falls der korrekte Wert einen Überlauf ergibt. `errno` wird gesetzt, um den Fehler anzuzeigen.
`HUGE_VAL`	falls x eine nichtpositive Ganzzahl ist. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `lgamma()` schlägt fehl, wenn gilt:

`ERANGE`	Überlauf, das Resultat ist zu groß.
`EDOM`	x ist eine nichtpositive Ganzzahl.

Siehe auch `gamma()`, `math.h`.

_ _LINE_ _ - Makro für aktuelle Quellprogramm-Zeilennummer

Syntax

```
_ _LINE_ _
```

Beschreibung

Dieses Makro generiert die aktuelle Zeilennummer des Quellprogramms als Dezimalzahl.

Hinweis Dieses Makro muss in keiner Include-Datei definiert werden. Sein Name wird vom Compiler erkannt und ersetzt.

link - Verweis auf eine Datei erzeugen

Syntax

```
#include <unistd.h>

int link(const char *path1, const char *path2);
```

Beschreibung

`link()` erzeugt einen neuen Verweis (Dateiverzeichniseintrag) für die existierende Datei *path1*.

path1 zeigt auf einen Pfadnamen, der eine existierende Datei benennt. *path2* zeigt auf einen Pfadnamen, der den neuen, zu erzeugenden Dateiverzeichniseintrag benennt. Die Funktion `link()` erzeugt automatisch einen neuen Verweis für die existierende Datei und der Verweiszähler dieser Datei wird um 1 erhöht.

Wenn *path1* ein Dateiverzeichnis benennt, schlägt `link()` fehl.

Bei erfolgreicher Beendigung kennzeichnet `link()` die Strukturkomponente `st_ctime` der Datei zum Aktualisieren. Ebenso werden `st_ctime` und `st_mtime` des Dateiverzeichnisses, das den neuen Eintrag enthält, zum Aktualisieren gekennzeichnet.

Wenn die Funktion `link()` fehlschlägt, wird kein Verweis erzeugt und der Verweiszähler der Datei bleibt unverändert.

Der aufrufende Prozess muss das Zugriffsrecht auf die existierende Datei haben.

`link()` wird nicht zwischen Dateien verschiedener Dateisysteme durchgeführt.

Wenn bei einem erfolgreichen Aufruf von `link(*path1, *path2)` sowohl *path1* als auch *path2* auf Dateien des POSIX-Dateisystems zeigen, wird ein interner Verweiszähler um 1 erhöht. Bei einem erfolgreichen Aufruf von `unlink(*path)` oder `remove(*path)` wird dieser Verweiszähler um 1 vermindert. Ist dieser Zähler = 0 und die Datei nicht mehr von einem Prozess geöffnet, wird die Datei gelöscht.

Returnwert

0	bei Erfolg
-1	wenn der Prozess, der `link()` aufruft, auf die betreffende Datei nicht zugreifen darf. `errno` wird auf `EACCES` gesetzt, um den Fehler anzuzeigen. Bei einem Aufruf mit einer BS2000-Datei (`/BS2/`*name*) wird `errno` auf `EINVAL` gesetzt.

Fehler

`link()` schlägt fehl, wenn gilt:

`EACCES`	Für eine Komponente des Pfades existiert kein Durchsuchrecht oder der geforderte Verweis verlangt das Schreiben in ein Dateiverzeichnis mit Zugriffsrechten, die das Schreibrecht verweigern. Oder der aufrufende Prozess besitzt nicht das Recht, auf die existierende Datei zuzugreifen.
`EEXIST`	Der durch *path2* benannte Verweis existiert.

Erweiterung

EFAULT *path1* oder *path2* weist über den zugewiesenen Adressraum hinaus.

EINTR Ein Signal wurde während des Systemaufrufs `link()` abgefangen.

EINVAL Es wurde versucht, auf eine BS2000-Datei zuzugreifen.

ELOOP Beim Übersetzen von *path1* oder *path2* waren zu viele symbolische Verweise vorhanden. ❑

EMLINK Die Anzahl der Verweise auf die durch *path1* benannte Datei würde {LINK_MAX} überschreiten.

ENAMETOOLONG
Die Länge von *path1* oder *path2* überschreitet {PATH_MAX}, oder eine Pfadnamenkomponente ist länger als {NAME_MAX}.

ENOENT Eine Komponente eines der Pfade oder die durch *path1* benannte Datei existiert nicht, oder *path1* oder *path2* zeigt auf eine leere Zeichenkette.

ENOSPC Das den Verweis enthaltende Dateiverzeichnis kann nicht erweitert werden.

ENOTDIR Eine Komponente eines der Pfade ist kein Dateiverzeichnis.

EPERM Die durch *path1* benannte Datei ist ein Dateiverzeichnis, und der Prozess besitzt keine Sonderrechte.

EROFS Der gewünschte Verweis erfordert das Schreiben in einem Dateiverzeichnis auf einem nur zum Lesen eingehängten Dateisystem.

EXDEV Der durch *path2* benannte Verweis und die durch *path1* benannte Datei befinden sich auf verschiedenen Dateisystemen.

Hinweis `link()` wird nur für POSIX-Dateien ausgeführt.

Siehe auch `readlink()`, `remove`, `symlink()`, `unlink()`, `unistd.h`.

llabs - Absolutbetrag einer ganzen Zahl (long long int)

Syntax #include <stdlib.h>

long long int llabs(long long int *j*);

Beschreibung

`llabs()` berechnet den Absolutbetrag einer ganzen Zahl *j* vom Typ `long long int`.

Returnwert

`|j|`	für einen ganzzahligen Wert *j*.
undefiniert	bei Über- oder Unterlauf. `errno` wird auf `ERANGE` gesetzt, um den Fehler anzuzeigen.

Fehler `llabs()` schlägt fehl, wenn gilt:

`ERANGE`	Der Absolutbetrag der betragsmäßig größten negativen Zahl des Typs `long long int` ist nicht darstellbar. Wenn als Argument *j* die betragsmäßig größte negative Zahl angegeben wird, wird das Programm mit Fehler beendet.

Siehe auch `abs(), cabs(), labs()`

lldiv - Division mit ganzen Zahlen (long long int)

Syntax #include <stdlib.h>

lldiv_t lldiv(long long int *dividend*, long long int *divisor*);

Beschreibung

`lldiv()` berechnet den Quotienten und den Rest der Division *dividend* durch *divisor*. Sowohl die Argumente als auch das Ergebnis sind vom Typ `long long int`.

Das Vorzeichen des Quotienten ist gleich dem Vorzeichen des algebraischen Quotienten. Die Größe des Quotienten ist die größte ganze Zahl kleiner oder gleich dem absoluten Wert des algebraischen Quotienten.

Der Rest ergibt sich aus der Gleichung

Quotient * Divisor + Rest = Dividend

Returnwert Struktur vom Typ `lldiv_t`,
die sowohl den Quotienten *quot* als auch den Rest *rem* als `long long`-Werte enthält.

Siehe auch `div()`, `ldiv()`

llrint, llrintf, llrintl - auf nächste ganze Zahl runden (long long int)

Syntax #include <math.h>

long long int llrint(double *x*);

long long int llrintf (float *x*);

long long int llrintl (long double *x*);

Beschreibung

Die Funktionen geben jeweils die ganze Zahl zurück, die *x* am nächsten liegt - dargestellt als Zahl vom Typ `long long int`.

Der zurückgegebene Wert ist entsprechend dem aktuell gesetzten Rundungsmodus des Rechners gerundet. Wenn der Rundungsmodus 'round-to-nearest' gesetzt ist und die Differenz zwischen *x* und dem gerundeten Ergebnis genau 0.5 ist, wird die nächste gerade Ganzzahl zurückgegeben.

Wenn der aktuell eingestellte Rundungsmodus in Richtung positiv unendlich rundet, ist `llrint()` äquivalent zu `ceil()`. Wenn der aktuell eingestellte Rundungsmodus in Richtung negativ unendlich rundet, ist `llrint()` äquivalent zu `floor()`.

In dieser Version ist der Rundungsmodus fest auf Richtung positiv unendlich eingestellt.

Returnwert

ganze Zahl	dargestellt als Zahl vom Typ `long long int` bei Erfolg.
undefiniert	bei Über- oder Unterlauf. `errno` wird auf `ERANGE` gesetzt, um den Fehler anzuzeigen.

Fehler `llrint(), llrintf(), llrintl()` schlagen fehl, wenn gilt:

`ERANGE`	Der Wert ist zu groß. `errno` wird gesetzt, um den Fehler anzuzeigen.

Siehe auch `abs(), ceil(), floor(), llround(), lrint(), lround(), rint(), round()`

llround, llroundf, llroundl - auf nächste ganze Zahl runden (long long int)

Syntax

```
#include <math.h>

long long int llround(double x);

long long int llroundf (float x);

long long int llroundl (long double x);
```

Beschreibung

Die Funktionen geben jeweils die ganze Zahl zurück, die *x* am nächsten liegt, dargestellt als Zahl vom Typ `long long int`.

Der zurückgegebene Wert ist unabhängig vom eingestellten Rundungsmodus. Wenn die Differenz zwischen *x* und dem gerundeten Ergebnis genau 0.5 ist, wird die betragsmäßig größere ganze Zahl zurückgegeben.

Returnwert

ganze Zahl	dargestellt als Zahl vom Typ `long long int`. bei Erfolg.
undefiniert	bei Über- oder Unterlauf. `errno` wird auf `ERANGE` gesetzt, um den Fehler anzuzeigen.

Fehler

`llround(), llroundf(), llroundl()` schlagen fehl, wenn gilt:

`ERANGE`	Der Wert ist zu groß. `errno` wird gesetzt, um den Fehler anzuzeigen.

Siehe auch `abs(), ceil(), floor(), llrint(), lrint(), lround(), rint(), round()`

loc1, loc2 - Zeiger beim Vergleich von regulären Ausdrücken verwenden

Syntax #include <regexp.h>

extern char **loc1*;
extern char **loc2*;

Beschreibung

Siehe `regexp()`.

Hinweis Diese Funktion wird zukünftig vom X/Open-Standard nicht mehr unterstützt.

In neuen Anwendungen sollten die Funktionen `fnmatch()`, `glob()`, `regcomp()` und `regexec()` verwendet werden. Sie garantieren die volle Funktionalität für internationalisierte reguläre Ausdrücke (siehe „Reguläre Ausdrücke“ im Handbuch „POSIX Kommandos (BS2000/OSD)“).

localeconv - Lokalitätskomponenten ändern

Syntax include <locale.h>

struct lconv *localeconv(void);

Beschreibung

`localeconv()` versieht die Komponenten einer Struktur vom Typ `struct lconv` (definiert in `locale.h`) mit Formatierungswerten für numerische Größen (monetäre und andere Größen) entsprechend der aktuellen Lokalität.

Die *`char`-Komponenten der Struktur `lconv` sind Zeiger auf Zeichenketten, von denen jeder außer `decimal_point` auf eine leere Zeichenkette `""` zeigen kann; dadurch wird angegeben, dass der Wert in der aktuellen Lokalität nicht definiert ist oder die Länge null hat.

Die `char`-Komponenten der Struktur `lconv` sind nichtnegative Zahlen, von denen jede den Wert `{CHAR_MAX}` annehmen kann (siehe `limits.h`); dadurch wird angegeben, dass der Wert in der aktuellen Lokalität nicht verfügbar ist.

Die Komponenten für nichtmonetäre numerische Werte (`LC_NUMERIC`) haben folgende Bedeutungen:

`char *decimal_point`
: Dezimalzeichen zur Formatierung nichtmonetärer Größen.

`char *thousands_sep`
: Trennzeichen zwischen Zifferngruppen links vom Dezimalpunkt zur Formatierung nichtmonetärer Größen.

`char *grouping`
: Zeichenkette, deren Elemente, wenn sie als Ein-Byte-Wert vom Typ integer behandelt werden, die Größe jeder Zifferngruppe in nichtmonetärer Größe angeben (siehe auch unten).

Die Komponenten für monetäre Werte (`LC_MONETARY`) haben folgende Bedeutungen:

`char *int_curr_symbol`
: Internationales Währungssymbol, das für die aktuelle Lokalität verwendet wird. Der Operand ist eine Zeichenkette aus vier Zeichen. Die ersten drei Zeichen bilden das internationale Währungssymbol, wie bei ISO 4217:1987 festgelegt. Das vierte Zeichen, das unmittelbar vor dem Nullbyte steht, ist das Trennzeichen zwischen Währungssymbol und monetärer Größe.
In der Lokalität "`De.EDF04F@euro`" ist der Wert “EUR“ als alphabetisches Währungssymbol eingetragen.

`char *currency_symbol`
: Lokales Währungssymbol, das für die aktuelle Lokalität verwendet wird.

`char *mon_decimal_point`
: Dezimalzeichen für die Formatierung von monetären Größen. Im ISO-C Standard ist diese Komponente auf ein Byte beschränkt. Wenn ein Multibyte-Operand spezifiziert wird, ist das Ergebnis unbestimmt.

`char *mon_thousands_sep`
: Trennzeichen für Zifferngruppen links vom Dezimalpunkt in formatierten, monetären Größen. Im ISO-C Standard ist diese Komponente auf ein Byte beschränkt. Wenn ein Multibyte-Operand spezifiziert wird, ist das Ergebnis unbestimmt.

`char *mon_grouping`
: Zeichenkette, deren Elemente, wenn sie als ganzzahlige Ein-Byte-Werte betrachtet werden, die Größe jeder Zifferngruppe in formatierten, monetären Größen anzeigen. Der Operand ist eine Folge ganzer Zahlen, die durch Semikolon voneinander getrennt sind. Jede Zahl gibt die Anzahl der Stellen in jeder Gruppe an, wobei die erste Zahl die Größe der Gruppe angibt, die direkt vor dem Dezimaltrennzeichen steht, und die folgenden Zahlen die vorangehenden Gruppen bestimmen. Wenn die letzte Zahl ungleich -1 ist, wird die vorhergehende Gruppe (falls es eine gibt) für den Rest der Stellen immer wieder verwendet. Wenn die letzte Zahl -1 ist, wird keine weitere Gruppierung durchgeführt (siehe auch unten).

`char *positive_sign`
: Zeichenkette, die eine nichtnegative, formatierte, monetäre Größe anzeigt.

`char *negative_sign`
: Zeichenkette, die eine negative, formatierte, monetäre Größe anzeigt.

`char int_frac_digits`
: Anzahl der Dezimalstellen, die in international, formatierten, monetären Größen angezeigt werden, wobei `int_curr_symbol` verwendet wird.

`char frac_digits`
: Anzahl der Dezimalstellen, die in einer formatierten, monetären Größe dargestellt werden, wobei `currency_symbol` verwendet wird.

`char p_cs_precedes`
: Wird auf 1 gesetzt, wenn `currency_symbol` oder `int_curr_symbol` dem Wert für eine monetäre Größe mit einem nichtnegativen Wert vorangehen, und wird auf 0 gesetzt, wenn eines dieser Symbole auf den Wert folgt.

`char p_sep_by_space`
: Wird auf 0 gesetzt, wenn kein Leerzeichen das `currency_symbol` oder `int_curr_symbol` vom Wert einer nichtnegativen, formatierten, monetä-

ren Größe trennt. Die Komponente wird auf 1 gesetzt, wenn ein Leerzeichen zwischen Symbol und Wert steht; sie wird auf 2 gesetzt, wenn ein Leerzeichen zwischen dem Symbol und einer angrenzenden Zeichenkette steht.

`char n_cs_precedes`
: Wenn diese Komponente den Wert 1 hat, wird das Währungssymbol `currency_symbol` oder `int_curr_symbol` vor den Wert einer negativen, formatierten, monetären Größe geschrieben. Sonst wird die Komponente auf 0 gesetzt.

`char n_sep_by_space`
: Wird auf 0 gesetzt, wenn kein Leerzeichen das `currency_symbol` oder `int_curr_symbol` vom Wert einer negativen, formatierten, monetären Größe trennt. Die Komponente wird auf 1 gesetzt, wenn ein Leerzeichen zwischen Symbol und Wert steht, und sie wird auf 2 gesetzt, wenn ein Leerzeichen zwischen dem Symbol und einer angrenzenden Zeichenkette steht.

`char p_sign_posn`
: Diese Komponente wird auf einen Wert gesetzt, der die Position des positiven Vorzeichens `positive_sign` für eine nichtnegative, formatierte, monetäre Größe angibt (siehe auch unten).

`char n_sign_posn`
: Wird auf einen Wert gesetzt, der die Position des negativen Vorzeichens `negative_sign` für eine negative, formatierte, monetäre Größe angibt (siehe auch unten).

Die Elemente von `grouping` und `mon_grouping` werden wie folgt interpretiert:

`CHAR_MAX` Keine weitere Gruppierung wird durchgeführt.

`0` Das vorherige Element wird für die restlichen Ziffern wiederholt verwendet.

other Der Wert ist die Anzahl der Ziffern, welche sich in der aktuellen Gruppe befinden. Das nächste Element wird überprüft, um die Größe der nächsten Zifferngruppe links von der aktuellen Gruppe zu bestimmen.

Die Werte von `p_sign_posn` und `n_sign_posn` werden wie folgt interpretiert:

`0` Größe und Währungssymbol `currency_symbol` oder `int_curr_symbol` werden in Klammern gesetzt.

`1` Das Vorzeichen steht vor der Größe und dem Währungssymbol `currency_symbol` oder `int_curr_symbol`.

`2` Das Vorzeichen steht hinter der Größe und dem Währungssymbol `currency_symbol` oder `int_curr_symbol`.

3 Das Vorzeichen steht direkt vor dem Währungssymbol `currency_symbol` oder `int_curr_symbol`.

4 Das Vorzeichen steht direkt hinter dem Währungssymbol `currency_symbol` oder `int_curr_symbol`.

Die Implementierung verhält sich, als ob keine Funktion `localeconv()` aufruft.

Returnwert Zeiger auf die Struktur, in die die Werte eingetragen wurden bei erfolgreicher Beendigung.

Hinweis Die Struktur, auf die der Returnwert zeigt, darf nicht durch das Programm verändert werden, kann aber durch einen weiteren Aufruf `localeconv()` überschrieben werden. Außerdem können `setlocale`-Aufrufe mit den Kategorien `LC_ALL`, `LC_MONETARY` oder `LC_NUMERIC` den Inhalt der Struktur überschreiben.

Beispiel Die folgende Tabelle demonstriert die Regeln zur Formatierung monetärer Größen anhand von drei Ländern:

Land	**Positives Format**	**Negatives Format**	**Internationales Format**
Deutschland	`EUR 1.234,56`	`-EUR 1.234,56`	`EUR 1.234,56`
Norwegen	`kr1.234,56`	`kr1.234,56-`	`NOK 1.234,56`
Schweiz	`SFrs.1,234.56`	`SFrs.1,234.56C`	`CHF 1,234.56`

Für diese drei Länder werden die entsprechenden Werte für die monetären Komponenten von `localeconv()` wie folgt zurückgegeben:

Komponentenwerte	**Deutschland**	**Norwegen**	**Schweiz**
`int_curr_symbol`	`"EUR"`	`"NOK "`	`"CHF "`
`currency_symbol`	`"?"`	`"kr"`	`"SFrs."`
`mon_decimal_point`	`","`	`","`	`"."`
`mon_thousands_sep`	`"."`	`"."`	`","`
`mon_grouping`	`3;3`	`"\3"`	`"\3`
`positive_sign`	`""`	`""`	`""`
`negative_sign`	`"-"`	`"-"`	`"C"`
`int_frac_digits`	2	2	2
`frac_digits`	2	2	2
`p_cs_precedes`	0	1	1
`p_sep_by_space`	1	0	0
`n_cs_precedes`	0	1	1

Komponentenwerte	Deutschland	Norwegen	Schweiz
n_sep_by_space	1	0	0
p_sign_posn	1	1	1
n_sign_posn	1	2	2

Siehe auch `isalpha()`, `isascii()`, `nl_langinfo()`, `printf()`, `scanf()`, `setlocale()`, `strcat()`, `strchr()`, `strcmp()`, `strcoll()`, `strcpy()`, `strftime()`, `strlen()`, `strpbrk()`, `strspn()`, `strtok()`, `strxrfm()`, `strtod()`, `langinfo.h`, `local.h`, Abschnitt „Lokalität“ auf Seite 52.

localtime - Datum und Uhrzeit in Ortszeit umwandeln

Syntax #include <time.h>

struct tm *localtime(const time_t **clock*);

Beschreibung

`localtime()` akzeptiert Argumente vom Typ `time_t`, auf die *clock* zeigt und die die Zeit in Sekunden seit 00:00:00 UTC (Universal Time Coordinated, 1. Januar 1970) enthalten.

`localtime()` gibt Zeiger auf `tm`-Strukturen zurück, die nachstehend erläutert werden. `localtime()` berücksichtigt Zeitzonen und eventuelle Sommerzeit-Korrekturen.

In der Include-Datei `time.h` sind die Vereinbarungen aller Funktionen und externer Werte sowie der `tm`-Struktur enthalten. Die Strukturvereinbarung ist wie folgt:

```
struct      tm {
    int     tm_sec;         /* Sekunden - [0, 61] für übersprungene Sek.*/
    int     tm_min;         /* Minuten - [0, 59] */
    int     tm_hour;        /* Stunden - [0, 23] */
    int     tm_mday;        /* Tag des Monats - [1, 31] */
    int     tm_mon;         /* Monate - [0, 11] */
    int     tm_year;        /* Jahre seit 1900 */
    int     tm_wday;        /* Tage seit Sonntag - [0, 6] */
    int     tm_yday;        /* Tage seit dem 1. Januar - [0, 365] */
    int     tm_isdst;       /* Option für Sommerzeit */
};
```

`tm_isdst` ist positiv, wenn Sommerzeit eingestellt ist, null, wenn Sommerzeit nicht eingestellt ist, und negativ, wenn die Information nicht verfügbar ist.

`localtime()` ist nicht threadsicher. Verwenden Sie bei Bedarf die reentrante Funktion `localtime_r()`.

BS2000

`localtime()` interpretiert die Zeitangabe vom Typ `time_t` als Anzahl der Sekunden, die seit dem 1. Januar 1950 00:00:00 vergangen sind. `localtime()` berechnet daraus Datum und Uhrzeit und speichert das Ergebnis in einer Struktur vom Typ `tm`.

`localtime()` entspricht in dieser Implementierung `gmtime()`, beide liefern die lokale Zeit. ❑

Returnwert Zeiger auf `tm`-Struktur

Hinweis Die Returnwerte von `localtime()` zeigen auf statische Daten, die bei jedem Aufruf überschrieben werden.

`localtime()` unterstützt nicht die lokalen Datums- und Zeit-Formate. Um maximale Portabilität zu erreichen, sollte `strftime()` verwendet werden.

`localtime()` schreibt sein Ergebnis in einen C-internen Datenbereich, der bei jedem Aufruf überschrieben wird.

Außerdem verwenden `localtime()` und `gmtime()` denselben Datenbereich, d.h., wenn sie hintereinander aufgerufen werden, wird das Ergebnis des ersten Aufrufs überschrieben.

Siehe auch `altzone`, `ctime()`, `daylight`, `gmtime()`, `localtime_r()`, `strftime()`, `tzname`, `tzset()`, `time.h`.

localtime_r - Datum und Uhrzeit threadsicher in Zeichenkette umwandeln

Syntax

#include <time.h>

struct tm *localtime_r(const time_t **clock*, struct tm **result*);

Beschreibung

`localtime_r()` wandelt den Zeitwert, auf den *clock* zeigt, in genau dieselbe Zeitform wie `localtime()` um und schreibt das Ergebnis in den Speicherbereich, auf den *result* zeigt (mit zumindest 26 Bytes).

Returnwert

Zeiger auf die Zeichenkette, auf die *result* zeigt,	bei Erfolg.
Nullzeiger	bei Fehler.

Siehe auch `asctime()`, `asctime_r()`, `ctime()`, `ctime_r()`, `localtime()`, `time()`.

lockf - Dateiabschnitt sperren

Name **lockf, lockf64**

Syntax #include <unistd.h>

int lockf(int *fildes*, int *function*, off_t *size*);
int lockf64(int *fildes*, int *function*, off64_t *size*);

Beschreibung

Mit `lockf()` können Dateiabschnitte gesperrt werden; dabei hängen empfohlene oder obligatorische Schreibsperren jeweils von den Modusbits der Datei ab (siehe `chmod()`). Sperraufrufe von anderen Prozessen, die versuchen, einen bereits gesperrten Dateiabschnitt zu sperren, führen entweder zur Rückgabe eines Fehlerwerts oder pausieren solange, bis das Betriebsmittel freigegeben wird. Alle Sperren für einen Prozess werden aufgehoben, wenn der Prozess beendet wird. `lockf()` kann auf normale Dateien angewendet werden.

fildes ist ein offener Dateideskriptor. Der Dateideskriptor muss die `O_WRONLY`- oder `O_RDWR`-Erlaubnis haben, damit die Sperre mit diesem Funktionsaufruf eingerichtet werden kann.

function ist ein Steuerwert, der die zu treffenden Maßnahmen angibt. Die zulässigen Werte für *function* sind, wie folgt, in `unistd.h` definiert:

```
#define    F_ULOCK   0   /* gesperrten Abschnitt freigeben */
#define    F_LOCK    1   /* Abschnitt exklusiv sperren */
#define    F_TLOCK   2   /* Abschnitt testen und exklusiv sperren */
#define    F_TEST    3   /* Abschnitt auf Sperren anderer Prozesse testen */
```

Alle anderen Werte von *function* sind für zukünftige Erweiterungen reserviert und führen zu einer Fehlermeldung, wenn sie nicht implementiert sind.

`F_TEST` wird verwendet, um festzustellen, ob in einem Abschnitt eine Sperre eines anderen Prozesses existiert. `F_LOCK` und `F_TLOCK` sperren jeweils einen Abschnitt einer Datei, wenn dieser Abschnitt verfügbar ist. `F_ULOCK` hebt die Sperren eines Dateiabschnitts auf.

size ist die Anzahl zusammenhängender Bytes, die gesperrt oder entsperrt werden sollen. Das zu sperrende oder entsperrende Betriebsmittel beginnt am aktuellen Offset in der Datei und erstreckt sich vorwärts für ein positives *size* und rückwärts für ein negatives *size* (die vorhergehenden Bytes bis ausschließlich des aktuellen Offsets). Wenn *size* null ist, wird der Abschnitt vom aktuellen Offset bis zum größten Datei-Offset gesperrt, d.h. vom aktuellen Offset bis zum gegenwärtigen oder bis zu jedem zukünftigen Dateiende. Ein Bereich braucht nicht einer Datei zugewiesen sein, damit er gesperrt werden kann, weil diese Sperren auch über das Ende der Datei hinausgehen können.

Die mit F_LOCK oder F_TLOCK gesperrten Abschnitte können einen vorher von demselben Prozess gesperrten Abschnitt ganz oder teilweise enthalten bzw. in diesem Abschnitt enthalten sein. Wenn diese Situation in diesem oder in benachbarten Abschnitten eintritt, werden die Abschnitte zu einem Abschnitt zusammengefasst. Wenn mit der Anforderung ein neues Element zur Tabelle der aktiven Sperren hinzugefügt werden muss und diese Tabelle bereits voll ist, erfolgt eine Fehlermeldung, und der neue Abschnitt wird nicht gesperrt.

Die Anforderungen von F_LOCK und F_TLOCK unterscheiden sich nur in der Maßnahme, die getroffen wird, wenn das Betriebsmittel nicht zur Verfügung steht. F_LOCK bewirkt, dass der aufrufende Prozess pausiert, bis das Betriebsmittel zur Verfügung steht. F_TLOCK bewirkt, dass die Funktion -1 zurückgibt und errno auf den Fehler EACCES setzt, wenn der Abschnitt bereits von einem anderen Prozess gesperrt ist.

Gesperrte Abschnitte werden durch den ersten close-Aufruf freigegeben, den der Prozess, der die Sperre gesetzt hat, für einen Dateideskriptor der zugehörigen Datei durchführt.

F_ULOCK-Anforderungen können einen oder mehrere gesperrte, vom Prozess gesteuerte Abschnitte teilweise oder ganz freisetzen. Gesperrte Abschnitte werden ab dem Punkt des Offsets entsperrt, bis *size* Bytes entsperrt worden sind oder bis zum Dateiende, wenn *size* den Wert (off_t)0 hat. Wenn die Abschnitte nicht ganz entsperrt werden, bleiben die übrigen Abschnitte weiterhin vom Prozess gesperrt. Die Freigabe des mittleren Abschnitts eines gesperrten Abschnitts erfordert einen zusätzlichen Eintrag in der Tabelle der aktiven Sperren. Wenn diese Tabelle voll ist, wird errno auf ENOLK gesetzt und der angeforderte Abschnitt nicht freigegeben.

Die Möglichkeit eines Deadlocks entsteht, wenn ein Prozess, der ein gesperrtes Betriebsmittel kontrolliert, durch Anforderung des gesperrten Betriebsmittels eines anderen Prozesses zum Pausieren veranlasst wird. Daher wird bei Aufruf von lockf() oder fcntl() zunächst auf mögliche Deadlocks geprüft, bevor der Prozess bis zur Freigabe eines noch gesperrten Betriebsmittels angehalten wird. Wenn das Warten auf ein gesperrtes Betriebsmittel einen Deadlock verursachen würde, schlägt der Aufruf fehl und errno wird auf EDEADLK gesetzt.

Das gleichzeitige Sperren mit lockf() und fcntl() führt zu undefinierten Wechselwirkungen.

Das Warten auf ein Betriebsmittel wird mit einem beliebigen Signal unterbrochen. Der Systemaufruf alarm() kann für die Bereitstellung einer Zeitsperre bei Anwendungen verwendet werden, die eine derartige Einrichtung benötigen.

Es besteht kein funktionaler Unterschied zwischen lock() und lock64(), außer dass lockf64() die Größe des zu sperrenden Bereichs in einem Offset-Typ off64_t angibt.

Werden Threads verwendet, so wirkt sich die Funktion auf den Prozess oder auf einen Thread wie folgt aus:

Sperren eines Dateiabschnitts; Sperraufrufe von anderen Threads, die versuchen, einen bereits gesperrten Dateiabschnitt zu sperren, führen entweder zur Rückgabe eines Fehlerwerts oder blockieren den aufrufenden Thread solange, bis der Abschnitt freigegeben wird. Alle Sperren für einen Prozess werden aufgehoben, wenn der Prozess beendet wird.

Returnwert

0	bei Erfolg.
-1	bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen. Bestehende Sperren werden nicht geändert.

Fehler `lockf() und lockf64()` schlagen fehl, wenn gilt:

`EBADF`	*fildes* ist kein gültiger, offener Dateideskriptor, oder *function* ist `F_LOCK` oder `F_TLOCK` und die über *fildes* angesprochene Datei ist nicht zum Schreiben geöffnet.
`EACCES`	*function* ist `F_TLOCK` oder `F_TEST`, und der Abschnitt ist bereits von einem anderen Prozess gesperrt.
`EDEADLK`	*function* ist `F_LOCK` und ein Deadlock würde auftreten.
`EINTR`	Während der Ausführung der Funktion wurde ein Signal abgefangen.
`EAGAIN`	*function* ist `F_LOCK` oder `F_TLOCK` und die Datei wurde mit `mmap()` erzeugt.
`ENOLCK`	*function* ist `F_LOCK`, `F_TLOCK` oder `F_ULOCK`, und der Speicherplatz reicht für weitere Einträge in der Sperrtabelle nicht mehr aus.
`EINVAL`	*fildes* zeigt auf einen Dateityp, der in dieser Implementierung nicht gesperrt werden kann oder der Inhalt von *function* ist ungültig; oder die Summe von *size* plus dem aktuellen Datei-Offset ist kleiner 0 oder größer als der höchste zulässige Datei-Offset.
`ECOMM`	*fildes* ist auf einem fernen Rechner, und die Verbindung zu diesem Rechner ist nicht mehr aktiv.
`EOVERFLOW`	Der Offset des ersten Byte oder, wenn die Größe ungleich 0 ist, des letzten Byte im angeforderten Abschnitt, kann in einem Objekt des Typs `off_t` nicht korrekt dargestellt werden.

Hinweis Unerwartete Ergebnisse können in Prozessen auftreten, die im Adressraum des Benutzers puffern. Der Prozess kann später Daten lesen oder schreiben, die gesperrt sind bzw. waren. Das Standard-E/A-Paket ist die häufigste Ursache für unerwartete Pufferungen. Es sollten statt dessen einfachere Funktionen verwendet werden, die ungepuffert arbeiten, wie z.B `open()`.

Da die Variable `errno` in Zukunft auf `EAGAIN` und nicht auf `EACCES` gesetzt wird, wenn ein Dateiabschnitt bereits von einem anderen Prozess gesperrt ist, müssen portable Anwenderprogramme beide Werte erwarten und prüfen.

Die Funktion `alarm()` kann verwendet werden, um einen eventuellen Timeout zu überwachen.

Siehe auch `alarm()`, `chmod()`, `close()`, `creat()`, `fcntl()`, `mmap()`, `open()`, `read()`, `write()`, `unistd.h`.

locs - Vergleich von regulären Ausdrücken in Zeichenketten anhalten

Syntax #include <regexp.h>

extern char *locs;

Beschreibung

Siehe `regexp()`.

Hinweis Diese Funktion wird zukünftig vom X/Open-Standard nicht mehr unterstützt.

In neuen Anwendungen sollten die Funktionen `fnmatch()`, `glob()`, `regcomp()` und `regexec()` verwendet werden. Sie garantieren die volle Funktionalität für internationalisierte reguläre Ausdrücke (siehe „Reguläre Ausdrücke“ im Handbuch „POSIX Kommandos (BS2000/OSD)“).

log - natürlichen Logarithmus berechnen

Syntax #include <math.h>

double log(double x);

Beschreibung

`log()` berechnet den natürlichen Logarithmus von der positiven Gleitpunktzahl x zur Basis e.

Returnwert

ln(x)	für positive x.
`-HUGE_VAL`	falls x kleiner oder gleich 0 ist. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `log()` schlägt fehl, wenn gilt:

`EDOM`	Der Wert von x ist negativ.
`ERANGE`	Der Wert von x ist gleich 0.

Siehe auch `exp()`, `log10()`, `pow()`, `sqrt()`, `math.h`.

log10 - Logarithmus zur Basis 10 berechnen

Syntax #include <math.h>

double log10(double *x*);

Beschreibung

`log10()` berechnet den Logarithmus von der positiven Gleitpunktzahl *x* zur Basis 10.

Returnwert

lg(*x*)	für positive *x*.
`-HUGE_VAL`	falls *x* kleiner oder gleich 0 ist. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `log10()` schlägt fehl, wenn gilt:

`EDOM`	Der Wert von *x* ist negativ.
`ERANGE`	Der Wert von *x* ist gleich 0.

Siehe auch `exp()`, `log()`, `pow()`, `sqrt()`, `math.h`.

log1p - natürlichen Logarithmus berechnen

Syntax #include <math.h>

double log1p (double *x*);

Beschreibung

Die Funktion `log1p()` berechnet $\log_e (1.0 + x)$, wobei *x* größer als -1.0 sein muss.

Returnwert

ln (1.0 + *x*)	bei Erfolg.
`-HUGE_VAL`	falls $x \leq -1.0$.

Fehler `log1p()` schlägt fehl, wenn gilt:

`EDOM`	Der Wert von *x* ist kleiner als -1.0.

Siehe auch `log()`, `math.h`.

logb - Exponententeil einer Gleitpunktzahl ermitteln

Syntax #include <math.h>

double logb(double *x*);

Beschreibung

.`logb()` ist identisch zu `ilogb()`, außer dass `logb()` den Exponententeil von *x* nicht als `int`, sondern als doppelt genaue, vorzeichenbehaftete Gleitpunktzahl zurückliefert.

Returnwert Exponententeil von *x*
bei Erfolg

`-HUGE_VAL` für *x* = 0.0. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `logb()` schlägt fehl, wenn gilt:

`EDOM` Der Wert von *x* ist 0.0.

Siehe auch `ilogb()`, `math.h`.

_longjmp, _setjmp - Nicht lokaler Sprung (ohne Signalmaske)

Syntax

```
#include <setjmp.h>

void _longjmp(jmp_buf env, int val);

int _setjmp(jmp_buf env);
```

Beschreibung

Die Funktionen `_longjmp()` und `_setjmp()` sind identisch zu `longjmp()` bzw. `setjmp()`, außer dass sie die Signalmaske unverändert lassen.

Wenn `_lngjmp()` aufgerufen wird, ohne dass *env* zuvor von `_setjmp()` initialisiert wurde oder wenn der letzte Aufruf von `_setjmp()` in einer Funktion lag, die mittlerweile schon zurückgekehrt ist, so ist das Verhalten undefiniert.

Returnwert Siehe `longjmp()` und `setjmp()`.

Hinweis Es können Fehler auftreten, wenn `_longjmp()` ausgeführt wird und die Umgebung, in der `_setjmp()` ausgeführt wurde, nicht mehr existiert. Die Umgebung des `_setjmp()`-Aufrufs existiert dann nicht mehr, wenn sich die Funktion beendet, die den Aufruf enthält, oder die Save Area mit den automatic-Variablen verlässt. Möglicherweise wird dieser Fehler nicht entdeckt, was dazu führt, dass `_longjmp()` ausgeführt wird. In diesem Fall ist der Inhalt der Save Area unvorhersehbar. Dieser Fehler kann auch dazu führen, dass sich der Prozess beendet. Wenn die Funktion zurückgekehrt ist, ist das Ergebnis undefiniert.

Wenn an `longjmp()`, `_longjmp()` oder `siglongjmp()` ein Zeiger auf einen Bereich übergeben wird, der nicht von `setjmp()`, `_setjmp()` bzw. `sigsetjmp()` erzeugt, oder wenn der Bereich vom Benutzer verändert wurde, können die oben beschriebenen Fehler sowie zusätzliche Probleme auftreten.

`_longjmp()` und `_setjmp()` werden aus Kompatibilitätsgründen angeboten. Neue Anwendungen sollten `siglongjmp()` bzw. `sigsetjmp()` verwenden.

Siehe auch `longjmp()`, `setjmp()`, `siglongjmp()`, `sigsetjmp()`, `setjmp.h`.

longjmp - nichtlokalen Sprung ausführen

Syntax #include <setjmp.h>

void longjmp(jmp_buf *env*, int *val*);

Beschreibung

`longjmp()` ist nur zusammen mit `setjmp()` anwendbar: Der Aufruf von `longjmp()` bewirkt, dass das Programm an eine zuvor mit `setjmp()` gespeicherte Stelle verzweigt. Im Unterschied zu `goto`-Sprüngen, die nur innerhalb derselben Funktion (also lokal) zulässig sind, erlaubt `longjmp` Sprünge von einer beliebigen Funktion in eine andere, noch aktive Funktion (nicht lokale Sprünge).

`setjmp()` speichert die aktuelle Prozessumgebung (Adresse im C-Laufzeitstack, Befehlszähler, Registerinhalte) in eine Variable vom Typ `jmp_buf` (siehe `setjmp.h`). `longjmp()` stellt die durch `setjmp()` gesicherte Prozessumgebung wieder her, und der Prozess wird mit der Anweisung fortgesetzt, die unmittelbar auf den `setjmp`–Aufruf folgt.

Wenn es vor dem `longjmp`-Aufruf keinen `setjmp`-Aufruf gab oder wenn die Funktion, die den Aufruf von `setjmp()` enthält, inzwischen ihre Ausführung beendet hat, ist das Verhalten undefiniert.

env ist der Vektor, in den `setjmp()` seine Werte abgelegt hat (siehe `setjmp.h`).

val ist eine ganze Zahl, die bei der Rückkehr des Prozesses als Returnwert des `setjmp`-Aufrufs interpretiert wird. Wenn *val* gleich 0 ist, liefert `setjmp()` den Wert 1 zurück; 0 würde bedeuten, dass an die Stelle nach dem `setjmp`-Aufruf „normal“, d.h. nicht mit `longjmp()` verzweigt wurde (siehe auch `setjmp()`).

Alle zugreifbaren Objekte besitzen die Werte, die sie zum Zeitpunkt des Aufrufs von `longjmp()` besaßen, mit Ausnahme der Werte von automatischen Objekten. Diese sind unter folgenden Bedingungen undefiniert:

– Sie sind lokal zu der Funktion, die den entsprechenden `setjmp`-Aufruf enthält.

– Sie sind nicht vom Typ `volatile`.

– Sie wurden zwischen dem `setjmp`- und dem `longjmp`-Aufruf geändert.

Da `longjmp()` den üblichen Funktionsaufruf- und Rückkehrmechanismus umgeht, arbeitet `longjmp()` im Zusammenhang mit Unterbrechungen, Signalen und den zugehörigen Funktionen korrekt. Trotzdem ist das Verhalten undefiniert, wenn `longjmp()` von einer geschachtelten Signalbehandlungsfunktion aus aufgerufen wird (d.h. von einer Funktion aus, die als Ergebnis eines Signals während der Behandlung eines anderen Signals aufgerufen wurde).

Nach der Beendigung von `longjmp()` setzt die Programmausführung fort, als ob der entsprechende Aufruf von `setjmp()` soeben den durch *val* angegebenen Wert geliefert hätte. `longjmp()` kann `setjmp()` nicht veranlassen, den Wert 0 zurückzugeben. Wenn *val* gleich 0 ist, gibt `setjmp()` 1 zurück.

Das Ergebnis eines Aufrufs dieser Funktion ist undefiniert, wenn die Struktur `jmp_buf` nicht im aufrufenden Thread initialisiert wurde.

Die Struktur jmp_buf muss durch setjmp() initialisiert werden. Bei Threads kommt dazu, dass dies im selben Thread passieren muss.

Hinweis Nicht lokale Sprünge sind nützlich bei der Unterbrechungsbehandlung (siehe `signal()`). Erfolgt z.B. die Behandlung von Fehlern oder Unterbrechungen in Routinen auf niedriger Stufe (d.h. es sind eine Reihe zuvor aufgerufener Funktionen noch aktiv), lässt sich mit `longjmp()` und `setjmp()` die normale Abarbeitung der noch aktiven Funktionen umgehen und sofort zu einer Funktion auf höherer Ebene verzweigen. Ein `longjmp`-Aufruf aus einer Unterbrechungs- oder Fehlerroutine leert die Einträge im Laufzeitstack bis zu der durch `setjmp()` markierten Stelle, d.h. alle bis dahin noch aktiven Funktionen auf niedrigerer Ebene sind nicht mehr aktiv und das Programm wird auf höherer Ebene fortgesetzt.

Beim Wiederaufsetzen der Programmausführung sind die Variablen wie nach einem `goto` belegt: Globale Variablen haben die Werte, die sie zum Zeitpunkt des `longjmp`-Aufrufs hatten. Register- und sonstige lokale Variablen sind undefiniert, d.h. sie sollten überprüft und ggf. neu belegt werden.

Siehe auch `setjmp()`, `sigaction()`, `siglongjmp()`, `sigsetjmp()`, `setjmp.h`.

lrand48 - Pseudo-Zufallszahlen zwischen 0 und 2^{31} generieren

Syntax #include <stdlib.h>

long int lrand48 (void);

Beschreibung

Siehe `drand48()`.

lrint, lrintf, lrintl - auf nächste ganze Zahl runden (long int)

Syntax

```
#include <math.h>

long int lrint(double x);

long int lrintf (float x);

long int lrintl (long double x);
```

Beschreibung

Die Funktionen geben jeweils die ganze Zahl zurück, die *x* am nächsten liegt - dargestellt als Zahl vom Typ `long int`.

Der zurückgegebene Wert ist entsprechend dem aktuell gesetzten Rundungsmodus des Rechners gerundet. Wenn der Rundungsmodus 'round-to-nearest' gesetzt ist und die Differenz zwischen *x* und dem gerundeten Ergebnis genau 0.5 ist, wird die nächste gerade Ganzzahl zurückgegeben.

Wenn der aktuell eingestellte Rundungsmodus in Richtung positiv unendlich rundet, ist `lrint()` äquivalent zu `ceil()`. Wenn der aktuell eingestellte Rundungsmodus in Richtung negativ unendlich rundet, ist `lrint()` äquivalent zu `floor()`.

In dieser Version ist der Rundungsmodus fest auf Richtung positiv unendlich eingestellt.

Returnwert

ganze Zahl	dargestellt als Zahl vom Typ `long int` bei Erfolg.
undefiniert	bei Über- oder Unterlauf. `errno` wird auf `ERANGE` gesetzt, um den Fehler anzuzeigen.

Siehe auch `abs()`, `ceil()`, `floor()`, `llrint()`, `llround()`, `lround()`, `rint()`, `round()`

lround, lroundf, lroundl - auf nächste ganze Zahl runden (long int)

Syntax

```
#include <math.h>

long int lround (double x);

long int lroundf (float x);

long int lroundl (long double x);
```

Beschreibung

Die Funktionen geben jeweils die ganze Zahl zurück, die x am nächsten liegt, dargestellt als Zahl vom Typ `long int`.

Der zurückgegebene Wert ist unabhängig vom eingestellten Rundungsmodus. Wenn die Differenz zwischen x und dem gerundeten Ergebnis genau 0.5 ist, wird die betragsmäßig größere ganze Zahl zurückgegeben.

Returnwert

ganze Zahl	dargestellt als Zahl vom Typ `long int`. bei Erfolg.
undefiniert	bei Über- oder Unterlauf. `errno` wird auf `ERANGE` gesetzt, um den Fehler anzuzeigen.

Siehe auch `abs()`, `ceil()`, `floor()`, `llrint()`, `llround()`, `lrint()`, `rint()`, `round()`

lsearch, lfind - linear suchen und aktualisieren

Syntax

```
#include <search.h>

void *lsearch (const void *key, void * base, size_t *nelp,
               size_t width, int (*compar) (const void *, const void *));

void *lfind (const void *key, const void *base, size_t *nelp,
             size_t width, int (*compar)(const void *, const void *));
```

Beschreibung

`lsearch()` ist eine lineare Suchfunktion. Sie gibt einen Zeiger in eine Tabelle zurück, der die Stelle angibt, an der ein gesuchter Wert gefunden wurde. Wenn der gesuchte Wert nicht auftritt, wird er am Ende der Tabelle eingetragen. *key* zeigt auf den Wert, der in der Tabelle gesucht werden soll. *base* zeigt auf das erste Element in der Tabelle. *nelp* zeigt auf eine ganze Zahl, die die aktuelle Anzahl der Elemente in der Tabelle enthält. Die Zahl wird erhöht, wenn der Wert zur Tabelle hinzugefügt wird. *width* ist die Größe eines Elements in Bytes. *compar* ist ein Zeiger auf die Vergleichsfunktion, die der Benutzer zur Verfügung stellen muss (`strcmp()` zum Beispiel). Es werden zwei Argumente erwartet, die auf die Elemente zeigen, die verglichen werden. Die Funktion muss 0 zurückgeben, wenn die Elemente gleich sind, sonst ungleich 0.

`lfind()` wirkt wie `lsearch()`, wobei jedoch der gesuchte Wert nicht zur Tabelle hinzugefügt wird, wenn er nicht gefunden wird. Stattdessen wird ein Nullzeiger zurückgegeben.

Returnwert

**key*	`lfind()`: bei Erfolg. `lsearch()`: bei Erfolg und auch bei neu eingefügtem Element.
Nullzeiger	`lfind()`: bei Fehler.

Hinweis

Die Vergleichsfunktion muss nicht jedes Byte vergleichen, und so können die Elemente zusätzlich zu den zu vergleichenden Werten beliebige Daten enthalten.

Undefinierte Ergebnisse können auftreten, wenn nicht genügend Speicherplatz für ein neues Element vorhanden ist.

Erweiterung
Die Zeiger auf den Schlüssel und das Element an der Basis der Tabelle können Zeiger auf einen beliebigen Typ sein.

Der zurückgegebene Wert sollte sich in den Typ Zeiger-auf-Element umwandeln lassen. ❑

Siehe auch `bsearch()`, `hsearch()`, `tsearch()`, `search.h`.

lseek - Lese-/Schreibzeiger in Datei auf aktuellen Wert positionieren

Name **lseek, lseek64**

Syntax

Optional
#include <sys/types.h>
#include <unistd.h>

off_t lseek (int *fildes*, off_t *offset*, int *whence*);
off64_t lseek64 (int *fildes*, off64_t *offset*, int *whence*);

Beschreibung

Wenn POSIX-Dateien ausgeführt werden, verhält sich die Funktion XPG-konform wie folgt:

`lseek()` setzt den Lese-/Schreibzeiger für die Datei mit dem Dateideskriptor *fildes* wie nachfolgend beschrieben:

Ist *whence* gleich `SEEK_SET`, wird der Zeiger gleich *offset* Bytes gesetzt.

Ist *whence* gleich `SEEK_CUR`, wird der Zeiger auf die aktuelle Position plus *offset* gesetzt.

Ist *whence* gleich `SEEK_END`, wird der Zeiger auf die Größe der Datei plus *offset* gesetzt.

Die symbolischen Konstanten `SEEK_SET`, `SEEK_CUR` und `SEEK_END` sind in der Include-Datei `unistd.h` definiert.

Die Funktion `lseek()` hat keine Wirkung, wenn sie auf eine Datei angewendet wird, auf der nicht positioniert werden kann.

`lseek()` erlaubt, dass der Lese-/Schreibzeiger hinter die existierenden Daten der Datei gesetzt werden kann. Werden später Daten an diese Position geschrieben, so liefern nachfolgende Leseoperationen in der Lücke Nullbytes, bis wirklich Daten in diese Lücke geschrieben wurden.

`lseek()` erweitert nicht von sich aus die Größe einer Datei.

Es besteht kein funktionaler Unterschied zwischen `lseek()` und `lseek64()`, außer dass `lseek64()` den Offset-Typ `off64_t` verwendet.

BS2000
Wenn BS2000-Dateien ausgeführt werden, ist Folgendes zu beachten:

`lseek()` positioniert den Lese-/Schreibzeiger für die Datei mit Dateideskriptor *fildes* gemäß den Angaben in *offset* und *whence*. Damit ist die Möglichkeit gegeben, eine Datei nicht-sequenziell zu bearbeiten. Als Ergebnis liefert `lseek()` die aktuelle Position in der Datei.

Textdateien (SAM, ISAM) lassen sich absolut auf Dateianfang und -ende positionieren sowie auf eine vorher mit `tell()` gemerkte Position.

Binärdateien (PAM, INCORE) lassen sich sowohl absolut positionieren (s.o.) als auch relativ um eine gewünschte Anzahl Bytes, bezogen auf Dateianfang, Dateiende oder aktuelle Position.
SAM-Dateien werden mit elementaren Funktionen stets als Textdateien verarbeitet.

Bedeutung, Kombinationsmöglichkeiten und Wirkung von *offset* und *whence* sind für Text- und Binärdateien unterschiedlich und werden deshalb im Folgenden getrennt beschrieben.

Textdateien (SAM, ISAM)

Mögliche Werte:

offset `0L` oder Wert, der durch einen vorhergehenden `tell/lseek`-Aufruf ermittelt wurde.

whence `SEEK_SET` (Dateianfang)
`SEEK_CUR` (aktuelle Position)
`SEEK_END` (Dateiende)

Sinnvolle Kombinationsmöglichkeiten und Wirkung:

offset	*whence*	**Wirkung**
`tell/lseek`-Wert	`SEEK_SET`	Positionieren auf die durch `tell()` oder `lseek()` gemerkte Position.
`0L`	`SEEK_SET`	Positionieren auf Dateianfang.
`0L`	`SEEK_CUR`	Abfrage der aktuellen Position ohne Positionierung.
`0L`	`SEEK_END`	Positionieren auf Dateiende.

Binärdateien (PAM, INCORE)

Mögliche Werte:

offse Anzahl der Bytes, um die der aktuelle Lese-/Schreibzeiger verschoben werden soll, und zwar

–- positive Zahl: Vorwärts positionieren Richtung Dateiende

–- negative Zahl: Rückwärts positionieren Richtung Dateianfang

–- `0L`: absolut Positionieren auf Dateianfang bzw. -ende.

whence Bei absoluter Positionierung auf Dateianfang oder -ende, Zielpunkt, auf den der Lese-/Schreibzeiger verschoben werden soll und
bei relativer Positionierung, Bezugspunkt, von dem aus der Lese-/Schreibzeiger um *offset* Bytes verschoben werden soll:
`SEEK_SET` (Dateianfang)
`SEEK_CUR` (aktuelle Position)
`SEEK_END` (Dateiende)

Sinnvolle Kombinationsmöglichkeiten und Wirkung:

offset	*whence*	**Wirkung**
`0L`	`SEEK_SET`	Positionieren auf Dateianfang.
`0L`	`SEEK_CUR`	Abfrage der aktuellen Position ohne Positionierung.
`0L`	`SEEK_END`	Positionieren auf Dateiende.

positive Zahl	`SEEK_SET`
	`SEEK_CUR`
	`SEEK_END`
		Vorwärts positionieren ab Dateianfang,
		ab aktueller Position,
		ab Dateiende (über das Dateiende hinaus).

negative Zahl	`SEEK_CUR` `SEEK_END`	Rückwärts positionieren ab aktueller Position, ab Dateiende.
`tell`/`lseek`-Wert	`SEEK_SET`	Positionieren auf die durch einen `tell()` oder `lseek`-Aufruf gemerkte Position.

Returnwert neuer Wert des Lese-/Schreibzeigers, gemessen in Bytes vom Anfang der Datei,
bei Erfolg.

(off_t) -1 bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen. Der Wert des Lese-/Schreibzeigers bleibt unverändert.

BS2000
neuer Wert des Lese-/Schreibzeigers, gemessen in Bytes vom Anfang der Datei,
bei Binärdateien,
bei Erfolg

absolute Position
in Textdateien,
bei Erfolg.

-1 bei Fehler.

Fehler `lseek() und lseek64()` schlagen fehl, wenn gilt:

`EBADF`	*fildes* ist kein offener Dateideskriptor.
`EINVAL`	*whence* besitzt keinen erlaubten Wert, oder die sich ergebende Dateiposition wäre nicht zulässig.
`ESPIPE`	*fildes* ist einer Pipe oder FIFO zugeordnet.
`EOVERFLOW`	Der resultierende Datei-Offset kann in der Struktur, auf die `offset` zeigt, nicht korrekt dargestellt werden.

Hinweis Ob eine BS2000- oder eine POSIX-Datei erzeugt wird, hängt von der Programmumgebung ab.

BS2000
Die Aufrufe `lseek`(*stream*, `0L, SEEK_CUR`) und `tell`(*stream*) sind äquivalent, d.h. sie rufen beide die aktuelle Position in der Datei ab, ohne zu positionieren.

Werden in eine Textdatei neue Sätze geschrieben (geöffnet zum Neuerstellen oder Anfügen) und erfolgt ein `lseek`-Aufruf, dann werden zunächst ggf. restliche Daten aus dem C-internen Puffer in die Datei geschrieben und mit Zeilenende (`\n`) abgeschlossen.

Ausnahme bei ANSI-Funktionalität:
Wenn die Daten einer ISAM-Datei im Puffer nicht mit einem Zeilenendezeichen abschließen, bewirkt `lseek()` keinen Zeilenwechsel bzw. Satzwechsel. D.h., die Daten werden beim Schreiben aus dem Puffer nicht automatisch mit einem Zeilenendezeichen abgeschlossen. Nachfolgende Daten verlängern den Satz in der Datei. Beim Lesen einer ISAM-Datei werden daher nur Zeilenendezeichen eingelesen, die vom Programm explizit geschrieben wurden.

Wird bei einer Binärdatei hinter das Dateiende positioniert, entsteht ein Lücke zwischen den letzten physisch gespeicherten Daten und den neu geschriebenen Daten. Lesen aus dieser Lücke liefert binäre Nullen.

Auf Systemdateien (SYSDTA, SYSLST, SYSOUT) kann nicht positioniert werden.

Da die Informationen über die Dateiposition in einem 4 Byte langen Feld abgelegt werden, ergeben sich für die Größe von SAM- und ISAM-Dateien folgende Einschränkungen bei der Bearbeitung mit `tell()/lseek()`:

SAM-Datei

Satzlänge	≤ 2048 Byte
Satzanzahl/Block	≤ 256
Blockanzahl	≤ 2048

ISAM-Datei

Satzlänge	≤ 32 KByte
Satzanzahl	≤ 32 K

Siehe auch `fseek()`, `ftell()`, `open()`, `tell()`, `sys/types.h`, `unistd.h`.

lstat - Dateistatus abfragen

Name **lstat, lstat64**

Syntax
```
#include <sys/stat.h>
#include <sys/types.h>

int lstat (const char *path, struct stat *buf);
int lstat64 (const char *path, struct stat64 *buf);
```

Beschreibung

`lstat()` liefert genau wie `stat()` Dateiattribute. Nur wenn *path* auf einen symbolischen Verweis zeigt, gibt `lstat()` Informationen über den Verweis aus, während `stat()` Informationen über die Datei ausgibt, auf die sich der Verweis bezieht.

buf ist ein Zeiger auf eine `stat`-Struktur, in die die Informationen über die angegebene Datei geschrieben werden.

Es besteht kein funktionaler Unterschied zwischen `lstat()` und l`stat64()`, außer dass bei `lstat64()` der File Status in einer `stat64`-Struktur zurückgegeben wird.

Die Struktur `stat` enthält die folgenden Elemente:

```
mode_t    st_mode;    /* Dateimodus (siehe mknod()) */
ino_t     st_ino;     /* Dateikennziffer (i-Node) */
dev_t     st_dev;     /* Gerätekennung, die einen Verzeichniseintrag für
                         diese Datei enthält */
dev_t     st_rdev;    /* Gerätekennung, nur für zeichen- oder
                         blockorientierte Gerätedateien definiert */
nlink_t   st_nlink;   /* Anzahl der Verweise */
uid_t     st_uid;     /* Benutzerkennung des Dateibesitzers */
gid_t     st_gid;     /* Gruppenkennung des Dateibesitzers */
off_t     st_size;    /* Dateigröße in Bytes */
time_t    st_atime;   /* Zeit des letzten Zugriffs */
time_t    st_mtime;   /* Zeit der letzten Datenänderung */
time_t    st_ctime;   /* Zeit der letzten Änderung des Dateistatus
                         Die Zeit wird in Sekunden gemessen ab dem
                         1. Januar 1970, 00:00:00 Uhr */
long      st_blksize; /* Bevorzugte E/A-Blockgröße */
blkcnt_t  st_blocks;  /* Anzahl zugewiesener st_blksize-Blöcke */
```

Die Struktur `stat64` ist wie die von `stat` definiert, mit Ausnahme folgender Komponenten:

```
ino64_t    st_ino
off64_t    st_size und
blkcnt64_t st_blocks
```

Zusätzlich zu den in `mknod()` beschriebenen Modi kann `st_mode` auch `S_IFLNK` sein, wenn die Datei ein symbolischer Verweis ist.

Die Komponente `st_size` enthält die Länge des Pfadnamens, der in dem symbolischen Verweis steht. Abschließende Nullen werden nicht mitgezählt. Der Inhalt aller übrigen Komponenten der Struktur `stat` ist undefiniert.

Returnwert

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

Fehler `lstat()` und `lstat64()` schlagen fehl, wenn gilt:

`EACCES`	Eine Komponente des Pfades darf nicht durchsucht werden.
`EIO`	Es trat während des Lesens oder Schreibens vom Dateisystem ein Ein- oder Ausgabefehler auf.
`ELOOP`	Bei der Übersetzung von *path* wurden zuviele symbolische Verweise angetroffen.
`ENAMETOOLONG`	Die Länge des Pfadnamens überschreitet `{PATH_MAX}`, oder die Länge einer Komponente des Pfadnamens überschreitet `{NAME_MAX}`.
`ENOTDIR`	Eine Komponente des Pfadnamen-Präfix ist kein Dateiverzeichnis.
`ENOENT`	Eine Komponente des Pfadnamens existiert nicht, oder *path* zeigt auf eine leere Zeichenkette.
`EOVERFLOW`	Eine Komponenete ist zu groß, um in der Struktur, auf die *buf* zeigt, gespeichert zu werden.

BS2000

`EINVAL`	Es wurde versucht, auf eine BS2000-Datei zuzugreifen.
`ENAMETOOLONG`	Die Auflösung symbolischer Verweise im Pfadnamen führt zu einem Zwischenergebnis, dessen Länge `{PATH_MAX}` überschreitet.
`EOVERFLOW`	Eine Komponente ist zu groß, um in der Struktur, auf die *buf* zeigt, gespeichert zu werden.
`EFAULT`	*buf* oder *path* weisen auf eine ungültige Adresse.
`EINTR`	Ein Signal wurde während des Systemaufrufs `stat()` oder `lstat()` abgefangen.

major - höherwertige Komponente der Gerätenummer ermitteln

(Erweiterung)

Syntax
```
#include <sys/types.h>
#include <sys/mkdev.h>

major_t major(dev_t device);
```

Beschreibung

`major()` liefert die höherwertige Komponente der Gerätenummer für ein Gerät *device*.

Returnwert

formatierte Gerätenummer — bei Erfolg.

`NODEV` — bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler

`major()` schlägt fehl, wenn gilt:

`EINVAL` — Das Argument *device* ist `NODEV`, oder die höherwertige Komponente von *device* ist zu groß.

Siehe auch `makedev()`, `minor()`, `mknod()`, `stat()`.

makecontext, swapcontext - Benutzerkontext einrichten

Syntax

```
#include <ucontext.h>

void makecontext (ucontext_t *ucp, (void *func) (), int argc,...);
int swapcontext (ucontext_t *oucp, const ucontext_t *ucp);
```

Beschreibung

Diese Funktionen dienen der Implementierung eines Kontextwechsels zwischen mehreren Kontrollflüssen innerhalb eines Benutzerprozesses.

`makecontext()` verändert den durch *ucp* angegebenen Kontext, der über `getcontext()` initialisiert wurde. Wird dieser Kontext mit `swapcontext()` oder `setcontext()` aktiviert (siehe `getcontext()`), wird die Programmausführung mit dem Aufruf der Funktion *func* fortgesetzt.
Die Argumente, die auf *argc* folgen, werden an `makecontext()` übergeben. Der ganzzahlige Wert von *argc* muss der Anzahl der Argumente entsprechen, die auf *argc* folgt. Ansonsten ist das Verhalten undefiniert.

Bevor `makecontext()` aufgerufen wird, sollte dem zu modifizierenden Kontext ein Stack zugewiesen werden.
Das Strukturelement `uc_link` legt den Kontext fest, der aktiviert wird, wenn der durch `makecontext()` modifizierte Kontext zurückkehrt.

`swapcontext()` sichert den aktuellen Kontext in der Kontextstruktur, auf die *oucp* zeigt, und setzt den Kontext auf die Kontextstruktur, auf die *ucp* zeigt.

Returnwert

0	nach erfolgreicher Ausführung von `swapcontext()`.
-1	bei Fehler. `errno` wird gesetzt, um die Art des Fehlers anzuzeigen.

Fehler Diese Funktionen schlagen fehl, wenn gilt:

`ENOMEM`	*ucp* hat nicht mehr genügend Platz im Stack, um die Operation durchzuführen.

Siehe auch `exit()`, `getcontext()`, `sigaction()`, `sigprocmask()`, `ucontext.h`.

makedev - formatierte Gerätenummer ermitteln *(Erweiterung)*

Syntax
```
#include <sys/types.h>
#include <sys/mkdev.h>

dev_t makedev(major_t maj, minor_t min);
```

Beschreibung

`makedev()` liefert eine formatierte Gerätenummer. *maj* ist die höherwertige Komponente der Gerätenummer und *min* die niederwertige Komponente. `makedev()` kann verwendet werden, um eine Gerätenummer für `mknod()` zu erzeugen.

Returnwert formatierte Gerätenummer bei Erfolg.

`NODEV` bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `makedev()` schlägt fehl, wenn gilt:

`EINVAL` Eines oder beide der Argumente *maj* und *min* ist zu groß, oder die Gerätenummer, die aus *maj* und *min* erzeugt wurde, ist `NODEV`.

Siehe auch `major()`, `minor()`, `mknod()`, `stat()`.

malloc - Speicherbereich zuweisen

Syntax

```
#include <stdlib.h>

void *malloc(size_t size);
```

Beschreibung

`malloc()` beschafft zur Ausführungszeit zusammenhängenden Speicherplatz in der Größe von *size* Byte.
Wenn *size* = 0 den Wert 0 hat, gibt `malloc()` einen Nullzeiger zurück.

`malloc()` ist Teil des C-spezifischen Speicherverwaltungspaketes, das angeforderte und wieder freigegebene Speicherbereiche intern verwaltet. Neue Anforderungen werden zuerst aus bereits verwalteten Bereichen zu erfüllen versucht, dann erst vom Betriebssystem.

Returnwert

Zeiger	auf den neuen Speicherbereich wenn *size* nicht den Wert 0 hatte und `malloc()` neuen Speicherplatz zuweisen konnte. Dieser Zeiger kann für beliebige Datentypen verwendet werden.
Nullzeiger	wenn `malloc()` den Speicherplatz nicht beschaffen konnte, z.B. weil der noch vorhandene Speicherplatz nicht ausreicht oder ein Fehler auftrat. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `malloc()` schlägt fehl, wenn gilt:

`ENOMEM`	Es ist nicht genügend Speicherplatz verfügbar.

Hinweis

Der neue Datenbereich beginnt auf Doppelwortgrenze.

Die tatsächliche Länge des Datenbereichs ist die angeforderte Länge *size* + 8 Byte für interne Verwaltungsdaten. Diese Summe wird ggf. auf die nächste Zweierpotenz aufgerundet.

Um sicherzugehen, dass Sie ausreichend Platz für eine Variable anfordern, sollten Sie den Operator `sizeof` verwenden.
Wird die Länge des zur Verfügung gestellten Speicherbereiches beim Schreiben überschritten, führt dies zu schwer wiegenden Fehlern im Arbeitsspeicher.

`malloc()` ist ab dieser Version unterbrechungssicher, d.h. die Funktion kann nun auch in Signalbehandlungs- und Contingency-Routinen verwendet werden.

Siehe auch `calloc()`, `free()`, `realloc()`, `stdlib.h`.

mblen - Anzahl der Bytes eines Multibyte-Zeichens ermitteln

Syntax #include <stdlib.h>

int mblen(const char *s, size_t n);

Beschreibung

`mblen` liefert die Anzahl Bytes eines Multibyte-Zeichens, auf das *s* zeigt. Dabei werden maximal *n* Bytes in *s* ausgewertet.

In dieser Version sind Zeichen, die aus mehreren Bytes bestehen, nicht realisiert. Multibyte-Zeichen haben immer die Länge 1 (`MB_CUR_MAX` = 1).

Returnwert

-1	falls *n* = 0 ist.
0	falls *s* ein Nullzeiger ist oder auf ein Nullbyte zeigt.
1	in allen anderen Fällen.

Siehe auch `mbstowcs()`, `mbtowc()`, `wcstombs()`, `wctomb()`, `stdlib.h`.

mbrlen - Restlänge eines Multibyte-Zeichens ermitteln

Syntax #include <wchar.h>

size_t mbrlen(const char *s, size_t n, mbstate_t *ps);

Beschreibung

`mbrlen()` ermittelt die Anzahl Bytes ab der Position *s, die zur Vervollständigung eines Multibyte-Zeichens benötigt werden. Es werden maximal *n* Bytes überprüft.

`mbrlen()` entspricht dem Aufruf
`mbrtowc(NULL, s, n, ps!= NULL ? ps: internal)`
wobei *internal* das `mbstate_t`-Objekt für die Funktion ist.

Ausführliche Beschreibung siehe `mbrtowc()`.

mbrtowc - Multibyte-Zeichen vervollständigen und in Langzeichen umwandeln

Syntax

#include <wchar.h>

size_t mbrtowc(wchar_t *_pwc_, const char *_s_, size_t _n_, mbstate_t *_ps_);

Beschreibung

Falls _s_ kein Nullzeiger ist, ermittelt `mbrtowc()`, wie viele Bytes ab der Position, auf die *_s_, zeigt, zur Vervollständigung des nächsten Multibyte-Zeichens benötigt werden. Berücksichtigt werden auch eventuelle Umschalt-Sequenzen (Shift-Sequenzen). Es werden maximal die nächsten _n_ Bytes überprüft. Wenn `mbrtowc()` das Multibyte-Zeichen vervollständigen kann, wird das zugehörige Langzeichen ermittelt und unter *_pwc_ gespeichert, sofern _pwc_ kein Nullzeiger ist.
Ist das zugehörige Langzeichen das Nullzeichen, entspricht der Ergebniszustand dem „initial conversion“ Zustand.
Ist _s_ ein Nullzeiger, entspricht `mbrtowc()` dem Aufruf
`mbrtowc(NULL, "", 1, ps)`
In diesem Fall werden die Werte der Parameter _pwc_ und _n_ ignoriert.

Returnwert

Abhängig vom aktuellen Konvertierungs-Zustand gibt `mbrtowc()` den Wert der ersten zutreffenden Bedingung zurück:

`0` wenn die nächsten (maximal _n_) Bytes ein gültiges Multibyte-Zeichen ergeben, das dem Langzeichen Null entspricht.

Anzahl der zur Vervollständigung des Multibyte-Zeichens benötigten Bytes
falls die nächsten (maximal _n_) Bytes ein gültiges Multibyte-Zeichen ergeben. Gespeichert wird das diesem Multibyte-Zeichen entsprechende Langzeichen.

`(size_t)-2` wenn die nächsten _n_ Bytes ein unvollständiges, aber potenziell gültiges Multibyte-Zeichen ergeben. Es wird kein Wert gespeichert.

`(size_t)-1` wenn ein Kodierfehler auftritt, das heißt die nächsten (maximal _n_) Bytes ergeben kein vollständiges und gültiges Multibyte-Zeichen. Es wird kein Wert gespeichert und in `errno` wird der Wert des Makros `EILSEQ` geschrieben. Der Konversions-Zustand ist undefiniert.

Hinweis

In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Siehe auch

`mblen(), mbtowc(), wcstombs(), wctomb()`

mbsinit - auf „initial conversion“ Zustand überprüfen

Syntax #include <wchar.h>

int mbsinit(const mbstate_t *ps);

Beschreibung

Wenn *ps* kein Nullzeiger ist, überprüft `mbsinit()`, ob das `mbstate_t`-Objekt, auf das *ps* zeigt, einen „initial conversion“ Zustand beschreibt.

Returnwert

Wert ≠ 0	wenn *ps* ein Nullzeiger ist oder auf ein Objekt zeigt, das einen „initial conversion“-Zustand beschreibt
0	sonst.

mbsrtowcs - Multibyte-Zeichenkette in Langzeichenkette umwandeln

Syntax

```
#include <wchar.h>

size_t mbsrtowcs(wchar_t *dst, const char **src, size_t len, mbstate_t *ps);
```

Beschreibung

`mbsrtowcs()` konvertiert eine Folge von Multibyte-Zeichen aus dem Feld, auf das *src* indirekt zeigt, in Langzeichen. `mbsrtowcs()` beginnt die Umwandlung mit dem Konvertierungszustand, der in **ps* beschrieben wird. Die konvertierten Zeichen werden in das Feld geschrieben, auf das *dst* zeigt, sofern *dst* kein Nullzeiger ist. Jedes einzelne Zeichen wird so konvertiert, als sei die Funktion `mbrtowc()` aufgerufen worden.

Die Umwandlung ist beendet, wenn ein abschließendes Nullzeichen auftritt. Das Nullzeichen wird ebenfalls umgewandelt und in das Feld geschrieben.

Die Umwandlung wird vorher abgebrochen, wenn

- eine Bytefolge auftritt, die kein gültiges Multibyte-Zeichen darstellt oder
- *dst* kein Nullzeiger ist und *len* Zeichen in das Feld, auf das *dst* zeigt, geschrieben worden sind.

Wenn *dst* kein Nullzeiger ist, wird dem Zeigerobjekt, auf das *src* zeigt, einer der beiden folgenden Werte zugewiesen:

- ein Nullzeiger, falls die Umwandlung mit dem Erreichen eines Nullzeichens beendet wurde
- die Adresse direkt hinter dem letzten umgewandelten Multibyte-Zeichen.

Wenn *dst* kein Nullzeiger ist und die Umwandlung mit dem Erreichen eines Nullzeichens beendet wurde, entspricht der Ergebniszustand dem „initial conversion“ Zustand.

Returnwert

`(size_t)-1` wenn ein Konvertierungsfehler auftritt, das heißt eine Folge von Bytes, die kein gültiges Multibyte-Zeichen ergeben. In `errno` wird der Wert des Makros `EILSEQ` geschrieben. Der Konversions-Zustand ist undefiniert.

Anzahl der erfolgreich konvertierten Multibyte-Zeichen
sonst. Das abschließende Nullzeichen (falls vorhanden) wird nicht mitgezählt.

Siehe auch `mblen(), mbtowc(), wcstombs(), wctomb()`

mbstowcs - Multibyte-Zeichenkette in Langzeichenkette umwandeln

Syntax #include <stdlib.h>

size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);

Beschreibung

`mbstowcs()` wandelt eine Folge von Multibyte-Zeichen in der Zeichenkette *s* in die entsprechenden `wchar_t`-Werte um und speichert maximal *n* `wchar_t`-Werte in den Bereich *pwcs*. `mbstowcs()` wandelt um, bis entweder *n* Werte konvertiert sind oder der Wert Null auftritt (Null wird in den `wchar_t`-Wert 0 konvertiert).

Ist *pwcs* ein Nullzeiger, gibt `mbstowcs()` unabhängig vom Wert *n* die Länge zurück, die benötigt wird, um die gesamte Zeichenkette umzuwandeln, aber speichert keine Werte.

Wenn ein ungültiges Zeichen auftritt, liefert `mbstowcs()` den Wert (`size_t`)-1 zurück.

Die von `mbstowcs()` im Bereich *pwcs* abgespeicherten `wchar_t`-Werte (Typ long) entsprechen den Werten der einzelnen Bytes in der Zeichenkette *s*.

Returnwert Anzahl der in *pwcs* abgespeicherten `wchar_t`-Werte (ohne das abschließende Nullbyte), wenn *pwcs* kein Nullzeiger ist.
Wenn der Returnwert dem Wert *n* entspricht, ist der Ergebnisbereich *pwcs* nicht mit dem Nullbyte abgeschlossen.

Länge, die benötigt wird, um die gesamte Zeichenkette umzuwandeln, wenn *pwcs* ein Nullzeiger ist. Es werden keine Werte gespeichert.

(`size_t`)-1 bei Fehler.

Hinweis Bei sich überlappenden Speicherbereichen ist das Verhalten undefiniert.

In dieser Version sind Zeichen, die aus mehreren Bytes bestehen, nicht realisiert. Multibyte-Zeichen haben immer die Länge 1 Byte und `wchar_t`-Werte sind immer long-Werte.

Siehe auch `mblen()`, `mbtowc()`, `wcstombs()`, `wctomb()`, `stdlib.h`.

mbtowc - Multibyte-Zeichen in Langzeichen umwandeln

Syntax

#include <stdlib.h>

int mbtowc(wchar_t *pwc, const char *s, size_t n);

Beschreibung

`mbtowc()` wandelt ein Multibyte-Zeichen in *s* in den entsprechenden `wchar_t`-Wert um und speichert diesen in den Bereich *pwc*. Dabei werden maximal *n* Bytes in *s* ausgewertet.

Der von `mbtowc()` im Bereich *pwc* abgespeicherte `wchar_t`-Wert (Typ long) enspricht dem Wert des Bytes in *s*.

Keine Zuweisung erfolgt, wenn:
pwc oder *s* ein Nullzeiger ist,
n = 0 ist.

Returnwert

-1	falls *n* = 0 ist.
0	falls *s* ein Nullzeiger ist oder auf ein Nullbyte zeigt.
1	sonst in allen anderen Fällen.

Hinweis

In dieser Version sind Zeichen, die aus mehreren Bytes bestehen, nicht realisiert. Multibyte-Zeichen haben immer die Länge 1 Byte und `wchar_t`-Werte sind immer long-Werte.

Siehe auch

`mblen()`, `mbstowcs()`, `wcstombs()`, `wctomb()`, `stdlib.h`.

memalloc - Speicherbereich zuweisen *(BS2000)*

Syntax #include <stdlib.h>

void *memalloc(size_t *anz*);

Beschreibung

`memalloc()` beschafft zur Ausführungszeit zusammenhängenden Speicherplatz in der Größe von *anz* Byte.

`memalloc()` reicht die Speicheranforderung direkt an den entsprechenden Betriebssystemaufruf durch. Die Funktion eignet sich vor allem für Speicherbereiche mit einer Größe von mehr als 2 KByte (siehe auch `memfree()`).

Returnwert

Zeiger	auf den neuen Speicherbereich falls `memalloc()` neuen Speicherplatz zuweisen konnte. Dieser Zeiger kann für beliebige Datentypen verwendet werden.
Nullzeiger	falls `memalloc()` den Speicherplatz nicht beschaffen konnte, z.B. weil der noch vorhandene Speicherplatz nicht ausreicht.

Hinweis

Der neue Speicherbereich beginnt auf Doppelwortgrenze.

Die angeforderte Länge *anz* wird auf das nächste Vielfache von 2 KByte aufgerundet.

Wird die Länge dieses Speicherbereiches beim Schreiben überschritten, führt dies zu schwerwiegender Unordnung im Arbeitsspeicher.

Der mit `memalloc()` angeforderte Speicherbereich kann mit `memfree()` wieder freigegeben werden.

Siehe auch `memfree()`.

memccpy - Bytes im Speicher kopieren

Syntax #include <string.h>

void *memccpy(void **s1*, const void **s2*, int c, size_t *n*);

Beschreibung

`memccpy()` kopiert Bytes aus dem Speicherbereich *s2* nach *s1* bis

– entweder *c* zum ersten Mal kopiert wurde (wobei *c* in ein `unsigned char` konvertiert wird),

– oder *n* Bytes kopiert wurden.

Falls der Kopiervorgang Objekte betrifft, die sich überlappen, ist das Verhalten undefiniert.

Returnwert

Zeiger	auf das Byte nach der Kopie von *c* in *s1* bei Erfolg.
Nullzeiger	wenn *c* nicht in den ersten *n* Zeichen von *s2* gefunden wurde.

Hinweis `memccpy()` überprüft nicht, ob es in dem Speicherbereich, in den kopiert wird, zu einem Überlauf kommt.

Siehe auch `memchr()`, `memcmp()`, `memcpy()`, `memset()`, `string.h`.

memchr - Byte im Speicher finden

Syntax #include <string.h>

void *memchr(const void **s*, int *c*, size_t *n*);

Beschreibung

`memchr()` sucht das erste Vorkommen des Zeichens *c* in den ersten *n* Bytes des Speicherbereiches, auf den *s* zeigt.

s ist der Zeiger auf den Speicherbereich, in dem das Zeichen *c* gesucht werden soll.

c ist der EBCDIC-Wert des Zeichens, das gesucht werden soll.

n ist der ganzzahlige Wert, der die Anzahl der zu durchsuchenden Bytes in *s* angibt.

Returnwert

Zeiger	auf die Position von *c* im Bereich *s* bei Erfolg.
Nullzeiger	wenn *c* in dem angegebenen Bereich nicht vorkommt.

Hinweis Die Funktion eignet sich zum Bearbeiten von Zeichenvektoren, die im Unterschied zu Zeichenketten nicht mit dem Nullbyte (`\0`) abgeschlossen sein müssen.

Siehe auch `memcmp()`, `memcpy()`, `memset()`, `string.h`.

memcmp - Bytes im Speicher vergleichen

Syntax

```
#include <string.h>

int memcmp(const void *s1, const void *s2, size_t n);
```

Beschreibung

`memcmp()` vergleicht die Inhalte der Speicherbereiche, auf die *s1* und *s2* zeigen, in den ersten *n* Bytes.

s1 und *s2* sind Zeiger auf die Speicherbereiche, die verglichen werden sollen.

n ist ein ganzzahliger Wert, der die Anzahl der zu vergleichenden Bytes angibt.

Returnwert Ganzzahliger Wert, und zwar:

< 0	Der Inhalt von *s1* ist in den ersten *n* Bytes lexikalisch kleiner als der Inhalt von *s2*.
0	Die Inhalte von *s1* und *s2* sind in den ersten *n* Bytes lexikalisch gleich groß (d.h. identisch).
> 0	Der Inhalt von *s1* ist in den ersten *n* Bytes lexikalisch größer als der Inhalt von *s2*.

Hinweis Die Funktion eignet sich für die Bearbeitung von Zeichenvektoren, die im Unterschied zu Zeichenketten nicht mit dem Nullbyte (`\0`) abgeschlossen sein müssen.

Siehe auch `memchr()`, `memcpy()`, `memset()`, `string.h`.

memcpy - Bytes im Speicher kopieren

Syntax #include <string.h>

void *memcpy(void **s1*, const void **s2*, size_t *n*);

Beschreibung

`memcpy()` kopiert die ersten *n* Bytes des Speicherbereiches, auf den *s2* zeigt, in den Speicherbereich, auf den *s1* zeigt.

s1 ist ein Zeiger auf den Speicherbereich, in den kopiert werden soll.

s2 ist ein Zeiger auf den Speicherbereich, aus dem die ersten *n* Bytes kopiert werden sollen.

n ist ein ganzzahliger Wert, der die Anzahl der zu kopierenden Bytes in *s2* angibt.

Returnwert Zeiger auf den Speicherbereich *s1* bei Erfolg.

Hinweis Die Funktion eignet sich für die Bearbeitung von Zeichenvektoren, die im Unterschied zu Zeichenketten nicht mit dem Nullbyte (`\0`) abgeschlossen sein müssen.

`memcpy()` überprüft nicht, ob im Ergebnisbereich *s1* ein Überschreiben droht.

Bei sich überlappenden Speicherbereichen ist das Verhalten undefiniert.

Siehe auch `memccpy()`, `memchr()`, `memcmp()`, `memset()`, `string.h`.

memfree - Speicherbereich freigeben *(BS2000)*

Syntax #include <stdlib.h>

void memfree(const void **zg*, size_t *anz*);

Beschreibung

`memfree()` gibt den Speicherbereich, auf den *zg* zeigt, in der Größe von *anz* Byte frei.

`memfree()` reicht die Freigabeanforderung direkt an den entsprechenden Betriebssystemaufruf durch. `memfree()` kann nur in Zusammenhang mit `memalloc()` benutzt werden. Beide Funktionen eignen sich vor allem für Speicherbereiche mit einer Größe von mehr als 2 KByte.

zg ist ein Zeiger auf den freizugebenden Speicherbereich.
zg muss das Ergebnis eines vorangegangenen `memalloc`-Aufrufs sein.

anz ist ein ganzzahliger Wert, der die Größe des Speicherbereichs in Bytes angibt.

Hinweis Mit `memfree()` kann nur ein mit `memalloc()` angeforderter Speicherbereich freigegeben werden.

Die an `memfree()` übergebenen Werte müssen mit denen vom entsprechenden `memalloc`-Aufruf übereinstimmen. Zufällige Werte führen zu schwer wiegenden Fehlern im Arbeitsspeicher!

Siehe auch `memalloc()`.

memmove - Bytes von überlappenden Speicherbereichen kopieren

Syntax

```
#include <string.h>

void *memmove(void *s1, const void *s2, size_t n);
```

Beschreibung

`memmove()` kopiert die ersten *n* Bytes des Speicherbereiches, auf den *s2* zeigt, in den Speicherbereich, auf den *s1* zeigt.
`memmove()` kopiert die *n* Bytes zunächst in ein temporäres Feld, das die Speicherbereiche *s1* und *s2* nicht überlappt, und anschließend erst in den Speicherbereich *s1*.

s1 ist ein Zeiger auf den Speicherbereich, in den kopiert werden soll.

s2 ist ein Zeiger auf den Speicherbereich, aus dem die ersten *n* Bytes kopiert werden sollen.

n ist ein ganzzahliger Wert, der die Anzahl der zu kopierenden Bytes in *s2* angibt.

Returnwert

Zeiger	auf den Speicherbereich *s1* bei Erfolg.

Hinweis

Die Funktion eignet sich für die Bearbeitung von Zeichenvektoren, die im Unterschied zu Zeichenketten nicht mit dem Nullbyte (`\0`) abgeschlossen sein brauchen.

Im Unterschied zu `memcpy()` funktioniert `memmove()` auch mit Speicherbereichen, die sich überlappen.

Siehe auch `memcpy()`, `string.h`.

memset - Speicherbereich initialisieren

Syntax #include <string.h>

void *memset(void *s, int c, size_t n);

Beschreibung

`memset()` kopiert den Wert des Zeichens *c* in die ersten *n* Bytes des Speicherbereiches, auf den *s* zeigt.

s ist ein Zeiger auf den Speicherbereich, der mit dem Zeichen *c* initialisiert werden soll.

c ist ein EBCDIC-Wert des Zeichens, das kopiert werden soll.

n ist ein ganzzahliger Wert, der die Anzahl der Bytes in *s* angibt, die mit dem Zeichen *c* initialisiert werden sollen.

Returnwert Zeiger auf den Speicherbereich *s* bei Erfolg.

Hinweis Die Funktion eignet sich für die Bearbeitung von Zeichenvektoren, die im Unterschied zu Zeichenketten nicht mit dem Nullbyte (`\0`) abgeschlossen sein müssen.

`memset()` überprüft nicht, ob im Ergebnisbereich *s* ein Überschreiben droht.

Siehe auch `memcpy()`, `memchr()`, `memcmp()`, `memcpy()`, `string.h`.

minor - niederwertige Komponente der Gerätenummer ermitteln

(Erweiterung)

Syntax
```
#include <sys/types.h>
#include <sys/mkdev.h>

minor_t minor(dev_t device);
```

Beschreibung

`minor()` liefert die niederwertige Komponente der Gerätenummer für ein Gerät *device*.

Returnwert

formatierte Gerätenummer — bei Erfolg.

`NODEV` — bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler

`minor()` schlägt fehl, wenn gilt:

`EINVAL` — Das Argument *device* ist `NODEV`.

Siehe auch `makedev()`, `major()`, `mknod()`, `stat()`.

mkdir - Dateiverzeichnis erzeugen

Syntax

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

Optional
#include <sys/types.h>  ❑

int mkdir(const char *path, mode_t mode);
```

Beschreibung

`mkdir()` erstellt ein neues Dateiverzeichnis mit dem Namen *path*. Der Modus des neuen Dateiverzeichnisses wird mit *mode* (siehe `chmod()` für mögliche Werte für Modus) initialisiert. Der Schutzbitteil von *mode* wird durch die Dateimaske des Prozesses verändert (siehe `umask()`).

Die Eigentümernummer des Verzeichnisses wird auf die effektive Benutzernummer des Prozesses gesetzt. Die Gruppennummer des Verzeichnisses wird auf die effektive Gruppennummer des Prozesses gesetzt, oder die Gruppennummer dieses Verzeichnisses wird geerbt, wenn das Bit `S_ISGID` im übergeordneten Verzeichnis gesetzt ist. Das Bit `S_ISGID` des neuen Verzeichnisses wird vom übergeordneten Verzeichnis übernommen.

Wenn *path* ein symbolischer Verweis ist, wird er nicht verwendet.

Das neu erzeugte Verzeichnis ist mit Ausnahme der Einträge für sich selbst und sein übergeordnetes Verzeichnis leer.

Nach erfolgreicher Beendigung kennzeichnet `mkdir()` die Felder `st_atime`, `st_ctime` und `st_mtime` des Verzeichnisses zur Aktualisierung. Auch die Felder `st_ctime` und `st_mtime` des Dateiverzeichnisses, das den neuen Eintrag enthält, werden zur Aktualisierung gekennzeichnet.

Returnwert

0	bei Erfolg.
-1	bei Fehler. Es wird kein Dateiverzeichnis erzeugt und `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler

`mkdir()` schlägt fehl, wenn gilt:

`EACCES`	Entweder besteht für eine Komponente des Pfades kein Suchrecht, oder es besteht kein Schreibrecht für das dem neuen Dateiverzeichnis übergeordnete Verzeichnis.
`EEXIST`	Die angegebene Datei ist bereits vorhanden.
Erweiterung `EFAULT`	*path* weist über den zugewiesenen Adressraum des Prozesses hinaus.
`EIO`	Während des Zugriffs auf das Dateisystem ist ein Ein-/Ausgabefehler aufgetreten.

ELOOP	Bei der Übersetzung von *path* wurden zuviele symbolische Verweise angetroffen. ❑
EMLINK	Die Höchstzahl {LINK_MAX} von Verweisen im übergeordneten Dateiverzeichnis wurde überschritten.
ENAMETOOLONG	Die Länge des Arguments *path* überschreitet {PATH_MAX}, oder die Länge einer Komponente von *path* überschreitet {NAME_MAX}.
ENOENT	Eine Komponente des Pfades ist nicht vorhanden, oder *path* zeigt auf eine leere Zeichenkette.
Erweiterung ENOLINK	*path* weist auf einen fernen Rechner, und die Verbindung zu diesem Rechner ist nicht mehr aktiv. ❑
ENOSPC	Auf dem Gerät, das das Verzeichnis enthält, ist kein freier Platz verfügbar.
ENOTDIR	Eine Pfadkomponente ist kein Verzeichnis.
EROFS	Die angegebene Datei steht in einem schreibgeschützten Dateisystem.

Hinweis `mkdir()` wird nur für POSIX-Dateien ausgeführt.

Siehe auch `chmod()`, `mknod()`, `umask()`, `stat()`, `sys/stat.h`, `sys/types.h`.

mkfifo - FIFO-Datei erzeugen

Syntax

#include <sys/stat.h>

Optional
#include <sys/types.h> ❑

int mkfifo(const char **path*, mode_t *mode*);

Beschreibung

`mkfifo()` erzeugt eine neue FIFO-Gerätedatei (FIFO) mit dem Pfadnamen *path*. Die Zugriffsrechte der neuen FIFO werden durch *mode* initialisiert. Die Schutzbits des Arguments *mode* werden durch die Schutzbit-Maske des Prozesses verändert (siehe auch `umask()`).

Die Benutzernummer der FIFO wird auf die effektive Benutzernummer des aufrufenden Prozesses gesetzt. Die Gruppennummer der FIFO wird auf die effektive Gruppennummer des Prozesses gesetzt. Wenn jedoch das `S_ISGID`-Bit im übergeordneten Dateiverzeichnis gesetzt ist, wird die Gruppennummer der FIFO vom übergeordneten Verzeichnis übernommen.

Bei erfolgreicher Beendigung aktualisiert `mkfifo()` die `stat`-Strukturkomponenten `st_atime`, `st_ctime` und `st_mtime` der FIFO. Auch die `stat`-Strukturkomponenten `st_ctime` und `st_mtime` des Dateiverzeichnisses, das den neuen Eintrag enthält, werden aktualisiert (siehe `sys/stat.h`).

Returnwert

0	bei Erfolg.
-1	wenn keine FIFO erzeugt wurde. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler

`mkfifo()` schlägt fehl, wenn gilt:

`EACCES`	Entweder besteht für eine Komponente des Pfades kein Suchrecht, oder es besteht kein Schreibrecht für das der neuen FIFO-Datei übergeordnete Dateiverzeichnis.
`EEXIST`	Die angegebene Datei existiert bereits.
`ELOOP`	Bei der Auflösung von *path* wurden zuviele symbolische Verweise angetroffen.

Erweiterung

`EINVAL`	Es wurde versucht, auf eine BS2000-Datei zuzugreifen. ❑
`ENAMETOOLONG`	Die Länge des Arguments *path* überschreitet `{PATH_MAX}`, oder eine Komponente des Pfadnamens ist länger als `{NAME_MAX}` und `{_POSIX_NO_TRUNC}` ist aktiv.

ENOENT	Eine Komponente des Pfadpräfixes existiert nicht oder das Argument *path* zeigt auf die leere Zeichenkette.
ENOSPC	Das Dateiverzeichnis, das die neue Datei enthalten würde, kann nicht erweitert werden oder das Dateisystem kann keine Dateien mehr reservieren.
ENOTDIR	Eine Komponente des Pfadpräfixes ist kein Dateiverzeichnis.
EROFS	Die angegebene Datei befindet sich in einem nur zum Lesen freigegebenen Dateisystem.

Hinweis Wenn in *mode* andere Bits als die Datei-Schutzbits gesetzt sind, werden diese ignoriert.

path kann nur eine POSIX-Datei sein.

Siehe auch `umask()`, `sys/stat.h`, `sys/types.h`.

mknod - Dateiverzeichnis, Gerätedatei oder Textdatei erzeugen

Syntax #include <sys/stat.h>

int mknod(const char *_path_, mode_t _mode_, dev_t _dev_);

Beschreibung

`mknod()` erstellt eine neue Datei mit dem Pfadnamen, auf den *path* zeigt. Der Dateityp und die Zugriffsrechte der neuen Datei werden von *mode* bestimmt.
Wenn *path* ein symbolischer Verweis ist, wird er nicht verfolgt.

Der Dateityp für *path* wird durch bitweises ODER in das *mode*-Argument übernommen. Der Dateityp muss eine der folgenden symbolischen Konstanten sein:

`S_IFIFO`	FIFO-Datei
`S_IFCHR`	zeichenorientierte Datei (nicht portabel)
`S_IFDIR`	Verzeichnis (nicht portabel)
`S_IFBLK`	blockorientierte Datei (nicht portabel)
`S_IFPOSIXBS2`	Datei im POSIX-Dateisystem (nicht portabel)
`S_IFREG`	normale Datei (nicht portabel)

`mknod()` kann gemäß X/Open-Standard nur dann portabel verwendet werden, wenn eine FIFO-Datei erzeugt wird. Falls der Dateityp nicht `S_IFIFO` ist oder `dev` nicht den Wert 0 hat, ist das Verhalten von `mknod()` undefiniert.

Die Zugriffsrechte der Datei werden ebenfalls durch bitweises ODER in das *mode*-Argument übernommen. Die Zugriffsrechte können durch eine beliebige Kombination der folgenden symbolischen Konstanten definiert werden:

Symbolischer Name	Bitmuster	Bedeutung
`S_ISUID`	`04000`	Setzen der Benutzernummer bei Ausführung
`S_ISGID`	`020#0`	Setzen der Gruppennummer bei Ausführung
`S_IRWXU`	`00700`	Lesen, Schreiben, Ausführen (Durchsuchen, wenn es sich um ein Dateiverzeichnis handelt) durch Eigentümer
`S_IRUSR`	`00400`	Lesen durch Eigentümer
`S_IWUSR`	`00200`	Schreiben durch Eigentümer
`S_IXUSR`	`00100`	Ausführen durch Eigentümer (Durchsuchen, wenn es sich um ein Dateiverzeichnis handelt)
`S_IRWXG`	`00070`	Lesen, Schreiben, Ausführen (Durchsuchen) durch Gruppe
`S_IRGRP`	`00040`	Lesen durch Gruppe
`S_IWGRP`	`00020`	Schreiben durch Gruppe
`S_IXGRP`	`00010`	Ausführen (Durchsuchen) durch Gruppe

S_IRWXO	00007	Lesen, Schreiben, Ausführen (Durchsuchen) durch Andere
S_IROTH	00004	Lesen durch Andere
S_IWOTH	00002	Schreiben durch Andere
S_IXOTH	00001	Ausführen durch Andere
S_ISVTX	01000	Für Dateiverzeichnisse: eingeschränktes Löschungsrecht

Die Benutzernummer der Datei wird auf die effektive Benutzernummer des Prozesses gesetzt. Die Gruppennummer der Datei wird auf die effektive Gruppennummer des Prozesses gesetzt, sofern nicht das S_ISGID-Bit im übergeordneten Verzeichnis gesetzt ist: bei gesetztem S_ISGID-Bit wird die Gruppennummer des übergeordneten Verzeichnisses übernommen.

Die Bits für die Zugriffrechte in *mode* werden durch die Dateierzeugungsmaske des Prozesses geändert: mknod() setzt alle Bits auf 0, die in der Dateierzeugungsmaske gesetzt sind.

Falls *mode* eine zeichen- oder blockorientierte Datei angibt, ist *dev* die konfigurationsabhängige Angabe dieser Datei. Falls *mode* keine zeichen- oder blockorientierte Datei angibt, wird *dev* ignoriert. Siehe mkdev().

Für andere Dateitypen als FIFO kann mknod() nur durch Benutzer mit entsprechenden Zugriffsrechten (uid = 0) aufgerufen werden.

Returnwert

0	bei Erfolg.
-1	bei Fehler. errno wird gesetzt, um den Fehler anzuzeigen. Im Fehlerfall wird keine neue Datei erzeugt.

Fehler mknod() schlägt fehl, wenn gilt:

EACCES	Entweder besteht für eine Komponente des Pfades kein Suchrecht, oder es besteht kein Schreibrecht für das der neuen Datei übergeordnete Dateiverzeichnis.
EEXIST	Die angegebene Datei existiert bereits.
EINTR	Während des Systemaufrufs mknod() wurde ein Signal empfangen.
EINVAL	Ein Argument ist ungültig.
EIO	Beim Zugriff auf das Dateisystem trat ein Ein-/Ausgabefehler auf..
ELOOP	Bei der Auflösung von *path* traten zuviele symbolische Verweise auf.

ENAMETOOLONG
: Die Länge des *path*-Arguments überschreitet {PATH_MAX}, oder die Länge einer Komponente von *path* überschreitet {NAME_MAX}.

  Bei der Auflösung eines symbolischen Verweises in *path* kam es zu einem Zwischenergebnis, dessen Länge {PATH_MAX} überschreitet.

ENOENT
: Eine Komponente des Pfadpräfixes existiert nicht oder *path* ist ein leerer String.

ENOLINK
: *path* verweist auf einen fernen Rechner und die Verbindung zu diesem Rechner ist nicht mehr aktiv.

ENOSPC
: Das Verzeichnis, in dem die Datei erstellt werden soll, kann nicht erweitert werden, oder es ist kein Speicherplatz mehr vorhanden.

ENOTDIR
: Eine Komponente des Pfadpräfixes ist kein Verzeichnis.

EPERM
: Die effektive Benutzernummer ist nicht die des Systemverwalters und der Dateityp ist nicht FIFO.

EROFS
: Das Verzeichnis, in dem die Datei erstellt werden soll, liegt in einem Dateisystem, das nur gelesen werden kann.

Hinweis

mknod() wird nur für POSIX-Dateien ausgeführt.
Wenn mknod() mit RFS (remote file sharing) in einem fernen Verzeichnis eine Gerätedatei erzeugt, werden Geräteklasse und Gerätenummer vom Server interpretiert.

Aus Gründen der Portabilität zu Implementierungen, die sich an frühere Versionen des X/Open-Standards halten, wird für die Erzeugung von FIFO-Dateien die Funktion mkfifo() empfohlen.

Siehe auch

chmod(), creat(), exec(), mkdir(), mkfifo(), open(), stat(), umask(), sys/stat.h, sys/types.h.

mkstemp - eindeutigen temporären Dateinamen erzeugen

Syntax #include <stdlib.h>

int mkstemp(char **template*);

Beschreibung

`mkstemp()` erstellt einen eindeutigen Dateinamen, normalerweise in einem temporären Dateisystem, und gibt einen offenen Dateideskriptor für diese Datei zurück. Die Datei ist zum Lesen und Schreiben geöffnet.
`mkstemp()` verhindert auf diese Weise einen möglichen Wettlauf zwischen einer Existenzprüfung und dem Öffnen der Datei.

Die Zeichenkette, auf die *template* zeigt, sollte einen Dateinamen mit sechs nachfolgenden 'X' enthalten. `mkstemp()` ersetzt die 'X' zur Erstellung eines eindeutigen Dateinamens durch einen Buchstaben und die aktuelle Prozess-ID. Der Buchstabe wird so gewählt, dass sich keine doppelten Dateinamen ergeben.

Returnwert offener Dateideskriptor
bei Erfolg

-1 wenn keine geeignete Datei erstellt werden konnte.

Hinweis Es besteht die Möglichkeit, dass die Buchstaben ausgehen.

`mkstemp()` überprüft nicht, ob die Dateinamen-Komponente in *template* die maximal erlaubte Länge von Dateinamen überschreitet.

Aus Gründen der Portabilität zu Implementierungen, die sich an frühere Versionen des X/Open-Standards halten, wird zur Erzeugung eines eindeutigen Dateinamens die Funktion `tmpfile()` empfohlen.

`mkstemp()` ändert die übergebene Zeichenkette, die durch *template* angegeben wird. Dies bedeutet, dass Sie eine Zeichenkette, die durch *template* angegeben wird, nicht mehrmals verwenden können. Für jede eindeutige temporäre Datei, die Sie öffnen möchten, benötigen Sie eine neue Schablone.

Wenn `mkstemp()` einen neuen eindeutigen Dateinamen erstellt, wird zunächst überprüft, ob vorher bereits eine Datei mit diesem Namen existiert hat. Wenn Sie also mehr als einen eindeutigen Dateinamen erstellen, sollte für mehrere Aufrufe von `mkstemp()` nicht dieselbe Dateinamen-Komponente in *template* verwendet werden.

Siehe auch `getpid()`, `open()`, `tmpfile()`, `tmpnam()`, `stdlib.h`.

mktemp - eindeutigen temporären Dateinamen erzeugen *(Erweiterung)*

Syntax

```
#include <stdlib.h>

char *mktemp(char *template);
```

Beschreibung

`mktemp()` ersetzt den Inhalt der Zeichenkette, auf die *template* zeigt, durch einen eindeutigen Dateinamen und gibt die Adresse von *template* zurück.
Die Zeichenkette, auf die *template* zeigt, sollte einen Dateinamen mit sechs nachfolgenden 'X' enthalten. `mkstemp()` ersetzt die 'X' zur Erstellung eines eindeutigen Dateinamens durch einen Buchstaben und die aktuelle Prozess-ID. Der Buchstabe wird so gewählt, dass sich keine doppelten Dateinamen ergeben.

BS2000
`mktemp()` erzeugt einen eindeutigen Dateinamen für eine temporäre SAM-Datei. Der Name muss aus mindestens acht Zeichen bestehen und wird wie folgt gebildet:

- Die ersten drei Zeichen werden ersetzt durch „#T.“.
- Das vierte Zeichen wird durch ein Zeichen ersetzt, das sich bei jedem `mktemp`-Aufruf ändert (Buchstaben A - Z, Ziffern 0 - 9).
- die letzten vier Zeichen werden ersetzt durch die TSN-Nummer des aktuellen Prozesses (seit LOGON).
- Zeichen zwischen den ersten und letzten vier Zeichen bleiben unverändert.

Hat *template* z.B den Wert "XXXX.ABC.XXXX" und die TSN-Nummer des aktuellen Prozesses ist 6082, dann erzeugt `mktemp()` beim ersten Aufruf den temporären Namen #T.A.ABC.6082 ❑

Returnwert

Zeiger auf eine Zeichenkette, die den neuen Namen enthält,
bei Erfolg.

Zeiger auf eine leere Zeichenkette
wenn kein eindeutiger Name erstellt werden kann, weil z.B. keine Buchstaben mehr frei sind.

Hinweis

In dem Zeitraum zwischen der Erzeugung des Dateinamens und dem Öffnen der Datei kann ein anderer Prozess eine Datei mit demselben Namen erzeugen. Wenn Sie die Funktion `mkstemp()` verwenden, vermeiden Sie dieses Problem.

Aus Gründen der Portabilität zu Implementierungen, die sich an frühere Versionen des X/Open-Standards halten, wird zur Erzeugung eines eindeutigen Dateinamens die Funktion `tmpnam()` empfohlen.

`mktemp()` kann maximal 26 eindeutige Dateinamen pro Prozess für jedes eindeutige *template* erzeugen.

BS2000
Temporäre Dateien werden automatisch bei Beendigung eines Prozesses (LOGOFF) gelöscht. Wenn allerdings bei der Systemgenerierung das standardmäßige Präfix (#) für temporäre Dateien geändert wurde, bleiben die Dateien erhalten. ❑

Ob eine BS2000-Datei oder eine POSIX-Datei erstellt wird, hängt von der Programmumgebung ab.

Siehe auch `mkstemp()`, `tmpfile()`, `tmpnam()`, `stdlib.h`.

mktime - Ortszeit in Zeit seit Epochenwert umwandeln

Syntax

#include <time.h>

time_t mktime(struct tm **timeptr*);

Beschreibung

`mktime()` wandelt die Zeit, die durch die `tm`-Struktur dargestellt wird und auf die *timeptr* zeigt, in die Kalenderzeit um (die Anzahl der Sekunden seit 00:00:00 UTC, Universal Time Coordinated, 1. Januar 1970).

Die `tm`-Struktur hat das folgende Format:

```
struct tm {
      int     tm_sec;         /* Sekunden [0, 61]  */
      int     tm_min;         /* Minuten [0, 59] */
      int     tm_hour;        /* Stunde [0, 23] */
      int     tm_mday;        /* Monatstag [1, 31] */
      int     tm_mon;         /* Monat [0, 11] */
      int     tm_year;        /* Jahre seit 1900 */
      int     tm_wday;        /* Tage seit Sonntag [0, 6] */
      int     tm_yday;        /* Tage seit 1. Januar 1 [0, 365] */
      int     tm_isdst;       /* Schalter für Sommerzeit */
};
```

Neben der Berechnung der Kalenderzeit normalisiert `mktime()` die übergebene `tm`-Struktur. Die Originalwerte der Komponenten `tm_wday` und `tm_yday` werden ignoriert; die Originalwerte der anderen Komponenten der Struktur sind nicht auf die oben angegebenen Grenzen beschränkt. Bei erfolgreicher Ausführung werden die Komponenten `tm_wday` und `tm_yday` entsprechend gesetzt; die anderen Komponenten werden so eingestellt, dass sie die angegebene Kalenderzeit darstellen, wobei die entsprechenden Wertebereiche eingehalten werden. Der endgültige Wert von `tm_mday` wird nicht gesetzt, bis `tm_mon` und `tm_year` bestimmt sind.

Die Originalwerte der Komponenten können größer oder kleiner als die angegebenen Bereiche sein. Beispielsweise zeigt der Wert -1 für `tm_hour` eine Stunde vor Mitternacht an; enthält `tm_mday` den Wert 0, so wird der Tag vor dem aktuellen Monat bezeichnet; steht `tm_mon` auf -2, so bedeutet dies zwei Monate vor Januar des Jahres `tm_year`.

Ist `tm_isdst` > 0, wird angenommen, dass sich die ursprünglichen Werte in der alternativen Zeitzone befinden, d.h. dass Sommerzeit gilt. Stellt sich heraus, dass die alternative Zeitzone für die berechnete Kalenderzeit ungültig ist, werden die Komponenten an die primäre Zeitzone angepasst. Wenn `tm_isdst` null ist, wird angenommen, dass sich die Originalwerte in der primären Zeitzone befinden, d.h. dass Normalzeit gilt; diese Werte werden in die alternative Zeitzone übersetzt, falls die primäre Zeitzone ungültig ist. Wenn `tm_isdst` negativ ist, ermittelt `mktime()` die korrekte Zeitzone.

Die lokale Zeitzoneninformation wird so verwendet, als wenn `mktime()` die Funktion `tzset()` aufrufen würde.

BS2000
`mktime()` wandelt Datum und Uhrzeit, die der Benutzer in einer Struktur vom Typ `tm` angibt, in eine Zeitangabe vom Typ `time_t` um. Dies ist die Anzahl der vergangenen Sekunden, bezogen auf den Stichtag 1. Januar 1950 00.00.00 Uhr. ❑

Returnwert Anzahl der Sekunden
bei Erfolg.

`time_t` - 1 wenn die Kalenderzeit nicht dargestellt werden kann.

BS2000
Bei Ortszeiten ab dem 1. Januar 1950 00.00.00 die Anzahl der Sekunden, die seither vergangen sind (positiver Wert).

Bei Ortszeiten vor dem 1. Januar 1950 00.00.00 die Anzahl der Sekunden, die bis dahin vergangen sind (negativer Wert). ❑

Beispiel Welcher Wochentag ist der 4. Juli 2001?

```
#include <stdio.h>
#include <time.h>

struct tm time_str
char daybuf[20]

int main (void)

{

      time_str.tm_year = 2001 - 1900;
      time_str.tm_mon = 7 - 1;
      time_str.tm_mday = 4;
      time_str.tm_hour = 0;
      time_str.tm_min = 0;
      time_str.tm_sec = 1;
      time_str.tm_isdst = -1;

      if (mktime ( &time_str) == -1)
              (void) puts (" -unknown-“);
      else {
              (void) strftime (daybuf, sizeof (daybuf), "%A", &time_str);
       }

       return 0;

}
```

Hinweis `tm_year` in der `tm`-Struktur muss mindestens 1970 oder später sein. Kalenderzeiten vor 00:00:00 UTC, 1. Januar 1970 oder nach 03:14:07 UTC, 19. Januar 2038, können nicht dargestellt werden.

BS2000
`mktime()` liefert gültige Werte für Zeiten ab 1.1.1880 00:00:00 Uhr bis 1.1.2021 00:00:00 Uhr.

Siehe auch `ctime()`, `getenv()`, `timezone`, `time.h`.

mmap - Speicherseiten abbilden

Name **mmap**

Syntax #include <sys/mman.h>

void *mmap(void **addr*, size_t *len*, int *prot*, int *flags*, int *fildes*, off_t *off*);

Beschreibung

`mmap()` stellt eine Abbildung zwischen dem Adressbereich eines Prozesses ([*pa, pa* + *len*)) und einem Dateiabschnitt her ([*off, off* + *len*)).

Der Aufruf hat folgendes Format:

```
pa = mmap(addr, len, prot, flags, fildes, off);
```

Zwischen dem Adressraum des Prozesses an der Adresse *pa* für *len* Bytes einerseits und der durch den Dateideskriptor *fildes* beschriebenen Datei mit dem Offset *off* für *len*-Bytes andererseits wird eine Abbildung hergestellt.
Der Wert von *pa* ist eine implementierungsabhängige Funktion von *addr* und dem Wert *flags*. Ein erfolgreicher Aufruf von `mmap()` liefert *pa* als Ergebnis zurück. Die Adressbereiche, die durch [*pa, pa* + *len*) und [*off, off* + *len*) definiert werden, müssen für den möglichen (nicht notwendigerweise den aktuellen) Adressbereich des Prozesses bzw. der Datei zulässig sein. `mmap()` kann eine Datei nicht vergrößern.

Die Abbildung, die durch `mmap()` mit `MAP_FIXED` hergestellt wird, ersetzt alle vorhergehenden Abbildungen für die Seiten des Prozesses im Bereich [*pa, pa* + *len*).

Wenn die Größe der abgebildeten Datei nach dem Aufruf von `mmap()` verändert wird, ist nicht festgelegt, welchen Effekt Referenzen auf Abbildungsteile haben, die zu einem neu hinzugekommenen oder gelöschten Teil der Datei korrespondieren.

`mmap()` wird nur für normale Dateien unterstützt.

Der Parameter *prot* bestimmt, ob gelesen, geschrieben, ausgeführt oder Kombinationen dieser Zugriffe auf die abgebildeten Seiten erlaubt werden sollen. Die Zugriffsrechte werden in `sys/mman.h` wie folgt definiert:

`PROT_READ`	Seite kann gelesen werden.
`PROT_WRITE`	Seite kann geschrieben werden.
`PROT_EXEC`	Seite kann ausgeführt werden.
`PROT_NONE`	auf Seite kann nicht zugegriffen werden.

`PROT_WRITE` ist als `PROT_WRITE|PROT_EXEC` implementiert und `PROT_EXEC` als `PROT_READ|PROT_EXEC`.

Drei Zustände sind möglich:

– Seite ist nicht zugreifbar

– auf die Seite kann nur lesend zugegriffen werden
– auf die Seite kann lesend und schreibend zugegriffen werden

Das Verhalten von PROT_WRITE kann durch die Option MAP_PRIVATE in dem Parameter *flags* beeinflusst werden, wie weiter unten noch näher beschrieben wird.

Der Parameter *flags* enthält weitere Informationen über die Behandlung der abgebildeten Seiten. Die Optionen werden in sys/mman.h wie folgt definiert:

MAP_SHARED	Änderungen sind gemeinsam benutzbar
MAP_PRIVATE	Änderungen sind privat
MAP_FIXED	*addr* ist exakt zu interpretieren

MAP_SHARED und MAP_PRIVATE kontrollieren die Sichtbarkeit von Schreibzugriffen auf die Speicherseiten. Es muss entweder MAP_SHARED oder MAP_PRIVATE angegeben werden. Der Abbildungstyp wird nach einem fork() beibehalten.

Wenn MAP_SHARED angegeben wird, ändern Schreibzugriffe auf die Speicherseiten die Datei, und die Änderungen sind in allen mit MAP_SHARED hergestellten Abbildungen des entsprechenden Dateiabschnittes sichtbar.

Wenn MAP_PRIVATE angegeben wird, ändern Schreibzugriffe auf die Speicherseiten nicht die Datei, und die Änderungen sind für keinen anderen Prozess sichtbar, der den entsprechenden Dateiabschnitt abbildet. Der erste Schreibzugriff erzeugt eine privat gehaltene Kopie der Speicherseiten und leitet die Abbildung auf die Kopie um. Beachten Sie, dass die privat gehaltene Kopie erst beim ersten Schreibzugriff erzeugt wird; bis dahin können andere Benutzer, die den Dateiabschnitt mit MAP_SHARED abgebildet haben, den Dateiabschnitt ändern.

MAP_FIXED legt fest, dass der Wert von *pa* genau *addr* entsprechen muss. Die Benutzung von MAP_FIXED wird nicht empfohlen, da dieser Parameter eine effektive Nutzung der Systemressourcen verhindern kann.

Wenn MAP_FIXED nicht gesetzt ist, wird implementierungsabhängig eine Adresse *pa* zurückgegeben, indem ein Bereich aus dem Adressraum des Prozesses ausgewählt wird, den das System zur Abbildung von *len*-Bytes für passend hält.
Ein *addr*-Wert von null bedeutet, dass *pa* unter Einhaltung der unten beschriebenen Bedingungen frei gewählt werden kann. Enthält *addr* einen Wert ungleich null, so wird dies als Vorschlag gewertet, die Abbildung nahe an dieser Adresse zu wählen.
In keinem Fall wird für *pa* der Wert 0 ausgewählt, eine bestehende Abbildung überschrieben oder in dynamisch zugewiesene Speicherbereiche abgebildet.

Der Parameter *off* unterliegt bezüglich Größe und Ausrichtung Beschränkungen, die sich nach dem Rückgabewert von sysconf() bezüglich der Parameter _SC_PAGESIZE und _SC_PAGE_SIZE richten. Wenn MAP_FIXED angegeben wird, muss der Parameter *addr* ebenfalls diese Beschränkungen einhalten.
Das System führt Abbildungsoperationen über ganze Seiten aus. Da der Parameter *len* nicht an bestimmte Größen oder Ausrichtungen gebunden ist, bezieht das System jede

Restseite, die bei der Abbildung des Bereiches [*pa, pa + len*) anfällt, mit in die Abbildungsoperation ein.
Das System füllt solche Teilseiten am Ende eines Speicherbereiches [*pa, pa + len*) mit Nullen. Veränderungen dieses Bereichs werden nicht zurückgeschrieben.
Falls sich die Abbildung auf ganze Seiten erstreckt, die hinter dem letzten Byte der Datei liegen, erzeugen Referenzen auf diese Seiten ein SIGSEGV-Signal.
SIGSEGV-Signale können außerdem bei verschiedenen Fehlerbedingungen des Dateisystems gesendet werden, einschließlich der Überschreitung des Quotas.

mmap() erzeugt eine zusätzliche Referenz auf die Datei, die durch *fildes* beschrieben wird. Diese Referenz wird bei einem close() auf *fildes* nicht gelöscht, sondern erst, wenn keine Abbildung mehr auf die Datei existiert.

Returnwert

pa	Adresse, an der die Abbildung platziert wurde.
-1	bei Fehler. errno wird gesetzt, um den Fehler anzuzeigen.

Fehler

mmap() schlägt fehl, wenn gilt:

EACCES	*fildes* ist nicht zum Lesen geöffnet, unabhängig von dem angegebenen *prot*-Argument, oder *fildes* ist nicht zum Schreiben geöffnet und bei einer Abbildung vom Typ MAP_SHARED wurde PROT_WRITE angefordert.
EAGAIN	Die Abbildung kann im Speicher nicht gesperrt werden.
EBADF	*fildes* ist kein gültiger offener Dateideskriptor.
ENXIO	Adressen im Bereich [*off, off + len*) sind für *fildes* ungültig.
EINVAL	Das Argument *off* (oder *addr*, wenn MAP_FIXED angegeben wurde) enthält kein Vielfaches der von sysconf() zurückgelieferten Seitenlänge, oder *off* bzw. *addr* hat einen ungültigen Wert. Der Wert in *flags* ist ungültig (weder MAP_PRIVATE noch MAP_SHARED ist gesetzt). Das Argument *len* hat einen Wert kleiner oder gleich 0.
EMFILE	Die Anzahl der Abbildungen überschreitet den maximal zulässigen Wert.
ENOMEM	MAP_FIXED wurde angegeben, und der Bereich [*addr, addr + len*) überschreitet den für einen Prozess erlaubten Adressbereich, oder MAP_FIXED wurde nicht angegeben, aber es steht nicht genügend Speicherplatz im Adressbereich für die Abbildung zur Verfügung.

ENODEV	*fildes* bezieht sich auf eine Datei, deren Typ von mmap() nicht unterstützt wird, wie zum Beispiel eine Gerätedatei.
EOVERFLOW	Der Wert von off plus len überschreitet das Offset-Maximum, das in der *fildes* zugeordneten internen Beschreibung der offenen Datei festgelegt ist.

Hinweis

Die Verwendung von mmap() verringert den Speicherplatz, der für andere Funktionen zur Verfügung steht, die ebenfalls Speicherplatz belegen.

Die Angabe MAP_FIXED wird nicht empfohlen, da dieser Parameter eine effektive Nutzung der Systemressourcen verhindern kann.

Die Anwendung muss auf eine Synchronisation der Dateizugriffe achten, wenn mmap() zusammen mit anderen Dateizugriffsmethoden wie read(), write(), Standardein/-ausgabe und shmat() verwendet wird.

mmap() erlaubt Zugriff auf Ressourcen über Adressbereichsmanipulationen an Stelle der read/write-Schnittstelle. Wird eine Datei abgebildet, muss ein Prozess lediglich auf die Adresse zugreifen, an die das Dateiobjekt abgebildet wird. Man betrachte den folgenden (unvollständigen) Code:

```
fildes = open(...)
lseek(fildes, some_offset)
read(fildes, buf, len)
   /* Daten in buf verwenden */
```

Unter Verwendung von mmap() kann der Code folgendermaßen umgeschrieben werden:

```
fildes = open(...)
address =mmap(0, len, PROT_READ, MAP_PRIVATE, fildes, some_offset)
   /* Daten über address verwenden */
```

Siehe auch

exec(), fcntl(), fork(), lockf(), munmap(), msync(), mprotect(), shmat(), sysconf(), sys/mman.h.

modf - Gleitkommazahl in ganzzahligen und gebrochenen Teil zerlegen

Syntax #include <math.h>

double modf(double *x*, double **iptr*);

Beschreibung

`modf()` zerlegt eine Gleitkommazahl *x* in ihren ganzzahligen und ihren gebrochenen Teil. Beide Teile erhalten das Vorzeichen von *x*. `modf()` liefert als Ergebnis den Bruchteil von *x* zurück und schreibt den ganzzahligen Teil als Wert vom Typ `double` an die Adresse, auf die *iptr* zeigt.

Returnwert Bruchteil von *x* mit Vorzeichen von *x* bei Erfolg.

0 bei Fehler.

Hinweis Das Argument *iptr* muss ein Zeiger sein!

Siehe auch `frexp()`, `ldexp()`, `math.h`.

mount - Dateisystem einhängen *(Erweiterung)*

Syntax

```
#include <sys/types.h>
#include <sys/mount.h>

int mount(const char *spec, const char *dir, int mflag,
          [int fstyp, const char *datptr, size_t datalen]);
```

Beschreibung

`mount()` hängt ein aushängbares Dateisystem, das sich in der durch *spec* gekennzeichneten blockorientierten Gerätedatei befindet, in das bestehende Dateiverzeichnis *dir* ein (Einhängepunkt).

spec und *dir* sind Zeiger auf Pfadnamen.

mflag kann folgende Werte annehmen:

`MS_FSS`	Wenn ein Dateisystemtyp beschrieben werden soll.
`MS_DATA`	Wenn ein Block dateisystemspezifischer Daten ab der Adresse *dataptr* mit der Länge *datalen* beschrieben werden soll.
`MS_RDONLY`	Wenn das eingehängte Dateisystem nur lesbar sein soll. Es werden keine weiteren Argumente erwartet.

fstyp wird von `mount()` ausgewertet, wenn entweder `MS_FSS` oder `MS_DATA` in *mflag* gesetzt ist. *fstyp* ist die Nummer des Dateisystemtyps oder ein Zeiger auf eine Zeichenkette, die den Dateisystemtyp enthält. Der Systemaufruf `sysfs()` kann zur Bestimmung der Nummer des Dateisystemtyps verwendet werden.

Wenn weder `MS_FSS` noch `MS_DATA` in *mflag* gesetzt ist, verwendet `mount()` den Dateisystemtyp des Root-Dateisystems.

Wenn `MS_DATA` in *mflag* gesetzt ist, erwartet das System die Argumente *dataptr* und *datalen*. Diese Daten werden von dateisystemspezifischem Code im Betriebssystem interpretiert; ihr Format hängt vom Dateisystemtyp ab. Ein Dateisystemtyp benötigt diese Daten möglicherweise nicht; in diesem Fall sollten sowohl *dataptr* als auch *datalen* auf 0 gesetzt werden.

Nach erfolgreicher Beendigung von `mount()` zeigt der Name in *dir* auf das Root-Dateiverzeichnis des neu eingehängten Dateisystems.

Returnwert

0	bei erfolgreicher Beendigung.
-1	bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `mount()` schlägt fehl, wenn gilt:

`EBUSY`	*dir* ist zum gegebenen Zeitpunkt bereits eingehängt, oder *dir* hat einem anderen Eigentümer, oder *dir* ist auf andere Weise belegt, oder die zu *spec* gehörende Gerätedatei ist gegenwärtig eingehängt, oder es stehen keine weiteren Einträge in der Einhängetabelle zur Verfügung.
`EFAULT`	*spec*, *dir* oder *datalen* weisen über den zugewiesenen Adressraum des Prozesses hinaus.
`EINVAL`	Der Superblock hat eine ungültige Magic Number, oder *fstyp* ist ungültig.
`ELOOP`	Während der Übersetzung von *dir* wurden zu viele symbolische Verweise angetroffen.
`ENAMETOOLONG`	Die Länge des Arguments *dir* ist größer als `{PATH_MAX}` oder `{NAME_MAX}`.
`ENOENT`	Eine der angegebenen Dateien ist unbekannt.
`ENOSPC`	Der Dateisystemstatus im Superblock ist nicht FsOKAY, und *mflag* fordert das Schreibrecht an.
`ENOTBLK`	*spec* ist keine blockorientierte Gerätedatei.
`ENOTDIR`	Eine Komponente von *spec* oder *dir* ist kein Dateiverzeichnis.
`ENXIO`	Die zu *spec* gehörende Gerätedatei ist unbekannt.
`EPERM`	Die effektive Benutzernummer ist nicht die eines Prozesses mit Sonderrechten.
`EREMOTE`	*spec* ist nicht lokal und kann nicht eingehängt werden.
`EROFS`	*spec* ist schreibgeschützt, und *mflag* fordert das Schreibrecht an.

Hinweis `mount()` darf nur unter der effektiven Benutzernummer eines Prozesses mit Sonderrechten aufgerufen werden.

Sobald ein Dateiverzeichnis eingehängt ist, wird es wie ein Unterbaum behandelt. Prozesse können nun auf Dateien im eingehängten Dateisystem zugreifen, ohne berücksichtigen zu müssen, dass es ein eingehängtes Dateisystem ist. Lediglich Verweise mit `link()` über Dateisystemgrenzen hinweg sind nicht gestattet, da diese Funktion das Dateisystem einer Datei überprüft.

Die Schnittstelle ist nur für das `mount`-Kommando vorgesehen.

Siehe auch `sysfs()`, `umount()`, Kommandos `mount`, `fsck` im Handbuch „POSIX Kommandos (BS2000/OSD)“.

mprotect - Zugriffsschutz für Speicherabbildung ändern

Syntax

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

int mprotect(void *addr, size_t len, int prot);
```

Beschreibung

Die Funktion `mprotect()` ändert die Zugriffsrechte für die Abbildungen im Bereich [*addr, addr + len*) auf das in *prot* angegebene Zugriffsrecht. Der in *len* angegebene Wert wird dabei auf ein Vielfaches der durch `sysconf()` vorgegebenen Seitengröße gerundet.
Für *prot* sind alle Werte zulässig, die auch in `mmap()` angegeben werden können:

Die Werte für *prot* sind in `sys/mman.h` wie folgt definiert:

`PROT_READ`	Seite kann gelesen werden.
`PROT_WRITE`	Seite kann geschrieben werden.
`PROT_EXEC`	Seite kann ausgeführt werden.
`PROT_NONE`	auf Seite kann nicht zugegriffen werden.

Falls `mprotect()` fehlschlägt, die Ursache aber nicht EINVAL ist, kann es sein, dass die Zugriffsrechte einiger Seiten in dem angegebenen Bereich [*addr, addr + len*) bereits geändert wurden. Wenn der Fehler an der Adresse *addr2* auftritt, dann werden die Zugriffsrechte aller ganzen Seiten im Bereich [*addr, addr2*] verändert.

Returnwert

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

Fehler

Unter den folgenden Bedingungen schlägt die Funktion `mprotect()` fehl und setzt `errno` auf die folgenden Werte:

`EACCES`	*prot* enthält einen Wert, der nicht zu den Zugriffsrechten des Prozesses auf die zu Grunde liegende Datei passt.
`EAGAIN`	*prot* enthält den Wert `PROT_WRITE` für eine Abbildung vom Typ `MAP_PRIVATE` und es tritt ein Speicherengpass auf, d.h. die Speicherressourcen zum Reservieren und Sperren der privaten Seite reichen nicht aus.
`EINVAL`	*addr* ist kein Vielfaches der durch `sysconf()` vorgegebenen Seitengröße oder das Argument *len* enthält einen Wert kleiner oder gleich 0.
`ENOMEM`	Adressen im Bereich [*addr, addr + len*) sind für den Adressbereich des Prozesses ungültig, oder es sind eine oder mehrere Seiten angegeben, welche nicht abgebildet sind.

Siehe auch `mmap()`, `sysconf()`, `sys/mman.h`.

mrand48 - Pseudo-Zufallszahlen zwischen -2^{31} und 2^{31} generieren

Syntax #include <stdlib.h>

long int mrand48 (void);

Beschreibung

Siehe `drand48()`.

msgctl - Steueroperationen für Nachrichten liefern

Syntax

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

int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

Beschreibung

`msgctl()` liefert Operationen für die Nachrichtensteuerung, die durch *cmd* angegeben werden. Die möglichen Werte für *cmd* und die dazugehörenden Nachrichtensteuerungs-Operationen sind:

`IPC_STAT` Die aktuellen Werte aller Elemente der *msqid* zugeordneten Datenstruktur werden in die Struktur eingetragen, auf die mit *buf* verwiesen wird. Der Inhalt dieser Struktur wird in `sys/msg.h` definiert.

`IPC_SET` Die Werte folgender Elemente der *msqid* zugeordneten Datenstruktur des Typs `msgqid_ds` werden auf die entsprechenden Werte aus der Struktur gesetzt, auf die *buf* zeigt:

```
msg_perm.uid
msg_perm.gid
msg_perm.mode
msg_qbytes
```

`IPC_SET` kann nur von einem Prozess mit Sonderrechten ausgeführt werden, oder einem Prozess, dessen effektive Benutzernummer gleich dem Wert von `msg_perm.cuid` oder `msg_perm.uid` in der `msqid_ds`-Struktur ist, die *msqid* zugeordnet ist. Nur ein Prozess mit Sonderrechten kann `msg_qbytes` erhöhen.

`IPC_RMID` Gelöscht werden die mit *msqid* angegebene Warteschlangenkennzahl sowie die Warteschlange samt zugeordneter Datenstruktur. `IPC_RMID` kann nur von einem Prozess ausgeführt werden, der über Sonderrechte verfügt oder dessen effektive Benutzernummer mit `msg_perm.cuid` bzw. `msg_perm.uid` in der `msquid_ds`-Struktur zu *msqid* übereinstimmt.

Returnwert

0 bei Erfolg.

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

Fehler `msgctl()` schlägt fehl, wenn gilt:

`EACCES` *cmd* ist `IPC_STAT` und der aufrufende Prozess hat kein Leserecht.

Erweiterung
`EFAULT` *buf* zeigt auf eine unzulässige Adresse. ❑

EINVAL	*msqid* ist keine gültige Kennzahl einer Nachrichten-Warteschlange, oder *cmd* ist keine gültige Operation, oder *cmd* ist IPC_SET und msg_perm.uid oder msg_perm.gid sind ungültig.
EPERM	*cmd* ist IPC_SET und die effektive Benutzernummer des aufrufenden Prozesses ist nicht gleich der eines Prozesses mit Sonderrechten und nicht gleich dem Wert von msg_perm.cuid oder msg_perm.uid in der *msqid* zugeordneten Datenstruktur.
EPERM	*cmd* ist IPC_SET, ein Versuch wurde gemacht, den Wert von msg_qbytes zu erhöhen, und die effektive Benutzernummer des aufrufenden Prozesses besitzt keine Sonderrechte.

Hinweis Das Kommittee des IEEE 1003.4-Standards entwickelt gerade eine alternative Schnittstelle für die Interprozeßkommunikation. Anwendungsprogrammierer, die Interprozeßkommunikation einsetzen, sollten die Anwendungen so konzipieren, daß Module, die derzeit beschriebene Funktionen für Interprozeßkommunikation benutzen, einfach geändert werden können.

Siehe auch msgget(), msgrcv(), msgsnd(), sys/msg.h, Abschnitt „Interprozesskommunikation“ auf Seite 116.

msgget - Nachrichten-Warteschlange ermitteln

Syntax

#include <sys/msg.h>

int msgget(key_t *key*, int *msgflg*);

Beschreibung

`msgget()` liefert die Warteschlangenkennzahl, die *key* zugeordnet ist.

Warteschlangenkennzahl, zugehörige Warteschlange und Datenstruktur (siehe auch `sys/msg.h`) werden für *key* dann erzeugt, wenn eine der folgenden Bedingungen erfüllt ist:

– *key* ist `IPC_PRIVATE`.

– *key* besitzt noch keine zugeordnete Warteschlangenkennzahl und `(`*msgflg* `& IPC_CREAT)` ist ungleich 0.

Bei der Erzeugung wird die der neuen Warteschlangenkennzahl zugeordnete Datenstruktur wie folgt initialisiert:

– `msg_perm.cuid`, `msg_perm.uid`, `msg_perm.cgid` und `msg_perm.gid` werden gleich der effektiven Benutzer- bzw. Gruppennummer des aufrufenden Prozesses gesetzt.

– Die niederwertigen 9 Bit von `msg_perm.mode` werden gleich den niederwertigen 9 Bit von *msgflg* gesetzt.

– `msg_qnum`, `msg_lspid`, `msg_lrpid`, `msg_stime` und `msg_rtime` werden gleich 0 gesetzt.

– `msg_ctime` wird gleich der aktuellen Uhrzeit gesetzt.

– `msg_qbytes` wird gleich der durch das System festgelegten Grenze gesetzt.

Returnwert

nichtnegative ganze Zahl (Warteschlangenkennzahl) bei Erfolg.

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

Fehler

`msgget()` schlägt fehl, wenn gilt:

`EACCES` Es existiert eine Warteschlangenkennzahl für das Argument *key*, aber die in den niederwertigsten 9 Bit von *msgflg* angegebenen Zugriffsrechte werden nicht erteilt (siehe auch Abschnitt „Interprozesskommunikation“ auf Seite 116).

`EEXIST` Es existiert eine Warteschlangenkennzahl für das Argument *key*, aber der Wert von `((`*msgflg* `& IPC_CREAT) && (`*msgflg* `& IPC_EXCL))` ist ungleich 0.

`ENOENT` Es existiert keine Warteschlangenkennzahl für das Argument *key* und `(`*msgflg* `& IPC_CREAT)` ist gleich 0.

`ENOSPC` Es soll eine Warteschlangenkennzahl erzeugt werden, aber die durch das System festgelegte Grenze für die Maximalzahl der erlaubten Warteschlangenkennzahlen würde dadurch überschritten werden.

Hinweis Das Kommittee des IEEE 1003.4-Standards entwickelt gerade eine alternative Schnittstelle für die Interprozeßkommunikation. Anwendungsprogrammierer, die Interprozeßkommunikation einsetzen, sollten die Anwendungen so konzipieren, daß Module, die derzeit beschriebene Funktionen für Interprozeßkommunikation benutzen, einfach geändert werden können.

Siehe auch `msgctl()`, `msgrcv()`, `msgsnd()`, `sys/msg.h`, Abschnitt „Interprozesskommunikation“ auf Seite 116.

msgrcv - Nachricht aus Warteschlange empfangen

Syntax #include <sys/msg.h>

int msgrcv(int *msqid*, void **msgp*, size_t *msgsz*, long int *msgtyp*, int *msgflg*);

Beschreibung

`msgrcv()` liest eine Nachricht aus der Warteschlange, der die durch *msqid* angegebene Warteschlangenkennzahl zugeordnet ist, und legt diese in dem vom Benutzer definierten Puffer ab, auf den *msgp* zeigt.

msgp zeigt auf einen vom Benutzer definierten Puffer, der zunächst eine Komponente des Typs `long int` für den Nachrichtentyp und dann einen Datenbereich für die Datenbytes der Nachricht enthalten muss. Die nachstehende Struktur ist ein Beispiel dafür, wie dieser vom Benutzer definierte Puffer aussehen könnte:

```
struct mymsg
{
   long int mtype;       /* Nachrichtentyp  */
   char mtext[1];        /* Nachrichtentext */
}
```

Die Strukturkomponente `mtype` ist der Nachrichtentyp der empfangenen Nachricht, wie durch den sendenden Prozess angegeben.

Die Strukturkomponente `mtext` ist der Nachrichtentext.

msgsz gibt die Größe von `mtext` in Bytes an. Wenn die empfangene Nachricht länger als *msgsz* und `(`*msgflg* `& MSG_NOERROR)` ungleich 0 ist, wird sie auf *msgsz* Bytes gekürzt. Der abgeschnittene Teil der Nachricht geht verloren; dem aufrufenden Prozess wird dies nicht mitgeteilt.

msgtyp gibt den Typ der geforderten Nachricht wie folgt an:

– wenn *msgtyp* gleich 0 ist, wird die erste Nachricht in der Nachrichtenwarteschlange empfangen;
– wenn *msgtyp* größer als 0 ist, wird die erste Nachricht des Typs *msgtyp* empfangen;
– wenn *msgtyp* kleiner als 0 ist, wird die erste Nachricht kleiner oder gleich dem Absolutwert von *msgtyp* empfangen.

msgflg gibt an, welche Aktion ausgeführt werden soll, wenn sich keine Nachricht des geforderten Typs in der Warteschlange befindet. Folgende Aktionen sind möglich:

– Wenn `(`*msgflg* `& PC_NOWAIT)` ungleich 0 ist, kehrt die Funktion sofort mit dem Ergebnis -1 zum aufrufenden Prozess zurück und `errno` ist gleich `ENOMSG` gesetzt.

– Wenn `(`*msgflg* `& IPC_NOWAIT)` gleich 0 ist, unterbricht der aufrufende Prozess seine Ausführung, bis eines der folgenden Ereignisse eintritt:
  – Eine Nachricht des geforderten Typs wird in die Warteschlange eingetragen.

- Die Warteschlangenkennzahl *msqid* wird aus dem System entfernt; wenn dies geschieht, wird `errno` gleich `EIDRM` gesetzt und das Ergebnis -1 wird zurückgeliefert.
- Der aufrufende Prozess empfängt ein abzufangendes Signal; in diesem Fall wird die Nachricht nicht empfangen und der Prozess setzt seine Ausführung so fort, wie dies unter `sigaction()` beschrieben ist.

Bei erfolgreicher Beendigung werden die folgenden Aktionen auf der *msqid* zugeordneten Datenstruktur ausgeführt:

- `msg_qnum` wird um 1 vermindert.
- `msg_lrpid` wird gleich der Prozessnummer des aufrufenden Prozesses gesetzt.
- `msg_rtime` wird auf die aktuelle Zeit gesetzt.

Werden Threads verwendet, so wirkt sich die Funktion auf den Prozess oder auf einen Thread wie folgt aus: Der Parameter *msgflg* bezieht sich auf den aufrufenden Thread.

Returnwert

Anzahl der in `mtext` abgelegten Bytes	bei Erfolg.
-1	bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `msgrcv()` schlägt fehl, wenn gilt:

`E2BIG`	Der Wert von `mtext` ist größer als *msgsz*, und `(`*msgflg* `&` `MSG_NOERROR)` ist gleich 0.
`EACCES`	Der aufrufende Prozess erhält keine Erlaubnis für diese Operation.
Erweiterung `EFAULT`	*msgp* verweist auf eine unzulässige Adresse. ❑
`EIDRM`	Die Warteschlangenkennzahl *msqid* wurde aus dem System entfernt.
`EINTR`	`msgrcv()` wurde durch ein Signal unterbrochen.
`EINVAL`	*msqid* ist keine gültige Warteschlangenkennzahl, oder der Wert von *msgsz* ist kleiner als 0.
`ENOMSG`	Die Warteschlange enthält keine Nachricht des geforderten Typs, und `(`*msgtyp* `&` `IPC_NOWAIT)` ist ungleich 0.

Hinweis *msgp* sollte in den Typ `void *` umgewandelt werden.

Das Kommittee des IEEE 1003.4-Standards entwickelt gerade eine alternative Schnittstelle für die Interprozeßkommunikation. Anwendungsprogrammierer, die Interprozeßkommunikation einsetzen, sollten die Anwendungen so konzipieren, daß Module, die derzeit beschriebene Funktionen für Interprozeßkommunikation benutzen, einfach geändert werden können.

Siehe auch `msgctl()`, `msgget()`, `msgsnd()`, `sigaction()`, `sys/msg.h`,
Abschnitt „Interprozesskommunikation“ auf Seite 116.

msgsnd - Nachricht an Warteschlange senden

Syntax #include <sys/msg.h>

int msgsnd(int *msqid*, const void **msgp*, size_t *msgsz*, int *msgflg*);

Beschreibung

`msgsnd()` sendet eine Nachricht an die Warteschlange, die durch die Warteschlangenkennzahl *msqid* angegeben wird.

msgp zeigt auf einen benutzerdefinierten Puffer, der eine Komponente des Typs `long int` für den Nachrichtentyp und einen Datenbereich für die Bytes der Nachricht enthalten muss. Die nachstehende Struktur ist ein Beispiel dafür, wie der benutzerdefinierte Puffer aussehen könnte:

```
struct mymsg
{
   long int mtype;       /* Nachrichtentyp  */
   char mtext[1];        /* Nachrichtentext */
}
```

Die Strukturkomponente `mtype` ist ein von 0 verschiedener Wert vom Typ `long int`, der vom empfangenden Prozess zur Nachrichtenauswahl verwendet werden kann.

Die Strukturkomponente `mtext` ist ein Text der Länge *msgsz* Bytes. *msgsz* kann von 0 bis zu einem systembedingten Grenzwert reichen.

msgflg gibt an, welche Aktion ausgeführt werden soll, wenn eine oder mehrere der folgenden Bedingungen erfüllt sind:

– Die Anzahl der Bytes in der Warteschlange ist bereits gleich `msg_qbytes` (siehe auch `sys/msg.h`).

– Die Gesamtzahl der Nachrichten in allen Warteschlangen des Systems ist bereits gleich der durch das System festgelegten Grenze.

Folgende Aktionen können dann ausgeführt werden:

– Wenn `(`*msgflg* `& IPC_NOWAIT)` ungleich 0 ist, wird keine Nachricht gesendet und die Funktion kehrt sofort zum aufrufenden Prozess zurück.

– Wenn `(`*msgflg* `& IPC_NOWAIT)` gleich 0 ist, unterbricht der aufrufende Prozess seine Ausführung, bis eines der folgenden Ereignisse eintritt:

  – Die Bedingung, die für die Unterbrechung verantwortlich war, existiert nicht mehr; in diesem Fall wird die Nachricht gesendet.

  – Die Warteschlangenkennzahl *msqid* wird aus dem System entfernt; wenn dies geschieht, wird `errno` gleich `EIDRM` gesetzt und der Returnwert -1 zurückgeliefert.

– Der aufrufende Prozess empfängt ein abzufangendes Signal; in diesem Fall wird die Nachricht nicht gesendet und der Prozess setzt seine Ausführung so fort, wie dies unter `sigaction()` beschrieben wird.

Bei erfolgreicher Beendigung werden die folgenden Aktionen auf der *msqid* zugeordneten Datenstruktur ausgeführt:

– `msg_qnum` wird um 1 erhöht

– `msg_lspid` wird gleich der Prozessnummer des aufrufenden Prozesses gesetzt

– `msg_stime` wird auf die aktuelle Zeit gesetzt.

Werden Threads verwendet, so wirkt sich die Funktion auf den Prozess oder auf einen Thread wie folgt aus: Der Parameter *msgflg* bezieht sich auf den aufrufendenThread.

Returnwert

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

Fehler `msgsnd()` schlägt fehl, wenn gilt:

`EACCES`	Der aufrufende Prozess erhält keine Erlaubnis für diese Operation.
`EAGAIN`	Die Nachricht kann aus einem der oben genannten Gründe nicht gesendet werden und `(`*msgflg* `& IPC_NOWAIT)` ist ungleich 0.
Erweiterung `EFAULT`	*msgp* verweist auf eine unzulässige Adresse. ❑
`EIDRM`	Die Warteschlangenkennzahl *msqid* wurde aus dem System entfernt.
`EINTR`	`msgsnd()` wurde durch ein Signal unterbrochen.
`EINVAL`	*msqid* ist keine gültige Warteschlangenkennzahl, oder der Wert von `mtype` ist kleiner als 0, oder der Wert von *msgsz* ist kleiner als 0 oder größer als der systembedingte Grenzwert.

Hinweis Der Wert des Arguments *msgp* sollte in den Typ `void *` umgewandelt werden.

Das Kommittee des IEEE 1003.4-Standards entwickelt gerade eine alternative Schnittstelle für die Interprozeßkommunikation. Anwendungsprogrammierer, die Interprozeßkommunikation einsetzen, sollten die Anwendungen so konzipieren, daß Module, die derzeit beschriebene Funktionen für Interprozeßkommunikation benutzen, einfach geändert werden können.

Siehe auch `msgctl()`, `msgget()`, `msgrcv()`, `sigaction()`, `sys/msg.h`,
Abschnitt „Interprozesskommunikation“ auf Seite 116.

msync - Speicher synchronisieren

Syntax #include <sys/mman.h>

int msync(void *_addr_, size_t _len_, int _flags_);

Beschreibung

Die Funktion `msync()` schreibt alle veränderten Kopien von Seiten im Bereich [*addr, addr* + *len*) auf die zugehörigen Speichermedien zurück oder macht Kopien im Speicher ungültig, so dass bei späteren Zugriffen auf diese Seiten auf das Speichermedium zugegriffen wird.
Das Speichermedium für eine veränderte Abbildung vom Typ `MAP_SHARED` ist die Datei, auf die die Seite abgebildet wird; das Speichermedium für eine veränderte Abbildung vom Typ `MAP_PRIVATE` ist ihr Paging-Bereich.

flags muss einen der folgenden Werte haben:

`MS_ASYNC`	asynchrone Schreibzugriffe durchführen
`MS_SYNC`	synchrone Schreibzugriffe durchführen
`MS_INVALIDATE`	Abbildungen als ungültig markieren

Wenn `MS_ASYNC` oder `MS_SYNC` gesetzt sind, synchronisiert `msync()` den Dateiinhalt mit dem aktuellen Inhalt des zugeordneten Speicherbereichs:
Alle Schreibzugriffe auf den Speicherbereich, die vor dem Aufruf von `msync()` stattgefunden haben, sind nach `msync()` bei Lesezugriffen auf die Datei sichtbar.
Vor dem Aufruf von `msync()` ist es dagegen undefiniert, ob Schreibzugriffe auf den entsprechenden Dateiabschnitt bei anschließenden Lesezugriffen sichtbar sind.

Wenn `MS_ASYNC` gesetzt ist, kehrt `msync()` zurück, sobald alle Schreiboperationen veranlasst wurden; wird `MS_SYNC` gesetzt, kehrt `msync()` erst dann zurück, wenn alle Schreiboperationen durchgeführt wurden.

Wenn `MS_INVALIDATE` gesetzt ist, synchronisiert `msync()` den Speicherbereich mit dem aktuellen Inhalt des zugeordneten Dateiabschnitts. Anschließend werden alle Kopien von Daten, die sich in einem Cache-Speicher befinden, als ungültig markiert. Spätere Referenzen auf diese Seiten werden vom System über das zu Grunde liegende Speichermedium bedient.
Alle Schreibzugriffe auf den abgebildeten Dateiabschnitt, die vor dem Aufruf von `msync()` stattgefunden haben, sind bei anschließenden Lesezugriffen auf den zugeordneten Speicherbereich sichtbar.
Vor dem Aufruf von `msync()` ist es dagegen undefiniert, ob Schreibzugriffe auf den entsprechenden Dateiabschnitt bei anschließenden Lesezugriffen sichtbar sind.

Returnwert

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

Fehler Unter den folgenden Bedingungen schlägt die Funktion `msync()` fehl und setzt `errno` auf die folgenden Werte:

`EINVAL`	*addr* ist kein Vielfaches der durch `sysconf()` vorgegebenen Seitengröße.
`ENOMEM`	Adressen im Bereich [*addr, addr* + *len*) sind für den Adressbereich des Prozesses ungültig, oder es sind eine oder mehrere Seiten angegeben, welche nicht abgebildet sind.
`EIO`	Beim Lese- oder Schreibzugriff auf die Datei trat ein Ein-Ausgabefehler auf.

Hinweis `msync()` sollte verwendet werden, wenn verlangt wird, dass sich ein Speicherobjekt in einem bekannten Zustand befindet, z.B. bei Transaktionsverarbeitung.

Auch im Zuge normaler Systemabläufe können Speicherseiten auf Platte geschrieben werden. Es kann daher nicht garantiert werden, dass nur beim Aufruf von `msync()` Speicherseiten auf Platte geschrieben werden.

Siehe auch `mmap()`, `sysconf()`, `sys/mman.h`

munmap - Abbildung von Speicherseiten aufheben

Syntax

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

int munmap(void *addr, size_t len);
```

Beschreibung

Die Funktion `munmap()` entfernt Abbildungen von Seiten im Bereich [*addr, addr + len*). Der in *len* angegebene Wert wird dabei auf ein Vielfaches der durch `sysconf()` vorgegebenen Seitengröße gerundet.
Weitere Referenzen auf diese Seiten resultieren in einem `SIGSEGV`-Signal an den Prozess, sofern nicht zwischenzeitlich eine neue Abbildung dieser Seiten etabliert wurde.

Bereiche innerhalb des angegebenen Intervalls, die keine `mmap`-Abbildungen sind, werden ignoriert.

Returnwert

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

Fehler `munmap()` schlägt fehlt, wenn gilt:

`EINVAL`	*addr* ist kein Vielfaches der durch `sysconf()` vorgegebenen Seitengröße oder Adressen im Bereich [*addr, addr + len*) sind für den Adressbereich des Prozesses ungültig oder das Argument *len* enthält einen Wert kleiner oder gleich 0.

Siehe auch `mmap()`, `sysconf()`, `signal.h`, `sys/mman.h`.

Funktionen und Variablen alphabetisch (n - y)

nanosleep - aktuellen Thread suspendieren

Syntax #include <time.h>

int nanosleep(const struct timespec * *rqtp*, struct timespec * *rmtp*);

Beschreibung

Die Funktion `nanosleep()` suspendiert den aktuellen Thread, bis entweder das durch *rqtp* angegebene Zeitintervall abgelaufen ist oder bis dem rufenden Thread ein Signal zugestellt wurde, dessen Aktion es ist, eine Signalbehandlungsroutine aufzurufen oder den Prozess zu beenden. Die Zeit der Suspendierung kann eventuell länger sein als angegeben, weil der Wert auf ein Vielfaches der sleep resolution aufgerundet wurde oder weil das System noch andere Aktivitäten ausführt.

Returnwert

0	wenn die angegebene Zeit abgelaufen ist.
- 1	wenn `nanosleep()` von einem Signal unterbrochen wurde. Wenn *rmtp* kein Nullzeiger ist, wird in diesem Fall außerdem die verbleibende Zeit in der Struktur, auf die *rmtp* zeigt, abgelegt. Ist *rmtp* NULL, wird die verbleibende Zeit nicht zurückgegeben. `errno` gesetzt, um den Fehler anzuzeigen.

Fehler `nanosleep()` schlägt fehl, wenn gilt:

`EINTR`	`nanosleep()` wurde von einem Signal unterbrochen.
`EINVAL`	Im Argument *rqtp* ist ein Wert in Nanosekunden angegeben, der kleiner als 0 oder größer/gleich 1000 Millionen ist.
`ENOSYS`	Die Funktion `nanosleep()` wird in dieser Implementierung nicht unterstützt.

Siehe auch `sleep()`, `time.h` .

nextafter - nächste darstellbare Gleitpunktzahl

Syntax #include <math.h>

double nextafter (double *x*, double *y*);

Beschreibung

`nextafter()` liefert die nächste darstellbare Gleitkommazahl, die in Richtung *y* auf *x* folgt. Wenn *y* kleiner als *x* ist, wird die größte darstellbare Gleitkommazahl zurückgeliefert, die kleiner als *x* ist.

Returnwert nächste darstellbare Gleitpunktzahl, die in Richtung *y* auf *x* folgt
bei Erfolg.

Wenn *x* endlich ist, aber das Ergebnis von `nextafter(x, y)` einen Überlauf verursachen würde, wird der Wert HUGE_VAL zurückgegeben und `errno` auf `ERANGE` gesetzt.

Fehler `nextafter()` schlägt fehlt, wenn gilt:

`ERANGE` der korrekte Wert würde einen Überlauf verursachen.

Siehe auch `math.h`.

nftw - Dateibaum durchwandern

Syntax #include <ftw.h>

int nftw (const char *_path_,
int (*_fn_) (const char *, const struct stat *, in , struct FTW *),
int _depth_, int _flags_);

Beschreibung

nftw() durchsucht rekursiv die Dateiverzeichnis-Hierarchie, die mit *path* beginnt. nftw() arbeitet ähnlich wie ftw(), verarbeitet aber zusätzlich das Argument *flags*, das durch bitweises inklusives ODER der folgenden Werte gebildet wird:

FTW_CHDIR das jeweils durchsuchte Verzeichnis wird zum aktuellen Arbeitsverzeichnis. Ist FTW_CHDIR nicht gesetzt, bleibt das aktuelle Arbeitsverzeichnis unverändert.

FTW_DEPTH vor dem Verzeichnis selbst werden erst alle Unterverzeichnisse durchwandert. Ist FTW_DEPTH nicht gesetzt, wird erst das Verzeichnis durchwandert.

FTW_MOUNT es werden nur Verzeichnisse durchwandert, die in demselben Dateisystem liegen wie *path*. Ist FTW_MOUNT nicht gesetzt, werden auch gemountete Verzeichnisse durchwandert.

FTW_PHYS Die Dateiverzeichnis-Hierarchie wird physikalisch durchwandert; nftw() folgt keinen symbolischen Verweisen, sondern meldet die Verweise. Ist FTW_PHYS nicht gesetzt, folgt nftw() symbolischen Verweisen. nftw() meldet nicht zweimal die gleiche Datei.

Für jede gefundene Datei bzw. jedes gefundene Verzeichnis ruft nftw() die benutzerdefinierte Funktion *fn* mit folgenden vier Argumenten auf:

1. Pfadname des Objekts.
2. Zeiger auf den stat-Puffer, der Informationen über das Objekt enthält.
3. Zahl vom Typ Integer, in der nftw() zusätzliche Informationen liefert.

   FTW_F Das Objekt ist eine Datei.

   FTW_D Das Objekt ist ein Verzeichnis.

   FTW_DP Das Objekt ist ein Verzeichnis, Unterverzeichnisse wurden bereits durchwandert (dieser Fall kann nur auftreten, wenn in *flags* der Wert FTW_DEPTH enthalten ist).

   FTW_SLN Das Objekt ist ein symbolischer Verweis, der auf eine nicht vorhandene Datei zeigt (dieser Fall kann nur auftreten, wenn in *flags* nicht der Wert FTW_PHYS enthalten ist).

`FTW_DNR` Das Objekt ist ein Verzeichnis, das nicht gelesen werden kann. *fn*() wird für keine der darin liegenden Dateien oder darunter liegenden Verzeichnisse aufgerufen.

`FTW_NS` `stat()` kann das Objekt auf Grund unzureichender Zugriffsrechte nicht bearbeiten. Der an *fn* übergebene `stat`-Puffer ist undefiniert. Wenn `stat()` aus anderen Gründen scheitert, schlägt `nftw()` fehl und gibt -1 zurück.

4. Zeiger auf ein `struct FTW`, das die folgenden Elemente enthält:

```
int base;
int level;
```

`nftw()` verwendet jeweils einen Dateideskriptor für jede Ebene im Dateibaum. Das Argument *depth* begrenzt die Anzahl der verwendeten Dateideskriptoren. Ist *depth* 0 oder negativ, hat das die gleiche Wirkung wie der Wert 1. *depth* darf nicht größer sein als die Anzahl der zum gegebenen Zeitpunkt zur Verfügung stehenden Dateideskriptoren. Wenn die Funktion `nftw()` zurückkehrt, schließt sie alle Dateideskriptoren, die sie geöffnet hat; sie schließt aber keine Dateideskriptoren, die von *fn* geöffnet wurden.

Der Dateibaum wird von der obersten Hierarchiestufe an durchwandert, bis der Baum vollständig durchwandert ist, ein Aufruf von *fn* einen Wert ungleich 0 zurückgibt oder ein Fehler innerhalb `nftw ()` (wie z.B. ein E/A-Fehler) festgestellt wird.

Returnwert

0 wenn der Baum vollständig durchwandert ist und *fn*() immer den Wert 0 zurückgeliefert hat.

Rückgabewert der Funktion *fn*()
wenn *fn()* einen Wert ≠ 0 zurückgibt, stoppt `nftw()` das Durchwandern des Dateibaums und gibt den Wert zurück, der von *fn* zurückgegeben wurde

-1 wenn `nftw()` einen anderen Fehler als `EACCES` feststellt. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `nftw()` schlägt fehlt, wenn gilt:

`EACCES` Für eine Komponente von *path* besteht kein Suchrecht oder
für *path* besteht kein Leserecht oder
fn() gibt den Wert -1 zurück und setzt nicht zurück.

`ENAMETOOLONG`
Die Länge des Arguments *path* ist größer als `{PATH_MAX}` oder eine Komponente des Pfadnamens ist länger als `{NAME_MAX}`.

Bei der Auflösung eines symbolischen Verweises in *path* kam es zu einem Zwischenergebnis, dessen Länge `{PATH_MAX}` überschreitet.

ENOENT	Eine Komponente des Pfadpräfixes existiert nicht oder *path* ist eine leere Zeichenkette.
ENOTDIR	Eine Komponente von *path* ist kein Dateiverzeichnis.
ELOOP	Bei der Auflösung von *path* traten zuviele symbolische Verweise auf.
EMFILE	Es sind bereits {OPEN_MAX} Dateideskriptoren geöffnet.
ENFILE	Es sind zu viele Dateien geöffnet.

Außerdem kann errno gesetzt sein, wenn die Funktion, auf die *fn*() zeigt, errno setzt.

Hinweis Da nftw() rekursiv ist, besteht die Möglichkeit, dass es mit einem Speicherfehler abbricht, wenn es auf Dateibäume mit zu vielen Hierarchieebenen angewendet wird.

Siehe auch lstat(), opendir(), readdir(), stat(), ftw.h.

nice - Priorität eines Prozesses ändern

Syntax

```
#include <unistd.h>

int nice(int incr);
```

Beschreibung

`nice()` addiert den Wert von *incr* auf den Prioritätswert des aufrufenden Prozesses. Im C-Laufzeitsystem hat die Veränderung von *incr* jedoch keine Auswirkung auf die Priorität eines Prozesses. Die Funktion wird nur aus Kompatibilitätsgründen zu XPG4 angeboten.

Ein Prioritätswert ist eine nichtnegative ganze Zahl, bei der aus einem höheren Wert eine niedrigere Prozessor-Priorität resultiert. Ein maximaler Prioritätswert von 2*{NZERO}-1 und ein minimaler Prioritätswert von 0 werden durch das System festgelegt (siehe `limits.h`). Anforderungen für Werte oberhalb oder unterhalb dieser Grenzen bewirken, dass der Prioritätswert auf den entsprechenden Grenzwert gesetzt wird. Nur ein Prozess mit geeigneten Zugriffsrechten kann den Prioritätswert erniedrigen.

Werden Threads verwendet, so wirkt sich die Funktion auf den Prozess oder auf einen Thread wie folgt aus: Ändert die Priorität eines Prozesses. Wenn der Prozess ”multithreaded” ist, wirkt sich die Scheduling-Priorität auf alle Threads des Prozesses mit system scope aus.

Returnwert

neuer Prioritätswert abzüglich {NZERO}	bei erfolgreicher Beendigung.
-1	bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen. Der Prioritätswert des aufrufenden Prozesses bleibt unverändert.

Fehler

`nice()` schlägt fehl, wenn gilt:

`EPERM`	*incr* ist negativ oder größer als 2*{NZERO}-1 und der aufrufende Prozess besitzt keine Sonderrechte.

Hinweis

Da bei Erfolg auch -1 zurückgeliefert werden kann, sollte eine Anwendung zur Überprüfung von Fehlersituationen `errno` vor dem Aufruf von `nice()` gleich 0 setzen und dann `nice()` aufrufen. Wenn -1 zurückgeliefert wird, dann sollte sie prüfen, ob `errno` ungleich 0 ist.

Siehe auch

`limits.h`, `limits.h`, `unistd.h`.

nl_langinfo - Lokalitätswerte ermitteln

Syntax #include <langinfo.h>

char *nl_langinfo(nl_item *item*);

Beschreibung

`nl_langinfo()` liefert den Wert, den die Konstante *item* in der aktuellen Lokalität oder Umgebung besitzt. Die verfügbaren Konstanten und Werte für *item* sind in `langinfo.h` definiert.

Returnwert Zeiger auf eine Zeichenkette der Lokalität
wenn in einer Umgebung keine `langinfo`-Daten definiert sind.

Nullzeiger wenn *item* ungültig ist.

Hinweis Der Vektor, auf den der Returnwert zeigt, sollte vom Programm nicht verändert werden, aber weitere Aufrufe von `nl_langinfo()` können ihn ändern. Außerdem können auch `setlocale`-Aufrufe mit einer Kategorie, die der von *item* entspricht (siehe auch `langinfo.h`), oder mit der Kategorie `LC_ALL` diesen Vektor überschreiben.

Wenn in einer Anwendung kein Aufruf von `setlocale()` erfolgt, ist die aktuelle Lokalität im POSIX-Subsystem auf "POSIX" voreingestellt. Die Returnwerte von `nl_langinfo()` richten sich nach der aktuellen Lokalität. Wenn die aktuelle Lokalität für den jeweiligen Parameter keinen Wert enthält, wird der entsprechende Wert der Voreinstellung zurückgegeben.

Siehe auch `setlocale()`, `langinfo.h`, `nl_types.h`, Abschnitt „Lokalität“ auf Seite 52 und Abschnitt „Umgebungsvariablen“ auf Seite 71.

nrand48 - Pseudo-Zufallszahlen zwischen 0 und 2^{31} mit Startwert generieren

Syntax #include <stdlib.h>

long int nrand48 (unsigned short int *xsubi*[3]);

Beschreibung

Siehe `drand48()`.

offsetof - Abstand einer Strukturkomponente zum Strukturbeginn liefern *(BS2000)*

Syntax

```
#include <stddef.h>

size_t offsetof(typ, komponente);
```

Beschreibung

`offsetof()` liefert den Abstand in Byte, den die Strukturkomponente *komponente* vom Beginn der Struktur vom Typ *typ* entfernt ist.
`offsetof()` ist ein Makro.

typ ist der Name des Strukturtyps (Etikett).

komponente ist der Name der Strukturkomponente.

Returnwert

Abstand	der Strukturkomponente vom Strukturbeginn in Byte bei Erfolg.

Hinweis

Ist die angegebene Strukturkomponente ein Bitfeld, ist das Verhalten undefiniert.

open - Datei öffnen

Name **open, open64**

Syntax

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open (const char *path, int oflag , .../* mode_t mode*/);
int open64 (const char *path, int oflag , .../* mode_t mode*/);
```

Beschreibung

Wenn POSIX-Dateien ausgeführt werden, verhält sich die Funktion XPG4-konform wie folgt:

Die Funktion `open()` verbindet eine Datei mit einem Dateideskriptor. Sie erzeugt eine Dateibeschreibung, die auf eine Datei verweist und einen Dateideskriptor, der auf diese Dateibeschreibung verweist. Der Dateideskriptor wird von anderen Ein-/Ausgabefunktionen genutzt, um auf diese Datei zu verweisen. Das Argument *path* zeigt auf einen Pfadnamen, der die Datei bezeichnet.

`open()` liefert einen Dateideskriptor für die genannte Datei, der der kleinste, noch nicht geöffnete Dateideskriptor des Prozesses ist. Die Dateibeschreibung ist neu, daher teilt dieser Dateideskriptor sie nicht mit anderen Prozessen im System. Das Dateideskriptor-Kennzeichen `FD_CLOEXEC`, das mit dem neuen Dateideskriptor verbunden ist, wird gelöscht (siehe `fcntl()`).

Der Lese-/Schreibzeiger wird auf den Dateianfang gesetzt.

Das Dateistatus-Byte und der Zugriffsmodus werden entsprechend dem Wert von *oflag* gesetzt.

Die Werte für *oflag* werden durch bitweise inklusive Oder-Verknüpfung aus den nachfolgenden Kennzeichen erzeugt, die in `fcntl.h` definiert sind. In Anwendungen muss genau eines der ersten drei der unten aufgeführten Kennzeichen (Zugriffsmodi) im Wert von *oflag* angegeben sein:

`O_RDONLY` Nur zum Lesen öffnen.

`O_WRONLY` Nur zum Schreiben öffnen.

`O_RDWR` Zum Lesen und Schreiben öffnen. Das Ergebnis ist nicht definiert, wenn dieses Kennzeichen auf eine FIFO-Datei angewendet wird.

Jede Kombination der folgenden zusätzlichen Kennzeichen kann benutzt werden:

`O_APPEND` Der Lese-/Schreibzeiger wird vor jedem Schreiben auf das Dateiende gesetzt.

O_CREAT Ist die Datei vorhanden, bleibt dieses Kennzeichen wirkungslos, außer die unter O_EXCL angegebenen Bedingungen existieren. Anderenfalls wird die Datei erzeugt und die Benutzernummer der Datei auf die effektive Benutzernummer des Prozesses gesetzt. Die Gruppennummer der Datei wird auf die effektive Gruppennummer des Prozesses oder auf die Gruppennummer des übergeordneten Dateiverzeichnisses der Datei gesetzt. Die Zugriffsrechte der Datei (siehe auch sys/stat.h) werden auf den Wert von *mode* gesetzt und dann folgendermaßen verändert: die einzelnen Bits werden mit dem Komplement der Schutzbitmaske des Prozesses Und-verknüpft (siehe auch umask()); das heißt, alle Bits in den Zugriffsrechten, die in der Schutzbitmaske gesetzt sind, werden gelöscht. Sind andere Bits, als die Schutzbits einer Datei gesetzt, so ist die Wirkung undefiniert. *mode* hat keinen Einfluss darauf, ob die Datei zum Lesen, zum Schreiben oder zum Lesen und Schreiben geöffnet wird.

O_EXCL open() ist erfolglos, wenn O_CREAT und O_EXCL gesetzt sind und die Datei vorhanden ist. Ist die Datei nicht vorhanden, so werden die zwei Aktionen, Prüfung auf Existenz der Datei und Erzeugung der Datei, als eine einzige Aktion behandelt. In diese Aktion kann kein anderer Prozess eingreifen, den open() für denselben Dateinamen und dasselbe Dateiverzeichnis ausführen soll, und der ebenfalls O_EXCL und O_CREAT gesetzt hat. Die Wirkung ist undefiniert, wenn O_CREAT nicht gesetzt ist.

O_NOCTTY Wenn dieses Kennzeichen gesetzt ist und *path* ein Terminal bezeichnet, bewirkt open(), dass dieses Terminal nicht das steuernde Terminal des Prozesses wird.

O_NONBLOCK Wenn eine FIFO zum Lesen oder zum Schreiben geöffnet wird (O_RDONLY oder O_WRONLY):

- O_NONBLOCK ist gesetzt:
  Ein open() zum Lesen kehrt ohne Verzögerung zurück. Ein open() zum Schreiben liefert nur dann einen Fehler, wenn kein Prozess diese Datei zu diesem Zeitpunkt zum Lesen geöffnet hat.
- O_NONBLOCK ist nicht gesetzt:
  Ein open() zum Lesen wartet, bis ein Prozess die Datei zum Schreiben öffnet. Ein open() zum Schreiben wartet, bis ein Prozess die Datei zum Lesen öffnet.

Wenn eine block- oder zeichenorientierte Gerätedatei geöffnet wird, die nichtwartendes Öffnen unterstützt:

- O_NONBLOCK ist gesetzt:
  open() kehrt zurück, ohne darauf zu warten, dass das Gerät fertig oder verfügbar ist. Das nachfolgende Verhalten des Gerätes ist gerätespezifisch.

– O_NONBLOCK ist nicht gesetzt:
Die Funktion open() wartet, bis das Gerät fertig oder verfügbar ist, bevor sie zurückkehrt. Anderenfalls ist das Verhalten von O_NONBLOCK undefiniert.

O_SYNC Wenn O_SYNC für eine normale Datei gesetzt ist, dann verursacht ein Schreibzugriff auf diese Datei, dass der Prozess solange wartet, bis die Daten an die zu Grunde liegende Hardware übergeben wurden.

O_TRUNC Wenn die Datei existiert, eine normale Datei ist und erfolgreich mit O_RDWR oder O_WRONLY geöffnet wurde, dann wird ihre Länge auf 0 gekürzt und Eigentümer und Zugriffsrechte bleiben unverändert. Dies hat keine Wirkung auf FIFO- oder Terminal-Gerätedateien. Die Wirkung auf andere Dateiarten ist nicht definiert, da sie von vielen Faktoren abhängig ist. Das Ergebnis bei einer Verwendung von O_TRUNC zusammen mit O_RDONLY ist undefiniert.

O_LARGEFILE
Falls angegeben, ist das in der internen Beschreibung der offenen Datei festgelegte Offset-Maximum der höchste Wert, der in einem Objekt des Typs off64_t korrekt dargestellt werden kann.

Wenn O_CREAT gesetzt ist und die Datei vorher nicht existierte, dann markiert open() im Erfolgsfall die Felder st_atime, st_ctime und st_mtime der Datei und die Felder st_ctime und st_mtime des übergeordneten Dateiverzeichnisses zum Aktualisieren.

Wenn O_TRUNC gesetzt ist und die Datei vorher bereits existierte, dann markiert open() im Erfolgsfall die Felder st_ctime und st_mtime der Datei zum Aktualisieren.

Es besteht kein funktionaler Unterschied zwischen open() und open64(), außer dass open64() im File Status Flag implizit das Bit O_LARGEFILE setzt. Die Funktion open64() entspricht der Verwendung der Funktion open(), bei der O_LARGEFILE in *oflag* gesetzt ist.

Werden Threads verwendet, so wirkt sich die Funktion auf den Prozess oder auf einen Thread wie folgt aus:

Öffnen einer Datei; Ist beim Parameter *oflag* O_NONBLOCK nicht gesetzt, gilt für FIFO: Ein open() zum Lesen blockiert den aufrufenden Thread, bis ein Thread die Datei zum Schreiben öffnet. Ein open() zum Schreiben blockiert den Thread, bis ein Thread die Datei zum Lesen öffnet. Wird eine block- oder zeichenorientierte Gerätedatei geöffnet wird, die nichtwartendes Öffnen unterstützt, gilt: Die open()-Funktion blockiert den aufrufenden Thread, bis das Gerät fertig oder verfügbar ist.

Erweiterung
Wenn O_CREAT und O_EXCL gesetzt sind, und *path* ein symbolischer Verweis ist, wird der Verweis nicht verfolgt.

BS2000
Wenn BS2000-Dateien ausgeführt werden, ist Folgendes zu beachten:

const char **path* ist eine Zeichenkette, die die zu öffnende Datei angibt. *path* kann sein:

Jeder gültige BS2000-Dateiname.

– `link`=*linkname*
linkname bezeichnet einen BS2000-Linknamen.

– (SYSDTA), (SYSOUT), (SYSLST), die entsprechende Systemdatei.

– (SYSTERM), Terminal-Ein-/Ausgabe

– (INCORE), temporäre Binärdatei, die nur im virtuellen Speicher angelegt wird.

mode ist eine ganzzahlige Variable, deren oktaler Wert die gewünschte Zugriffsart angibt, und zwar:

0000	Öffnen zum Lesen. Die Datei muss bereits vorhanden sein.
0001	Öffnen zum Schreiben. Die Datei muss bereits vorhanden sein. Der alte Inhalt bleibt erhalten.
01001	Öffnen zum Schreiben. Ist die Datei vorhanden, wird der alte Inhalt gelöscht. Ist die Datei nicht vorhanden, wird sie neu erstellt.
0002	Öffnen zum Lesen und Schreiben. Die Datei muss bereits vorhanden sein. Der alte Inhalt bleibt erhalten.
01002	Öffnen zum Lesen und Schreiben. Ist die Datei vorhanden, wird der alte Inhalt gelöscht. Ist die Datei nicht vorhanden, wird sie neu erstellt.
0003	Öffnen zum Neuschreiben und Lesen. Ist die Datei vorhanden, wird der alte Inhalt gelöscht. Ist die Datei nicht vorhanden, wird sie neu erstellt.
0401	Öffnen zum Anfügen ans Ende der Datei. Die Datei muss bereits vorhanden sein. Es wird auf das Dateiende positioniert, d.h. der alte Inhalt bleibt erhalten und der neue Text wird ans Ende der Datei angehängt.
0402	Öffnen zum Anfügen ans Ende der Datei und zum Lesen. Die Datei muss bereits vorhanden sein. Der alte Inhalt bleibt erhalten und der neue Text wird ans Ende der Datei angehängt. Nach dem Öffnen ist die Datei bei KR-Funktionalität (nur bei C/C++ Versionen kleiner V3 vorhanden) auf das Dateiende positioniert, bei ANSI-Funktionalität auf den Dateianfang. ❑

Returnwert

nichtnegative ganze Zahl, die die kleinste, nicht benutzte Dateideskriptor-Zahl darstellt,	bei Erfolg.
-1	bei Fehler. Es werden keine Dateien erzeugt oder aktualisiert. `errno` wird gesetzt, um denFehler anzuzeigen.

Fehler `open()` und `open64()` schlagen fehl, wenn gilt:

`EACCES` Für eine Komponente des Pfades existiert kein Durchsuchrecht.

Die Datei existiert nicht, und die durch *oflag* angegebenen Zugriffsrechte werden nicht erteilt.

Die Datei existiert nicht, und das übergeordnete Verzeichnis der zu erstellenden Datei hat kein Schreibrecht.

`O_TRUNC` ist gesetzt, und es gibt kein Schreibrecht für die Datei.

Erweiterung

`EAGAIN` Die Datei ist vorhanden, obligatorisches Sperren von Dateien und Dateisätzen ist gesetzt, und Datensatzsperren sind noch in der Datei vorhanden (siehe `chmod()`).

`EEXIST` `O_CREAT` und `O_EXCL` sind gesetzt, und die angegebene Datei ist bereits vorhanden.

`EFAULT` *path* weist über den zugewiesenen Adressraum des Prozesses hinaus.

`EINTR` Während des Systemaufrufs `open()` wurde ein Signal abgefangen.

`EINVAL` Der Wert des Arguments *oflag* ist ungültig.

`EIO` Während des Öffnens eines stream-orientierten Gerätes ist ein Verbindungsabbau oder ein Fehler aufgetreten.

`EISDIR` Die angegebene Datei ist ein Dateiverzeichnis und *oflag* enthält `O_WRONLY` oder `O_RDWR`.

`ELOOP` Bei der Auflösung von *path* wurden zuviele symbolische Links angetroffen.

`EMFILE` `{OPEN_MAX}`-Dateideskriptoren sind bereits für den aufrufenden Prozess geöffnet.

`EMULTIHOP` Komponenten von *path* erfordern den Sprung auf mehrere ferne Rechner, aber der Dateisystemtyp erlaubt dies nicht.

`ENAMETOOLONG`
Die Länge von *path* überschreitet `{PATH_MAX}`, oder eine Komponente des Pfades ist länger als `{NAME_MAX}`.

`ENFILE` Die maximal erlaubte Anzahl von Dateien im System ist bereits geöffnet.

`ENOENT` `O_CREAT` ist nicht gesetzt, und die angegebene Datei ist nicht vorhanden.

`O_CREAT` ist gesetzt, und entweder existiert der Pfadnamen-Anfang nicht oder *path* zeigt auf eine leere Zeichenkette.

`ENOLINK` *path* weist auf einen fernen Rechner, zu dem keine aktive Verbindung existiert.

ENOSPC	Die Datei existiert nicht, und O_CREAT ist gesetzt, oder das Dateiverzeichnis oder Dateisystem, in dem eine neue Datei erstellt werden soll, kann nicht erweitert werden.
ENOSR	Ein Datenstrom kann nicht zugewiesen werden.
ENOTDIR	Eine Komponente des Pfades ist kein Dateiverzeichnis.
ENXIO	Die angegebene Datei ist eine Gerätedatei für ein zeichen- oder blockorientiertes Gerät und das dieser Gerätedatei zugewiesene Gerät existiert nicht. O_NONBLOCK ist gesetzt, die angegebene Datei ist eine FIFO, O_WRONLY ist gesetzt, und kein Prozess hat die Datei zum Lesen geöffnet.
EROFS	Die angegebene Datei befindet sich auf einem Dateisystem, das nur Leserecht hat, und entweder O_WRONLY, O_RDWR, O_CREAT (wenn die Datei nicht existiert) oder O_TRUNC ist im Argument *oflag* gesetzt.
EOVERFLOW	Für eine Datei ist O_LARGEFILE nicht gesetzt und die Größe der Datei kann in einem Objekt des Typs off_t nicht korrekt dargestellt werden.

Hinweis

Ob open() für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

BS2000

Der BS2000-Dateiname bzw. -Linkname kann in Klein- und Großbuchstaben geschrieben werden, er wird automatisch in Großbuchstaben umgesetzt.

Wird eine nicht vorhandene Datei neu angelegt, so wird standardmäßig eine Datei mit folgenden Attributen erzeugt:
Bei KR-Funktionalität (nur bei C/C++ Versionen kleiner V3 vorhanden) eine SAM-Datei mit variabler Satzlänge und Standardblocklänge,
bei ANSI-Funktionalität eine ISAM-Datei mit variabler Satzlänge und Standardblocklänge.
SAM-Dateien sind beim Öffnen mit open() immer Textdateien.

Bei Verwendung eines Linknamens lassen sich mit dem ADD-FILE-LINK-Kommando folgende Dateiattribute ändern: Zugriffsmethode, Satzlänge, Satzformat, Blocklänge und Blockformat.
In allen Fällen, in denen der alte Inhalt einer bereits existierenden Datei gelöscht wird (0003, 01001), bleiben die Katalogeigenschaften dieser Datei erhalten.

Position des Lese-/Schreibzeigers im Anfügemodus:

Wenn der Lese-/Schreibzeiger in einer Datei, die im Anfügemodus eröffnet wurde (0401, 0402), explizit vom Dateiende wegpositioniert wurde (lseek()), wird er je nach KR- oder ANSI-Funktionalität unterschiedlich behandelt.
KR-Funktionalität (nur bei C/C++ Versionen kleiner V3 vorhanden): Der aktuelle Lese-/Schreibzeiger wird nur beim Schreiben mit der Elementarfunktion write() ignoriert und

automatisch ans Ende der Datei positioniert.
ANSI-Funktionalität: Der aktuelle Lese-/Schreibzeiger wird bei allen Schreibfunktionen ignoriert und automatisch ans Ende der Datei positioniert.

Der Versuch, eine nicht existierende Datei zum Lesen (0000, 0002), zum Ändern (0001) sowie zum Anfügen (0401, 0402) zu öffnen, endet mit Fehler.
Eine Datei kann gleichzeitig für verschiedene Zugriffsmodi eröffnet werden, sofern diese Modi im BS2000-Datenverwaltungssystem miteinander verträglich sind.
(INCORE)-Dateien können nur zum Neuschreiben (01001) oder zum Neuschreiben und Lesen (0003) geöffnet werden. Es müssen zuerst Daten geschrieben werden. Um die geschriebenen Daten wieder einlesen zu können, muss die Datei mit der Funktion `lseek()` auf den Dateianfang positioniert werden.

Wenn ein Programm startet, werden die Standarddateien für Eingabe, Ausgabe und Fehlerausgabe automatisch mit folgenden Dateideskriptoren geöffnet:

`stdin:` 0

`stdout:` 1

`stderr:` 2

Es können maximal _NFILE-Dateien gleichzeitig geöffnet sein. _NFILE ist in `stdio.h` mit 2048 definiert.

Siehe auch `chmod()`, `close()`, `creat()`, `creat64()`, `dup()`, `fcntl()`, `fdopen()`, `lseek()`, `lseek64()`, `read()`, `umask()`, `write()`, `fcntl.h`, `sys/types.h`, `sys/stat.h`.

opendir - Dateiverzeichnis öffnen

Syntax #include <dirent.h>

Optional
#include <sys/types.h> ❑

DIR *opendir(const char **dirname*);

Beschreibung

`opendir()` öffnet einen Dateiverzeichnisstrom, entsprechend dem durch *dirname* angegebenen Dateiverzeichnis. Der Dateiverzeichnisstrom wird auf den ersten Eintrag positioniert. Der Datentyp `DIR`, der in `dirent.h` definiert ist, repräsentiert einen Dateiverzeichnisstrom, der eine geordnete Folge aller Dateiverzeichnis-Einträge in einem speziellen Dateiverzeichnis ist. Der Datentyp `DIR` ist unter POSIX durch einen Dateideskiptor implementiert. Daher können Anwendungen höchstens `{OPEN_MAX}`-Dateien und Dateiverzeichnisse öffnen.

Der Nullzeiger wird zurückgegeben, wenn auf *dirname* nicht zugegriffen werden kann, wenn *dirname* kein Dateiverzeichnis ist oder wenn nicht genügend Speicherplatz zur Aufnahme einer `DIR`-Struktur bzw. eines Puffers für die Dateiverzeichniseinträge mit `malloc()` zur Verfügung gestellt werden kann.

Returnwert Zeiger auf ein `DIR`-Objekt
bei Erfolg.

Nullzeiger bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `opendir()` schlägt fehl, wenn gilt:

`EACCES` Das Durchsuchrecht für eine Komponente von *dirname* oder das Leserecht für *dirname* wird nicht erteilt.

Erweiterung
`EFAULT` *dirname* weist über den zugewiesenen Adressraum hinaus.

`ELOOP` Während der Übersetzung von *dirname* waren zu viele symbolische Verweise vorhanden. ❑

`EMFILE` Für den Prozess sind derzeit mehr als `{OPEN_MAX}`-Dateideskriptoren offen.

`ENAMETOOLONG`
Die Länge von *dirname* überschreitet `{PATH_MAX}`, oder eine Pfadnamen-Komponente ist länger als `{NAME_MAX}`.

`ENFILE` Im System sind derzeit zuviele Dateideskriptoren offen.

`ENOENT` *dirname* zeigt auf den Namen einer Datei, die nicht existiert, oder auf die leere Zeichenkette.

`ENOTDIR` Eine Komponente von *dirname* ist kein Dateiverzeichnis.

Hinweis `opendir()` sollte in Verbindung mit `readdir()`, `closedir()` und `rewinddir()` verwendet werden, um den Inhalt eines Dateiverzeichnisses zu untersuchen (siehe auch `readdir()`). Diese Methode wird aus Portabilitätsgründen empfohlen.

`opendir()` wird nur für POSIX-Dateien ausgeführt.

Siehe auch `closedir()`, `readdir()`, `rewinddir()`, `dirent.h`, `sys/types.h`, `limits.h`.

openlog - System Logging

Syntax #include <syslog.h>

void openlog(const char **ident*, int *logopt*, int *facility*);

Beschreibung

siehe `closelog()`.

optarg, opterr, optind, optopt - Variablen für Kommandooptionen

Syntax #include <unistd.h>

extern char *optarg;
extern int optind, opterr, optopt;

Beschreibung

Siehe `getopt()`.

pathconf, fpathconf - Wert einer Pfadnamen-Variablen ermitteln

Syntax #include <unistd.h>

long int pathconf(const char **path*, int *name*);
long int fpathconf(int *fildes*, int *name*);

Beschreibung

Mit `pathconf()` und `fpathconf()` kann der aktuelle Wert einer konfigurierbaren Systemvariablen *name* ermittelt werden, die einer Datei oder einem Dateiverzeichnis zugeordnet ist.

Für `pathconf()` zeigt *path* auf den Pfadnamen einer Datei oder eines Dateiverzeichnisses.

Für `fpathconf()` ist *fildes* der Deskriptor einer offenen Datei.

Das C-Laufzeitsystem unterstützt die Variablen, die in der folgenden Tabelle aufgeführt sind. Andere X/Open-kompatible Implementierungen können weitere Variablen unterstützen. Die folgende Tabelle enthält die Systemvariablen aus den Dateien `limits.h` oder `unistd.h`, die mit `pathconf()` oder `fpathconf()` abgefragt werden können; die symbolischen Konstanten sind in `unistd.h` definiert. Sie enthalten die entsprechenden Werte für das Argument *name*:

Systemvariable *name*	**Wert von *name* (Konstante)**	**Bemerkungen**
{LINK_MAX}	_PC_LINK_MAX	1.
{MAX_CANON}	_PC_MAX_CANON	2.
{MAX_INPUT}	_PC_MAX_INPUT	2.
{NAME_MAX}	_PC_NAME_MAX	3., 4.
{PATH_MAX}	_PC_PATH_MAX	4., 5.
{PIPE_BUF}	_PC_PIPE_BUF	6.
_POSIX_CHOWN_RESTRICTED	_PC_CHOWN_RESTRICTED	7.
_POSIX_NO_TRUNC	_PC_NO_TRUNC	3., 4.
_POSIX_VDISABLE	_PC_VDISABLE	2.

1. Wenn *path* oder *fildes* auf ein Dateiverzeichnis verweisen, bezieht sich der Returnwert auf das Dateiverzeichnis selbst.
2. Wenn *path* oder *fildes* nicht auf eine Gerätedatei für ein Terminal verweisen, werden die Variablen {MAX_CANON}, {MAX_INPUT} und _POSIX_VDISABLE ignoriert.
3. Wenn *path* oder *fildes* auf ein Dateiverzeichnis verweisen, bezieht sich der Returnwert auf Dateinamen im Dateiverzeichnis.

4. Wenn *path* oder *fildes* nicht auf ein Dateiverzeichnis verweisen, wird keine Verbindung der Variablen {NAME_MAX}, {PATH_MAX} und _POSIX_VDISABLE mit der spezifizierten Datei unterstützt.
5. Wenn *path* oder *fildes* auf ein Dateiverzeichnis verweisen, ist der Returnwert die maximale Länge eines relativen Pfadnamens, wenn das angegebene Dateiverzeichnis das aktuelle Dateiverzeichnis ist.
6. Wenn *path* auf eine FIFO oder *fildes* auf eine Pipe oder FIFO verweist, bezieht sich der Returnwert auf das referenzierte Objekt. Wenn *path* oder *fildes* auf ein Dateiverzeichnis verweisen, bezieht sich der Returnwert auf eine existierende oder innerhalb des Dateiverzeichnisses erzeugte FIFO. Wenn *path* oder *fildes* auf einen anderen Dateityp verweisen, ist das Verhalten undefiniert.
7. Wenn *path* oder *fildes* auf ein Dateiverzeichnis verweisen, bezieht sich der Returnwert auf irgendwelche, in diesem Standard definierte Dateien, die keine Dateiverzeichnisse sind und innerhalb des Dateiverzeichnisses existieren oder erzeugt werden können.

Returnwert aktueller Wert von *name*
bei Erfolg.
Der Returnwert ist nicht niedriger als der entsprechende Wert in der Anwendung, wenn diese mit limits.h oder unistd.h der jeweiligen Implementierung übersetzt wird.

-1 wenn die Variable zu *name* keine Grenze für *path* oder den Dateideskriptor hat. errno wird nicht gesetzt.

-1 wenn *name* einen ungültigen Wert hat,
oder wenn die Implementierung *path* oder *fildes* benutzen muss, um den Wert von *name* zu bestimmen und die Implementierung die Zuordnung von *name* zu der durch *path* oder *fildes* angegebenen Datei nicht unterstützt,
oder wenn der Prozess nicht die entsprechenden Rechte besitzt, um die durch *path* oder *fildes* angegebene Datei zu überprüfen,
oder wenn *path* nicht existiert,
oder wenn *fildes* kein gültiger Dateideskriptor ist.

In diesen Fällen wird errno gesetzt, um den Fehler anzuzeigen.

Fehler pathconf() schlägt fehl, wenn gilt:

Erweiterung

EACCES Für eine Komponente des Pfadnamens besteht kein Suchrecht. ❑

EINVAL Der Wert von *name* ist ungültig,
oder es wird versucht, auf eine BS2000-Datei zuzugreifen.

Erweiterung

ELOOP Es gibt zu viele symbolische Verweise beim Übersetzen von *path*.

ENAMETOOLONG
Die Länge der Zeichenkette *path* überschreitet den Wert {PATH_MAX}, oder eine Komponente des Pfadnamens ist länger als {NAME_MAX}, während _POSIX_NO_TRUNC aktiv ist.

ENOENT Die angegebene Datei existiert nicht, oder *path* zeigt auf eine leere Zeichenkette. ❑

ENOTDIR Eine Komponente des Pfadnamen-Anfangs ist kein Dateiverzeichnis.

fpathconf() schlägt fehl, wenn gilt:

EINVAL Der Wert von *name* ist ungültig, oder die Implementierung unterstützt die Zuordnung von *name* zur angegebenen Datei nicht.

EBADF *fildes* ist kein gültiger Dateideskriptor.

Siehe auch sysconf(), limits.h, unistd.h.

pause - Prozess bis zum Empfang eines Signals anhalten

Syntax

```
#include <unistd.h>

int pause(void);
```

Beschreibung

`pause()` hält den aufrufenden Prozess an, bis ein Signal zugestellt wird, dessen Signalaktion entweder die Ausführung einer Signalbehandlungsfunktion oder die Prozessbeendigung ist.

Wenn die Signalaktion die Prozessbeendigung ist, kehrt die Funktion `pause()` nicht zurück.

Wenn die Signalaktion die Ausführung einer Signalbehandlungsfunktion ist, kehrt die Funktion `pause()` zurück, nachdem die Signalbehandlungsfunktion zurückgekehrt ist.

Werden Threads verwendet, so wirkt sich die Funktion auf den Prozess oder auf einen Thread wie folgt aus: Suspendiert den Thread, bis er ein Signal erhält.

Returnwert

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

Da die Funktion `pause()` die Prozessausführung solange unterbricht, bis sie von einem Signal unterbrochen wird, kann `pause()` keinen Returnwert für erfolgreiche Beendigung haben.

Fehler

`pause()` schlägt fehl, wenn gilt:

`EINTR` Ein Signal wurde vom aufrufenden Prozess abgefangen und die Steuerung wurde von der Signalbehandlungsfunktion zurückgegeben.

Siehe auch `sigsuspend()`, `sleep()`, `unistd.h`.

pclose - Pipe-Strom schließen

Syntax #include <stdio.h>

int pclose(FILE **stream*);

Beschreibung

`pclose()` schließt den Datenstrom *stream*, der durch `popen()` geöffnet wurde, wartet auf die Beendigung des durch `popen()` gestarteten Kommandos und gibt dessen Endestatus zurück. Wenn jedoch der Endestatus für `pclose()` nicht verfügbar ist, wird -1 zurückgegeben und `errno` wird auf `ECHILD` gesetzt, um die Situation zu dokumentieren. Das kann eintreten, wenn die Anwendung den Endestatus bereits durch eine der folgenden Funktionen gelesen hat:

– `wait()`

– `waitpid()` mit einem pid-Argument, das kleiner oder gleich 0 oder gleich der Prozessnummer des Kommandointerpreters ist.

In jedem Fall kehrt `pclose()` nicht zurück, bevor der durch `popen()` erzeugte Sohnprozess beendet wurde.

Wenn der Kommandointerpreter nicht ausgeführt werden kann, liefert `pclose()` einen Endestatus, der dem entspricht, als ob der Kommandointerpreter durch `exit(127)` oder `_exit(127)` beendet worden wäre.

Returnwert Endestatus des Kommandointerpreters bei Erfolg.

-1 wenn *stream* nicht durch `popen()` erzeugt wurde.

Fehler `pclose()` schlägt fehl, wenn gilt:

`ECHILD` Der Endestatus des Sohnprozesses konnte nicht ermittelt werden.

Erweiterung
`EINVAL` Es wurde versucht, auf eine BS2000-Datei zuzugreifen. ❑

Hinweis `pclose()` wird nur für POSIX-Dateien ausgeführt.

Siehe auch `fork()`, `popen()`, `wait()`, `waitpid()`, `stdio.h`.

perror - Meldung auf Standard-Fehlerausgabe ausgeben

Syntax #include <stdio.h>

void perror(const char *s);

Beschreibung

`perror()` bildet die Fehlernummer in der externen Variablen `errno` auf eine sprachabhängige Fehlermeldung *error_message* ab, die wie folgt in den Standard-Fehlerausgabestrom geschrieben wird:
s : *error_message* `\n`

s ist eine Zeichenkette, die zumindest den Namen des Programms enthalten sollte, in dem der Fehler auftrat. Wenn *s* der Nullzeiger oder das Zeichen, auf das *s* zeigt, das Nullbyte ist, fehlt der Meldungsteil `"`*s* `: "`.

Die Inhalte der Meldungen richten sich nach der Umgebungsvariablen `LANG`. Fehlernummern und Fehlermeldungen sind in `errno.h` vollständig aufgeführt und erläutert.

`perror()` kennzeichnet die Datei, die mit dem Standard-Fehlerausgabestrom verbunden ist, als beschrieben (`st_ctime` und `st_mtime` werden zum Ändern markiert). Dies geschieht zwischen der erfolgreichen Ausführung von `perror()` und einem Aufruf von `exit()`, `abort()` oder einem Zugriff von `fflush()` oder `fclose()` auf `stderr`.

Hinweis Der Inhalt des Bereichs, in dem die Fehlernummer und der Fehlertext abgespeichert sind, wird nicht explizit gelöscht, d.h. der alte Inhalt bleibt solange erhalten, bis er bei neuerlichem Auftritt eines Fehlers mit den entsprechenden Informationen überschrieben wird. `perror`-Aufrufe sind daher nur sinnvoll, nachdem eine Funktion einen Fehler-Returnwert geliefert hat.
Ob `perror()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

BS2000
error_message enthält bei Ein-/Ausgabefehlern oder bei Ausführung von Systemkommandos als zusätzliche Information die entsprechende DVS-Fehlernummer.
Im KR-Modus (nur bei C/C++ Versionen kleiner V3 vorhanden) wird ein Returnwert vom Typ `char *` geliefert. Er enthält einen Zeiger auf einen C-internen Speicherbereich, in dem die Fehlermeldung steht. Der Inhalt wird bei jedem `perror`-Aufruf überschrieben (siehe auch Handbuch „C-Bibliotheksfunktionen V2.3A (BS2000)“).
Wenn das Programm in einer BS2000-Umgebung gestartet wird und die Datei nicht vorhanden ist, erhält man folgende Fehlermeldung auf die Standard-Ausgabe:
`Programm fopen: dataset not found (cmd: OPEN), errorcode=DD33`

DD33 ist dabei die DVS-Fehlernummer. ❑

Siehe auch `strerror()`, `errno.h`, `stdio.h`, Abschnitt „Wahl der Funktionalität“ auf Seite 40.

pipe - Pipe erzeugen

Syntax #include <unistd.h>

int pipe(int *fildes*[2]);

Beschreibung

`pipe()` erzeugt eine Pipe und trägt zwei Dateideskriptoren, die auf die offenen Dateibeschreibungen für die Lese- bzw. Schreibseite der Datei verweisen, in die Argumente *fildes*`[0]` und *fildes*`[1]` ein. Diese beiden ganzzahligen Werte sind die beiden zum Zeitpunkt des Aufrufs von `pipe()` niedrigsten verfügbaren. Das Bit `O_NONBLOCK` ist für keine der beiden Dateideskriptoren gesetzt (`fcntl()` kann verwendet werden, um das Bit `O_NONBLOCK` zu setzen).

Daten können dann über den Dateideskriptor *fildes*`[1]` geschrieben und über den Dateideskriptor *fildes*`[0]` gelesen werden. Ein Lesevorgang über *fildes*`[0]` greift auf die Daten zu, die über *fildes*`[1]` geschrieben wurden, und zwar nach der Methode first-in-first-out.

Ein Prozess hat die Pipe zum Lesen geöffnet, wenn er den Dateideskriptor besitzt, der auf die Leseseite der Pipe verweist, d.h. *fildes*`[0]` (entsprechend zum Schreiben bei der Schreibseite, d.h. *fildes*`[1]`).

Bei erfolgreicher Beendigung aktualisiert `pipe()` die `stat`-Strukturkomponenten `st_atime`, `st_ctime` und `st_mtime` der Pipe.

Das Bit `FD_CLOEXEC` ist für keine der beiden Dateideskriptoren gesetzt.

Returnwert

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

Fehler `pipe()` schlägt fehl, wenn gilt:

`EMFILE`	Für den Prozess sind derzeit bereits `{OPEN_MAX}` minus 2 Dateideskriptoren offen.
`ENFILE`	Die Anzahl der gleichzeitig geöffneten Dateien im System würde eine systemabhängige Grenze überschreiten.

Hinweis `pipe()` wird nur für POSIX-Dateien ausgeführt.

Siehe auch `fcntl()`, `read()`, `write()`, `unistd.h`.

poll - STREAMs-Ein-/Ausgabe multiplexen

Syntax #include <poll.h>

int poll(struct pollfd *fds*[], nfds_t *nfds*, int *timeout*);

Beschreibung

`poll()` stellt Anwendungen einen Mechanismus für das Multiplexen von Ein-/Ausgaben über einen Satz von offenen Dateideskriptoren zur Verfügung.
Für jedes Feldelement, auf das *fds* zeigt, überprüft `poll()`, ob für den entsprechenden Dateideskriptor eines oder mehrere der in *events* aufgelisteten Ereignisse eingetreten ist. Die Anzahl `pollfd`-Strukturen im Feld *fds* wird durch den Wert *nfds* angegeben. `poll()` identifiziert die Dateideskriptoren, auf denen die Anwendung lesen oder schreiben kann oder für die Ereignisse eingetroffen sind.

fds legt die zu prüfenden Dateideskriptoren fest sowie die Ereignisse, die für den jeweiligen Dateideskriptor abgefragt werden sollen. *fds* ist ein Zeiger auf ein Feld mit jeweils einem Element für jeden zu prüfenden Dateideskriptor. Die Elemente des Feldes sind `pollfd`-Strukturen, die Folgendes enthalten:

```
int fd;           /* offener Dateideskriptor */
short events;     /* abzufragende Ereignisse */
short revents;    /* eingetretene Ereignisse */
```

fd bezeichnet einen offenen Dateideskriptor, `events` und `revents` sind Bitmasken, die durch ODER-Verknüpfung aus den folgenden Flags aufgebaut werden (es sind beliebige Kombinationen zulässig):

`POLLIN` Daten, die nicht die höchste Priorität haben, können nichtblockierend gelesen werden. Für STREAMS wird dieses Flag in `revents` auch dann gesetzt, wenn die Nachricht die Länge 0 hat.

`POLLRDNORM` Normale Daten (Priorität = 0) können nichtblockierend gelesen werden. Für STREAMS wird dieses Flag in `revents` auch dann gesetzt, wenn die Nachricht die Länge 0 hat.

`POLLRDBAND` Daten mit Priorität ≠ 0 können nichtblockierend gelesen werden. Für STREAMS wird dieses Flag in `revents` auch dann gesetzt, wenn die Nachricht die Länge 0 hat.

`POLLPRI` Daten mit höchster Priorität können nichtblockierend empfangen werden. Für STREAMS wird dieses Flag in `revents` auch dann gesetzt, wenn die Nachricht die Länge 0 hat.

`POLLOUT` Normale Daten (Priorität = 0) können nichtblockierend geschrieben werden.

`POLLWRNORM` wie `POLLOUT`.

`POLLWRBAND` Daten mit Priorität ≠ 0 können geschrieben werden.

POLLMSG Eine M_SIG- oder M_PCSIG-Nachricht, die ein ASIGPOLL-Signal enthält, hat den Anfang der Stream-Kopf-Warteschlange erreicht.

POLLERR Es ist ein Fehler aufgetreten für den STREAM oder die Gerätedatei. Dieses Flag ist nur in der revents-Bitmaske gültig; in der Bitmaske events wird es ignoriert.

POLLHUP Im STREAM ist ein Hangup aufgetreten (die Verbindung zum Gerät ist unterbrochen). POLLHUP und POLLOUT schließen sich gegenseitig aus; auf einen Stream kann niemals geschrieben werden, wenn ein Hangup aufgetreten ist. Jedoch schließen sich dieses Ereignis und POLLIN bzw. POLLRDNORM, POLLRDBAND oder POLLPRI nicht gegenseitig aus.
Das Flag POLLHUP ist nur in der revents-Bitmaske gültig; in der Bitmaske events wird es ignoriert.

POLLNVAL Der angegebene *fd*-Wert ist ungültig. Dieses Flag ist nur in der revents-Bitmaske gültig; in der Bitmaske events wird es ignoriert.

Wenn der Wert in *fd* kleiner als null ist, wird events ignoriert, und revents wird bei der Rückkehr von poll() für diesen Feldeintrag auf 0 gesetzt.

Die Ergebnisse der poll()-Anfrage werden im revents-Feld in der pollfd-Struktur angezeigt. poll() setzt zunächst alle Bits in revents auf null. Falls eines oder mehrere der in events abgefragten Ereignisse eingetroffen ist, setzt poll() die entsprechenden Bits in revents . Die Bits für POLLHUP, POLLERR und POLLNVAL werden beim Eintreffen der entsprechenden Ereignisse automatisch in revents gesetzt; sie müssen in events nicht gesetzt sein.

Wenn die Überprüfung ergibt, dass keines der für die Dateideskriptoren abgefragten Ereignisse eingetreten ist, wartet poll() wenigstens *timeout* Millisekunden auf das Auftreten eines Ereignisses für einen der angegebenen Dateideskriptoren. Bei einem Rechner, bei dem die Genauigkeit auf Millisekunden nicht zur Verfügung steht, wird *timeout* auf den nächsten zulässigen Wert aufgerundet, der in diesem System zur Verfügung steht.
Wenn der Wert von *timeout* 0 ist, kehrt poll() sofort zurück. Hat *timeout* den Wert -1, wartet poll(), bis eines der abgefragten Ereignisse auftritt, oder bis der Aufruf unterbrochen wird (blockierender Aufruf von poll()).

poll() wird von den Flags O_NDELAY und O_NONBLOCK nicht beeinflusst.

poll() unterstützt Textdateien, Terminals, Pseudo-Terminals, STREAMS-basierte Dateien, FIFO-Dateien und Pipes, Sockets und XTI.

Bei Textdateien liefert poll() immer ein TRUE für das Lesen und Schreiben.

Returnwert

Wert ≥ 0	bei Erfolg. Ein positiver Wert zeigt die Gesamtanzahl der Dateideskriptoren an, für die das Feld `revents` ungleich null ist. 0 bedeutet, dass die Zeit für den Aufruf abgelaufen ist und keine Dateideskriptoren vorhanden sind, für die das Feld `revents` ungleich null ist.
-1	bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `poll()` schlägt fehl, wenn gilt:

`EAGAIN`	Die Zuweisung der internen Datenstrukturen ist fehlgeschlagen, könnte aber bei einer Wiederholung gelingen.
`EFAULT`	Ein Argument zeigt auf einen Speicherplatz außerhalb des zugewiesenen Adressraums.
`EINTR`	Während des Systemaufrufs `poll()` wurde ein Signal abgefangen.
`EINVAL`	Das Argument *nfds* ist kleiner als null oder größer als `OPEN_MAX` oder einer der *fd*-Einträge bezieht sich auf einen STREAM oder Multiplexer, der streamabwärts über einen Multiplexer angeschlossen ist.

Siehe auch `getmsg()`, `putmsg()`, `read()`, `select()`, `write()`, `poll.h`, `stropts.h`.

popen - Pipe-Strom von oder zu einem Prozess öffnen

Syntax #include <stdio.h>

FILE *popen (const char **command*, const char **mode*);

Beschreibung

`popen()` führt das Kommando aus, das durch die Zeichenkette *command* angegeben ist, und erzeugt eine Pipe zwischen dem aufrufenden Programm und dem auszuführenden Kommando. `popen()` gibt als Returnwert einen Dateizeiger zurück, der entweder zum Lesen (Ein-/Ausgabe-Modus `r`) von der Pipe oder zum Schreiben (Ein-/Ausgabe-Modus `w`) auf die Pipe eingesetzt werden kann.

Die Umgebung des ausgeführten Kommandos in einer XPG4-konformen Implementierung ist so, als ob der Sohnprozess innerhalb von `popen()` mit `fork()` erzeugt worden wäre und der Sohn das `sh`-Kommando wie folgt aufruft:

`execl (`*shell_path*`, "sh", "-c",` *command*`, (char *)0);`

shell_path ist ein nicht spezifizierter Name für das `sh`-Kommando.

`popen()` stellt sicher, dass Datenströme von vorherigen `popen`-Aufrufen, die in den Vaterprozessen offen bleiben, im neuen Sohnprozess geschlossen werden.

mode ist eine Zeichenkette, die den Ein-/Ausgabe-Modus festlegt:

1. Wenn bei Start des Sohnprozesses *mode* `r` ist, wird die Standardausgabe des Kommandos auf die Pipe umgelenkt. Der Dateideskriptor `STDOUT_FILENO` ist dann das beschreibbare Ende, der Dateideskriptor *fileno*(*stream*) das lesbare Ende der Pipe. *stream* ist der von `popen()` zurückgegebene Stromzeiger.
2. Wenn bei Start des Sohnprozesses *mode* `w` ist, wird die Standardeingabe des Kommandos auf die Pipe umgelenkt. Der Dateideskriptor `STDIN_FILENO` ist dann das lesbare Ende, der Dateideskriptor *fileno*(*stream*) das beschreibbare Ende der Pipe. *stream* ist der von `popen()` zurückgegebene Stromzeiger.

Nach `popen()` sind sowohl Vater- als auch Sohnprozess unabhängig voneinander ablauffähig, bevor sie sich beenden.

Returnwert Zeiger auf einen Datenstrom
bei Erfolg.

Nullzeiger wenn Dateien oder Prozesse nicht erzeugt werden können.

Hinweis Wenn der Vaterprozess und der durch `popen()` erzeugte Prozess gleichzeitig eine Datei lesen oder beschreiben, darf keiner der Prozesse gepufferte Ein-/Ausgabe verwenden. Probleme mit einem Ausgabefilter können durch sorgfältiges Entleeren des Puffers, z.B. mit `fflush()` vermieden werden (siehe auch `fclose()`).

popen() wird nur für POSIX-Dateien ausgeführt.

Siehe auch pclose(), pipe(), sysconf(), system(), stdio.h, Kommando sh im Handbuch „POSIX Kommandos (BS2000/OSD)“.

pow - Potenzfunktion anwenden

Syntax

#include <math.h>

double pow(double *x*, double *y*);

Beschreibung

pow() berechnet x^y .

x ist eine Gleitkommazahl, die Basis der Exponentialfunktion.

y ist auch eine Gleitkommazahl, der Exponent.

Falls *x* gleich 0 ist, muss *y* positiv sein,
falls *x* negativ ist, muss *y* ganzzahlig sein.

Returnwert

Wert von x^y	falls *x*, *y* und das Ergebnis im zulässigen Gleitkommaintervall liegen.
+/−HUGE_VAL	(je nach Vorzeichen), bei Überlauf. errno wird gesetzt, um den Fehler anzuzeigen.
1.0	wenn *x* und *y* gleich 0 sind.
−HUGE_VAL	wenn *x* gleich 0 und *y* kleiner 0 ist. errno wird gesetzt, um den Fehler anzuzeigen.
undefiniert	wenn *x* kleiner 0 und *y* nicht ganzzahlig ist. errno wird gesetzt, um den Fehler anzuzeigen.

Fehler

pow() schlägt fehl, wenn gilt:

EDOM	Der Wert von *x* ist negativ und *y* ist nicht ganzzahlig.
	Der Wert von *x* ist 0 und *y* ist negativ.
ERANGE	Der Wert von *x* verursacht einen Überlauf.

Siehe auch exp(), hypot(), log(), log10(), sinh(), sqrt(), math.h.

printf - formatiert in Standard-Ausgabestrom schreiben

Syntax #include <stdio.h>

int printf(const char **format*, *arglist*);

Beschreibung

Siehe `fprintf()`.

ptsname - Name eines Pseudoterminals

Syntax #include <stdlib.h>

char **ptsname*(int *fildes*);

Beschreibung

Die Funktion `ptsname()` liefert den Namen des Slave-Pseudo-Terminals, das dem Master-Pseudo-Terminal zugeordnet ist. *fildes* ist der Dateideskriptor, der sich auf das Master-Terminal bezieht. `ptsname()` liefert einen Zeiger auf eine Zeichenkette, die den Pfadnamen des zugehörigen Slave-Terminals enthält. Der Name wird mit dem Nullbyte abgeschlossen.

Der Name hat die Form `/dev/pts/N` , wobei N eine ganze Zahl zwischen 0 und 255 ist.

`ptsname()` ist nicht threadsicher.

Returnwert Zeiger auf eine Zeichenkette
bei Erfolg. Die Zeichenkette enthält den Namen des Slave-Terminals.

Nullzeiger bei Fehler. Dies kann passieren, wenn *fildes* kein gültiger Dateideskriptor ist oder wenn der Name des Slave-Terminals im Dateisystem nicht existiert.

Hinweis Der Zeiger zeigt auf einen statischen Datenbereich, der bei jedem Aufruf von `ptsname()` überschrieben wird.

Siehe auch `grantpt()`, `ttyname()`, `unlockpt()`, `stdlib.h`.

putc, putc_unlocked - Byte in Datenstrom schreiben

Syntax

#include <stdio.h>

int putc(int *c*, FILE **stream*);

int putc_unlocked(int *c*, FILE **stream*);

Beschreibung

`Die Funktion putc()` ist äquivalent zu `fputc()`. Es ist als Makro und als Funktion definiert. Als Makro kann `putc()` *c* und *stream* mehr als einmal auswerten. Daher sollten diese Argumente niemals Ausdrücke mit Seiteneffekten sein.

`Die Funktion putc_unlocked()` (siehe Seite 412 unter `getc_unlocked( )` ...) ist funktional gleichwertig mit `putc()`, mit der Ausnahme, dass sie nicht threadsicher implementiert ist. Sie kann deshalb in einem Multithread-Programm nur sicher genutzt werden, wenn der Thread, der sie aufruft, das entsprechende (FILE *) Objekt besitzt. Dies ist der Fall nach einem erfolgreichen Aufruf der Funktionen `flockfile()` oder `ftrylockfile()`.

Returnwert

Siehe `fputc()`.

Fehler

Siehe `fputc()`.

Hinweis

Wenn `putc()` als Makro verwendet wird, kann es die Argumente *c* oder *stream* mit Seiteneffekten falsch behandeln. Insbesondere `putc(`*c*`, *`*f*`++)` wird normalerweise nicht korrekt arbeiten. Statt dessen sollte `fputc()` benutzt werden.

Die Zeichen werden nicht sofort in die externe Datei geschrieben, sondern in einem C-internen Puffer zwischengespeichert (siehe Abschnitt „Pufferung von Datenströmen“ auf Seite 77).

Ob `putc()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

BS2000
Bei der Ausgabe in Textdateien werden die Steuerzeichen für Zwischenraum (`\n`, `\t`, etc.) je nach Art der Textdatei in ihre entsprechende Wirkung umgesetzt (siehe Abschnitt „Zwischenraumzeichen“ auf Seite 85). ❑

Siehe auch

`fputc()`, `getc_unlocked()`, `stdio.h`.

putchar - Byte threadsicher in Standard-Ausgabestrom schreiben

Syntax #include <stdio.h>

int putchar(int *c*);

int putchar_unlocked(int *c*);

Beschreibung

Der Funktionsaufruf `putchar(`*c*`)` ist äquivalent zu `putc(`*c*`, stdout)`. `putchar()` ist sowohl als Makro als auch als Funktion realisiert.

Die Funktion `putchar_unlocked()` (siehe Seite 412 unter `getc_unlocked( )` ...) ist funktional gleichwertig mit `putchar()`, mit der Ausnahme, dass sie nicht threadsicher implementiert ist. Sie kann deshalb in einem Multithread-Programm nur sicher genutzt werden, wenn der Thread, der sie aufruft, das entsprechende (FILE *) Objekt besitzt. Dies ist der Fall nach einem erfolgreichen Aufruf der Funktionen `flockfile()` oder `ftrylockfile()`.

Returnwert Siehe `fputc()`.

Hinweis Die Zeichen werden nicht sofort in die externe Datei geschrieben, sondern in einem C-internen Puffer zwischengespeichert (siehe Abschnitt „Pufferung von Datenströmen“ auf Seite 77).

Weitere Informationen zur Ausgabe in Textdateien, v.a. zur Umsetzung der Steuerzeichen für Zwischenraum (`\n`, `\t` etc.), finden Sie in Abschnitt „Zwischenraumzeichen“ auf Seite 85.

Ob `putchar()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

Siehe auch `getchar(), getchar_unlocked(), putc(), putc_unlocked(), stdio.h`.

putchar_unlocked - Byte threadsicher in Standard-Ausgabestrom schreiben

Syntax #include <stdio.h>

int putchar_unlocked(int *c*);

Beschreibung

siehe `getc_unlocked()`.

putenv - Umgebungsvariable ändern oder hinzufügen

Syntax

```
#include <stdlib.h>

int putenv (const char *string);
```

Beschreibung

`putenv()` ändert den Wert einer vorhandenen Umgebungsvariablen oder definiert eine neue Umgebungsvariable. *string* muss auf eine Zeichenkette der Form "*name*=*value*" zeigen. *name* steht für den Namen einer Umgebungsvariablen, *value* für den ihr zugewiesenen Wert. Wenn *name* mit einer existierenden Umgebungsvariablen identisch ist, wird der zugehörige Wert *value* mit der neuen Angabe überschrieben. Wenn *name* eine neue Umgebungsvariable ist, wird die Umgebung um diese erweitert. In jedem Fall wird *string* Teil der Umgebung und ändert damit die Umgebung.

Der von *string* belegte Speicherplatz wird nicht mehr verwendet, wenn `putenv()` einer vorhandenen Umgebungsvariablen einen neuen Wert zuweist.

`putenv()` ist nicht threadsicher.

Returnwert

0	bei Erfolg.
≠ 0	bei Fehler, z.B. wenn nicht genügend Speicherplatz vorhanden ist. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler

`putenv()` schlägt fehl, wenn gilt:

`ENOMEM`	Es steht nicht genügend Speicherplatz zur Verfügung.

Hinweis

`putenv()` verändert die Umgebung, auf die `environ` zeigt, und kann in Verbindung mit `getenv()` verwendet werden.

`putenv()` kann `malloc()` verwenden, um die Umgebung zu vergrößern.

Eine mögliche Fehlerquelle ist der Aufruf von `putenv()` mit einer automatischen Variablen als Argument und einer anschließenden Rückkehr von der aufrufenden Funktion, während *string* noch immer Teil der Umgebung ist.

BS2000
Beim Start eines Programms aus der POSIX-Shell wird neben den Voreinstellungen für die Umgebung auch die SDF-P-Variablenstruktur `SYSPOSIX` als Umgebungsdefinition ausgewertet (siehe auch `environ`). `putenv()` verändert die SDF-P-Variablen jedoch nicht. Die POSIX-Umgebung entspricht dem, worauf `environ` zeigt. `SYSPOSIX.`*name* ist im BS2000, also außerhalb des POSIX-Subsystems, definiert. ❑

Siehe auch `environ`, `exec`, `getenv()`, `malloc()`, `stdlib.h`, Abschnitt „Umgebungsvariablen“ auf Seite 71.

putmsg, putpmsg - Nachricht auf eine STREAMS-Datei senden

Syntax

```
#include <stropts.h>

int putmsg(int fildes, const struct strbuf *ctlptr,
        const struct strbuf *dataptr, int flags);

int putpmsg(int fildes, const struct strbuf *ctlptr,
        const struct strbuf *dataptr, int band, int flags);
```

Beschreibung

`putmsg()` erstellt aus den angegebenen Puffern eine Nachricht und sendet diese an eine STREAMS-Datei. Die Nachricht kann entweder einen Datenteil, einen Steuerteil oder beides enthalten. Die zu sendenden Daten- und Steuerteile werden voneinander unterschieden, indem sie in verschiedene Puffer geschrieben werden (siehe unten). Die Semantik der Teile ist durch das STREAMS-Modul definiert, das die Nachricht empfängt.

Die Funktion `putpmsg()` hat dieselbe Funktionalität wie `putmsg()`, aber sie gibt dem Benutzer die Möglichkeit, Nachrichten mit verschiedenen Prioritäten zu senden.
Alle hier für `putmsg()` beschriebenen Informationen gelten auch für `putpmsg()`, Ausnahmen werden explizit gekennzeichnet.

fildes ist ein Dateideskriptor, der auf einen offenen Stream verweist. *ctlptr* und *dataptr* weisen jeweils auf eine `strbuf`-Struktur, die folgende Elemente enthält:

```
int maxlen;     /* nicht verwendet */
int len;        /* Länge der Daten */
void *buf;      /* Zeiger auf Puffer für Daten */
```

ctlptr weist auf die Struktur, die den in die Nachricht aufzunehmenden Steuerteil beschreibt (falls vorhanden). Das Feld `buf` in der `strbuf`-Struktur weist auf den Puffer, in dem die Steuerinformationen stehen, und das Feld `len` gibt die Anzahl der zu sendenden Bytes an. Das Feld `maxlen` wird in `putmsg()` nicht verwendet (siehe `getmsg()`). Auf gleiche Weise beschreibt *dataptr* den Datenteil, der in die Nachricht aufgenommen werden soll. *flags* gibt an, was für ein Nachrichtentyp gesendet werden soll (siehe unten).

Zum Senden des Datenteils einer Nachricht muss *dataptr* ungleich dem Nullzeiger sein, und das Feld `len` von *dataptr* muss einen Wert ≥ 0 enthalten. Zum Senden des Steuerteils einer Nachricht müssen die entsprechenden Werte für *ctlptr* gesetzt sein. Ein Daten-(Steuer-)Teil wird nicht gesendet, wenn entweder *dataptr* (*ctlptr*) der Nullzeiger ist oder das entsprechende `len`-Feld auf -1 gesetzt ist.

Wird bei `putmsg()` ein Steuerteil angegeben, und ist *flags* auf `RS_HIPRI` gesetzt, wird eine Nachricht mit hoher Priorität geschickt.
Ist kein Steuerteil angegeben und *flags* auf `RS_HIPRI` gesetzt, schlägt `putmsg()` fehl und setzt `errno` auf `EINVAL`.

Wenn *flags* auf 0 gesetzt ist, wird eine normale Nachricht geschickt (Priorität=0).
Ist weder ein Steuer- noch ein Datenteil angegeben und ist *flags* auf 0 gesetzt, wird keine Nachricht gesendet und der Wert 0 zurückgegeben.

Der STREAMS-Kopf garantiert, dass der Steuerteil einer von `putmsg()` erzeugten Nachricht mindestens 64 Bytes lang ist.

Für `putpmsg()` werden andere Flags verwendet: *flags* ist eine Bitmaske, die entweder `MSG_HIPRI` oder `MSG_BAND` oder 0 enthält (die Werte schließen sich gegenseitig aus).
Wenn *flags* auf 0 gesetzt ist, schlägt `putpmsg()` fehl und setzt `errno` auf `EINVAL`.
Wenn ein Steuerteil angegeben ist und *flags* auf `MSG_HIPRI` und *band* auf 0 gesetzt sind, wird eine Nachricht mit hoher Priorität gesendet.
Wenn *flags* auf `MSG_HIPRI` gesetzt ist, und entweder kein Steuerteil angegeben ist oder *band* ≠ 0 ist, scheitert `putpmsg()` und setzt `errno` auf `EINVAL`.
Wenn *flags* auf `MSG_BAND` gesetzt ist, wird eine Nachricht in der durch *band* angegebenen Prioritätsklasse gesendet.
Wenn kein Steuer- und kein Datenteil angegeben und *flags* auf `MSG_BAND` gesetzt ist, wird keine Nachricht gesendet und 0 zurückgegeben.

Normalerweise blockiert `putmsg()`, wenn die Schreib-Warteschlange des Streams auf Grund von internen Kontrollfluss-Bedingungen voll ist. Bei Nachrichten mit hoher Priorität blockiert `putmsg()` in diesem Falle jedoch nicht.
Bei anderen Nachrichten blockiert `putmsg()` nicht bei voller Schreib-Warteschlange, wenn `O_NDELAY` oder `O_NONBLOCK` gesetzt ist. Statt dessen schlägt der Aufruf fehl, und `errno` wird auf `EAGAIN` gesetzt.

`putmsg()` oder `putpmsg()` blockieren unabhängig von der Priorität und `O_NDELAY` oder `O_NONBLOCK` auch dann, wenn sie auf die Verfügbarkeit von Nachrichtenblöcken im Stream warten. Eine Teilnachricht wird nicht gesendet.

Returnwert

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

Fehler `putmsg()` und `putpmsg()` schlagen fehl, wenn gilt:

`EAGAIN`	Eine Nachricht ohne Priorität wurde angegeben, das Flag `O_NDELAY` oder `O_NONBLOCK` ist gesetzt, und die Schreib-Warteschlange des STREAM ist auf Grund von internen Kontrollfluss-Bedingungen voll oder für die zu erzeugende Nachricht konnten keine Puffer zugewiesen werden.
`EBADF`	*fildes* ist kein gültiger, zum Schreiben offener Dateideskriptor.
`EINTR`	Ein Signal wurde während des Systemaufrufs `putmsg()` abgefangen.
`EFAULT`	*ctlptr* oder *dataptr* weisen über den zugewiesenen Adressraum hinaus.

EINVAL Ein undefinierter Wert wurde in *flags* angegeben, oder
flags ist auf RS_HIPRI oder MSG_HIPRI gesetzt, und es wurde kein Steuerteil bereitgestellt oder
der durch *fildes* referenzierte STREAM oder Multiplexer ist streamabwärts über einen Multiplexer angeschlossen.

nur für putpmsg():
flags ist auf MSG_HIPRI gesetzt und es gilt *band* ≠ 0.

ENOSR Für die zu erstellende Nachricht konnte wegen zu geringem STREAMS-Speicherplatz kein Puffer zugewiesen werden.

ENOSTR Zu *fildes* gehört kein STREAM.

ENXIO Ein Hangup wurde streamabwärts für den angegebenen Stream generiert.

EPIPE oder EIO
fildes referenziert eine STREAM-basierte Pipe und das andere Ende der Pipe ist geschlossen. Für den rufenden Prozess wird das Signal SIGPIPE erzeugt.

ERANGE Der Datenteil der Nachricht hat eine Größe, die nicht in dem Bereich liegt, der durch die maximale und minimale Paketgröße des obersten Stream-Moduls vorgegeben wurde.
ERANGE wird auch zurückgegeben, wenn der Steuerteil der Nachricht größer ist als die konfigurierte maximale Größe des Steuerteils einer Nachricht, oder wenn der Datenteil einer Nachricht größer ist als die konfigurierte maximale Größe des Datenteils einer Nachricht.

putmsg() und putpmsg() schlagen ebenfalls fehl, wenn eine asynchrone STREAMS-Fehlermeldung den Stream-Kopf vor dem Aufruf von putmsg() bzw. putpmsg() erreicht hat. In diesem Fall bezieht sich errno auf den Fehler, der in der STREAMS-Fehlermeldung enthalten ist.

Hinweis Wenn zwei Prozesse eine FIFO-Datei eröffnen, wobei der eine mit putmsg() eine Nachricht hoher Priorität schreibt und der andere mit getmsg() eine Nachricht hoher Priorität liest, können Nachrichten verlorengehen. Dieser Verlust kann vermieden werden, wenn der Sendeprozess durch sleep zwischen den einzelnen putmsg() verlangsamt wird.

Siehe auch getmsg(), poll(), read(), write(), stropts.h.

putpwent - Benutzer in Benutzerkatalog eintragen *(Erweiterung)*

Syntax

#include <pwd.h>

int putpwent(const struct passwd *p, FILE *f);

Beschreibung

`putpwent()` schreibt die Benutzerdaten aus der Kennwort-Struktur *p* in den Benutzerkatalog. Der aufrufende Prozess muss Sonderrechte haben.

p ist eine Kennwort-Struktur, die entweder mit `getpwent()`, `getpwuid()` oder `getpwnam()` ermittelt und anschließend verändert wurde.

f wird nur aus Kompatibilitätsgründen unterstützt, aber nicht ausgewertet.

Returnwert

0	bei Erfolg.
≠0	bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler

`putpwent()` schlägt fehl, wenn gilt:

`EINVAL`	Benutzerdaten sind ungültig.
`EFAULT`	Eine ungültige Adresse der `passwd`-Struktur wurde angegeben.
`ENOENT`	Benutzer ist unbekannt.
`EPERM`	Der aufrufende Prozess hat keine Sonderrechte.

Hinweis

Eine Kennwortdatei `/etc/passwd` gibt es im POSIX-Subsystem nicht. Die Benutzerdaten werden intern im Benutzerkatalog hinterlegt (siehe Handbuch „POSIX Grundlagen (BS2000/OSD)“).

Siehe auch

`getpwent()`, Handbuch „POSIX Grundlagen (BS2000/OSD)“.

puts - Zeichenkette in Standard-Ausgabestrom schreiben

Syntax #include <stdio.h>

int puts(const char **s*);

Beschreibung

`puts()` schreibt die Zeichenkette, auf die *s* zeigt, und ein Zeilenendezeichen in den Standard-Ausgabestrom `stdout`. Das abschließende Nullbyte wird nicht geschrieben.

Die Strukturkomponenten `st_ctime` und `st_mtime` der Datei werden zwischen der erfolgreichen Ausführung von `puts()` und der nächsten erfolgreichen Beendigung eines Aufrufs von `fflush()` oder `fclose()` für denselben Datenstrom oder einem Aufruf von `exit()` oder `abort()` für die Änderung markiert (siehe `sys/stat.h`).

Returnwert

nicht negative Zahl	bei Erfolg.
`EOF`	bei Fehler. Das Fehlerkennzeichen für den Datenstrom wird gesetzt. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler Siehe `fputc()`.

Hinweis Im Gegensatz zu `fputs()` fügt `puts()` ein Zeilenendezeichen an.

Das abschließende Nullbyte von *s* wird nicht mit ausgegeben.

Ob `puts()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

Weitere Informationen zur Ausgabe in Textdateien, v.a. zur Umsetzung der Steuerzeichen für Zwischenraum (`\n`, `\t` etc.), finden Sie im Abschnitt „Zwischenraumzeichen“ auf Seite 85.

Siehe auch `fputs()`, `fopen()`, `putc()`, `stdio`, `stdio.h`.

pututxline - utmpx-Eintrag schreiben

Syntax #include <utmpx.h>

struct utmpx *_pututxline_ (const struct utmpx *_utmpx_);

Beschreibung

Siehe `endutxent()`.

Returnwert Zeiger auf eine *utmpx*-Struktur, die eine Kopie des hinzugefügten utmpx-Eintrages enthält bei Erfolg.

Nullzeiger bei Fehler. `errno` wird nicht gesetzt.

Hinweis Um `pututxline()` aufrufen zu können, muss der Prozess über entsprechende Zugriffsrechte verfügen.

Siehe auch `utmpx.h`.

putw - Maschinenwort in Datenstrom schreiben

Syntax

```
#include <stdio.h>

int putw(int w, FILE *stream);
```

Beschreibung

`putw()` schreibt das Maschinenwort *w* auf den Ausgabestrom *stream* an die Position, auf die der Lese-/Schreibzeiger zeigt, falls er definiert ist. Die Größe eines Maschinenworts entspricht dem Datentyp `int` und ist von Rechner zu Rechner verschieden. Im C-Laufzeitsystem ist ein `int`-Datentyp 4 Byte lang. `putw()` setzt keine bestimmte Ausrichtung der Datei voraus und verursacht auch keine solche.

Die Strukturkomponenten `st_ctime` und `st_mtime` der Datei werden zwischen der erfolgreichen Ausführung von `putw()` und der nächsten erfolgreichen Beendigung eines Aufrufs von `fflush()` oder `fclose()` für denselben Datenstrom oder einem Aufruf von `exit()` oder `abort()` für die Änderung markiert (siehe `sys/stat.h`).

`putw()` ist nicht threadsicher.

Returnwert

0	bei Erfolg.
≠ 0	bei Fehler. Das Fehlerkennzeichen für den Datenstrom wird gesetzt. `errno` wird gesetzt, um den Fehler anzuzeigen.
BS2000 `EOF`	bei Fehler. ❑

Fehler

Siehe `fputc()`.

Hinweis

Auf Grund der möglichen Unterschiede in Maschinenwortgröße und Byteanordnung sind Dateien, die mit `putw()` geschrieben wurden, rechnerabhängig und sollten nicht mit `getw()` auf einem anderen Rechner gelesen werden.

Da `putw()` Fehler nicht explizit anzeigt (-1 ist ein gültiger Integer-Wert), sollten Sie zusätzlich `ferror()` verwenden, um abzuprüfen, ob vor oder nach dem Schreiben ein Fehler auftrat.
Die Zeichen werden nicht sofort in die externe Datei geschrieben, sondern in einem C-internen Puffer zwischengespeichert (siehe Abschnitt „Pufferung von Datenströmen“ auf Seite 77).

Bei der Ausgabe in Textdateien werden die Steuerzeichen für Zwischenraum (`\n`, `\t`, etc.) je nach Art der Textdatei in ihre entsprechende Wirkung umgesetzt (siehe Abschnitt „Zwischenraumzeichen“ auf Seite 85).

Ob `putw()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

Siehe auch `fopen()`, `fputc()`, `fwrite()`, `getw()`, `stdio.h`, `sys/stat.h`.

putwc - Langzeichen in Datenstrom schreiben

Syntax

```
#include <wchar.h>

Optional
#include <stdio.h> ❑

wint_t putwc(wint_t wc, FILE *stream);
```

Beschreibung

`putwc()` entspricht der Funktion `fputwc()` mit folgendem Unterschied: Wenn sie als Makro implementiert ist, kann sie *stream* mehrmals auswerten. *stream* sollte daher niemals ein Ausdruck mit Seiteneffekten sein.

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Returnwert Siehe `fputwc()`.

Fehler Siehe `fputwc()`.

Hinweis `putwc(wc, *f++)` funktioniert vermutlich nicht wie erwartet. Daher wird die Verwendung dieser Funktion nicht empfohlen. Stattdessen sollte `fputwc()` verwendet werden.

Siehe auch `fputwc()`, `stdio.h`, `wchar.h`.

putwchar - Langzeichen in Standard-Ausgabestrom schreiben

Syntax

```
#include <wchar.h>

wint_t putwchar(wint_t wc);
```

Beschreibung

Der Funktionsaufruf `putwchar(`*wc*`)` entspricht dem von `putwc(`*wc*`, stdout)`.

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Returnwert Siehe `fputwc()`.

Siehe auch `fputwc()`, `putwc()`, `wchar.h`.

qsort - Datentabelle sortieren

Syntax #include <stdlib.h>

void qsort (void* *base*, size_t *nel*, size_t *width*, int (**compar*) (const void *, const void *));

Beschreibung

Die Funktion `qsort()` ist eine Realisierung des Quicksort-Algorithmus. Sie sortiert eine Tabelle von Daten. Die Daten der Tabelle werden in aufsteigender Reihenfolge sortiert, entsprechend der Vergleichsfunktion. *base* zeigt auf das Element am Anfang der Tabelle. *nel* ist die Anzahl der Elemente in der Tabelle. *width* spezifiziert die Größe eines jeden Elements in Byte. *compar()* ist der Name der vom Benutzer definierten Vergleichsfunktion, die von `qsort()` mit zwei Argumenten aufgerufen wird, die auf die zu vergleichenden Elemente zeigen. Diese Funktion muss eine ganze Zahl kleiner, gleich oder größer als null zurückgeben, um anzuzeigen, ob das erste Argument kleiner, gleich oder größer als das zweite ist.

Die Vergleichsfunktion kann etwa wie folgt definiert sein:

```
/* Programmausschnitt 1 vergleicht zwei char-Werte */
int comp(const void *a, const void *b)
{
   if(*((const char *)a) < *((const char *) b) )
      return(-1);
   else if(*((const char *)a) > *((const char *) b ) )
      return(1);
   return(0);
}

/* Programmausschnitt 2 vergleicht zwei integer-Werte */
int compare(const void *a, const void *b)
{
   return  ( *((const int *) a) - *((const int *) b) );
}
```

Hinweis Die Vergleichsfunktion muss nicht jedes Byte vergleichen, und so können die Elemente zusätzlich zu den zu vergleichenden Werten beliebige Daten enthalten.

Erweiterung
Abweichend vom XPG4 wird die Reihenfolge von Vektorelementen, für die die Vergleichsfunktion Gleichheit feststellt, nicht verändert. ❑

Siehe auch `stdlib.h`.

raise - Signal an aufrufenden Prozess senden

Syntax #include <signal.h>

int raise (int *sig*);

Beschreibung

Wenn die Funktion mit POSIX-Funktionalität aufgerufen wird, verhält sie sich XPG4-konform, wie folgt:

`raise()` sendet das Signal *sig* an den aufrufenden Prozess. Die definierten Signale sind in `signal.h` aufgelistet.

Werden Threads verwendet, so wirkt sich die Funktion auf den Prozess oder auf einen Thread wie folgt aus:

– Sendet ein Signal an den ablaufenden Thread; Die Wirkung von `raise(sig)` ist äuqivalent zum Aufruf `pthread_kill(pthread_self()`, `sig`).

– *BS2000*
Wenn die Funktion mit BS2000-Funktionalität aufgerufen wird, verhält sie sich abweichend, wie folgt:

– Mit `raise()` lassen sich STXIT-Ereignisse simulieren sowie STXIT-unabhängige Signale senden (selbst definierte und vom C-Laufzeitsystem vordefinierte).

– Für *sig* kann folgende Untermenge der Signale, die in `signal.h` definiert sind, eingesetzt werden:

Signal	STXIT-Klasse	Bedeutung
SIGHUP	ABEND	Abbruch der Dialogstationsleitung
SIGINT	ESCPBRK	Unterbrechung von der Dialogstation mit K2
SIGILL	PROCHK	Ausführung einer ungültigen Instruktion
SIGABRT	–	raise-Signal für Programmbeendigung mit `_exit(-1)`
SIGFPE	PROCHK	fehlerhafte Gleitkommaoperation
SIGKILL	–	raise-Signal für Programmbeendigung mit `exit(-1)`
SIGSEGV	ERROR	Speicherzugriff mit unerlaubtem Segmentzugriff
SIGALRM	RTIMER	ein Zeitintervall ist abgelaufen (Realzeit)
SIGTERM	TERM	Signal bei Programmbeendigung
SIGUSR1	–	vom Benutzer definiert
SIGUSR2	–	vom Benutzer definiert
SIGDVZ	PROCHK	Division durch 0
SIGXCPU	RUNOUT	CPU-Zeit ist aufgebraucht
SIGTIM	TIMER	ein Zeit-Intervall ist abgelaufen (CPU-Zeit)
SIGINTR	INTR	SEND-MESSAGE-Kommando

❑

Returnwert 0 wenn das Signal erfolgreich gesendet wurde.

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

Fehler `raise()` schlägt fehl, wenn gilt:

Erweiterung
`EINVAL` Der Wert von *sig* ist eine ungültige Signalnummer. ❑

Hinweis `raise(int sig)` verwendet folgenden `kill`-Aufruf, um das Signal an den aufrufenden Prozess zu senden:

```
kill(getpid(), sig);
```

Unter `kill()` ist eine detaillierte Liste der Fehlerbedingungen aufgeführt.

BS2000
Mit Ausnahme von `SIGKILL` und `SIGSTOP` können die oben aufgelisteten Signale mit der Funktion `signal()` abgefangen werden (siehe `signal()`).

Wenn das Programm keine Behandlung von `raise`-Signalen vorsieht, wird der Prozess bei Eintritt eines Signals mit `exit(-1)` beendet und es werden folgende Meldungen ausgegeben:

```
"CCM0101 signal occured: signal"
"CCM0999 Exit -1"
```

Das Signal `SIGABRT` führt zu einer Programmbeendigung mit `_exit(-1)`. Im Unterschied zu `exit(-1)` werden die mit `atexit()` registrierten Beendigungsroutinen nicht aufgerufen und offene Dateien nicht geschlossen.

Das Signal `SIGKILL` führt zu einer Programmbeendigung mit `exit(-1)`. Im Unterschied zu `SIGABRT` kann `SIGKILL` nicht abgefangen werden, d.h. `signal`-Aufrufe, die als Argument den Namen einer selbst geschriebenen Funktion oder `SIG_IGN` angeben, sind für `SIGKILL` nicht zulässig. ❑

Siehe auch `atexit()`, `exit()`, `_exit()`, `kill()`, `sigaction()`, `signal()`, `signal.h`.

rand - Pseudo-Zufallszahlen (int) generieren

Syntax #include <stdlib.h>

int rand(void);

void srand(unsigned int *seed*);

Beschreibung

`rand()` liefert eine positive, ganze Zufallszahl aus dem Bereich [0, 2^{15}-1].

Ein `rand`-Aufruf wählt Werte aus einer Folge von Pseudo-Zufallszahlen aus, unter Verwendung eines multiplikativen, kongruenten Zufallsgenerators. Der Generator hat eine Periode von 2^{32}.

`rand()` ist nicht threadsicher. Verwenden Sie bei Bedarf die reentrante Funktion `rand_r()`.

Returnwert Zufallszahl aus dem Intervall [0, 2^{15}-1] bei Erfolg.

Hinweis Der Zufallsgenerator lässt sich mit `srand()` initialisieren bzw. rücksetzen. Unterbleibt die Initialisierung, beginnt der Zufallsgenerator mit seinem voreingestellten Wert.

Siehe auch `drand48()`, `rand_r()`, `random()`, `srand()`, `stdlib.h`.

rand_r - Pseudo-Zufallszahlen (int) threadsicher generieren

Syntax #include <stdlib.h>

int rand_r(unsigned int **seed*);

Beschreibung

Die Funktion `rand_r()` ist die threadsichere Version von `rand()`.

Die Funktion `rand_r()` liefert eine ganzzahlige Pseudo-Zufallszahl aus dem Bereich 0 bis 2^{15}-1. Wenn `rand_r()` mit demselben Anfangswert für das Objekt, auf das *seed* zeigt, aufgerufen wird und dieses Objekt zwischen aufeinander folgenden Aufrufen von `rand_r()` nicht verändert wird, wird die gleiche Folge von Pseudo-Zufallszahlen erzeugt.

Returnwert Die Funktion `rand_r()` gibt eine Pseudo-Zufallszahl zurück.

Siehe auch `rand(), stdlib()`.

random - Pseudo-Zufallszahlen erzeugen

Syntax

```
#include <stdlib.h>

long random(void);
```

Beschreibung

siehe `initstate()`.

`random()` erzeugt Pseudo-Zufallszahlen im Bereich von 0 bis 2^{31}-1.

`random()` ist nicht threadsicher. Verwenden Sie bei Bedarf die reentrante Funktion `rand_r()`.

Returnwert Pseudo-Zufallszahl (siehe `initstate()`).

Beispiel

```
/* Initialize an array and pass it to initstate. */

static long state1[32] = { 3, 0x9a319039, 0x32d9c024, 0x9b663182, 0x5da1f342,
0x7449e56b, 0xbeb1dbb0, 0xab5c5918, 0x946554fd, 0x8c2e680f, 0xeb3d799f,
0xb11ee0b7, 0x2d436b86, 0xda672e2a, 0x1588ca88, 0xe369735d, 0x904f35f7,
0xd7158fd6, 0x6fa6f051, 0x616e6b96, 0xac94efdc, 0xde3b81e0, 0xdf0a6fb5,
0xf103bc02, 0x48f340fb, 0x36413f93, 0xc622c298, 0xf5a42ab8, 0x8a88d77b,
0xf5ad9d0e, 0x8999220b, 0x27fb47b9 };

main()

{

unsigned seed;

int n;

seed = 1;

n = 128;

initstate(seed, state1, n);

setstate(state1);

printf("%d\0", random());

}
```

Siehe auch `drand48()`, `rand()`, `rand_r()`, `srand()`, `stdlib.h` .

read - Bytes aus Datei lesen

Syntax #include <unistd.h>

ssize_t read(int *fildes*, void **buf*, size_t *nbyte*);

Beschreibung

`read()` liest *nbyte* Bytes aus der Datei, die dem Dateideskriptor *fildes* zugeordnet ist, in den Puffer, auf den *buf* zeigt.

fildes ist ein Dateideskriptor, der von einem Aufruf von `creat()`, `open()`, `dup()`, `fcntl()` oder `pipe()` zurückgegeben wird.

Wenn *nbyte* gleich 0 ist, liefert `read()` nur den Wert 0 und *buf*.

In Dateien, die das Suchen unterstützen (z.B. normale Dateien), beginnt `read()` an einer Dateiposition, die vom mit *fildes* verbundenen Lese-/Schreibzeiger zur Verfügung gestellt wird. Der Lese-/Schreibzeiger wird um die Byteanzahl erhöht, die tatsächlich gelesen wurde.

Dateien, die das Suchen nicht unterstützen (z.B. Terminal-Gerätedateien) lesen immer von der aktuellen Position. Der Wert des Lese-/Schreibzeigers einer solchen Datei ist nicht definiert.

Nach dem aktuellen Dateiende findet keine Datenübertragung statt. Wenn die Anfangsposition am oder nach dem Dateiende liegt, wird der Wert 0 zurückgeliefert.

Beim Versuch, von einer leeren Pipe oder FIFO zu lesen, geschieht Folgendes:

– Wenn kein Prozess die Pipe zum Schreiben geöffnet hat, liefert `read()` den Wert 0, um das Dateiende anzuzeigen.

– Wenn ein Prozess die Pipe zum Schreiben geöffnet hat und `O_NONBLOCK` gesetzt ist, liefert `read()` den Returnwert -1 und besetzt `errno` mit `EAGAIN`.

– Wenn ein Prozess die Pipe zum Schreiben geöffnet hat und `O_NONBLOCK` nicht gesetzt ist, blockiert `read()` solange, bis Daten geschrieben wurden oder bis die Pipe von allen Prozessen geschlossen wird, die diese zum Schreiben geöffnet hatten.

Beim Versuch, aus einer Datei zu lesen, die keine Pipe oder FIFO ist, die nichtblockierendes Lesen unterstützt und für die zurzeit keine Daten verfügbar sind, geschieht Folgendes:

– Wenn `O_NONBLOCK` gesetzt ist, liefert `read()` den Wert -1 und besetzt `errno` mit `EAGAIN`.

– Wenn `O_NONBLOCK` nicht gesetzt ist, blockiert `read()` solange, bis Daten verfügbar werden.

– Die Verwendung des `O_NONBLOCK`-Flags hat keine Wirkung, wenn Daten verfügbar sind.

`read()` liest Daten, die zuvor in eine Datei geschrieben wurden. Wenn ein Teil einer normalen Datei vor dem Dateiende nicht beschrieben wurde, liefert `read()` Nullbytes. `lseek()` z. B. erlaubt es, den Lese-/Schreibzeiger hinter das Ende von in einer Datei existierenden Daten zu positionieren. Werden später Daten an diese Position geschrieben, liefern nachfolgende Leseoperationen in der Lücke zwischen dem ehemaligen Dateiende und den neu geschriebenen Daten solange Nullbytes, bis Daten in die Lücke geschrieben werden.

Bei erfolgreicher Beendigung und wenn *nbyte* größer als 0 ist, aktualisiert `read()` die Strukturkomponente `st_atime` der Datei (siehe `sys/stat.h`) und liefert die Anzahl der gelesenen Bytes. Diese Anzahl ist niemals größer als *nbyte*. Der Returnwert kann kleiner als *nbyte* sein, wenn die Anzahl der in der Datei verbleibenden Bytes kleiner als *nbyte* ist, wenn `read()` durch ein Signal unterbrochen wurde oder wenn die Datei eine Pipe, FIFO oder Gerätedatei ist und weniger als *nbyte* Bytes sofort zum Lesen verfügbar sind. So kann z.B. ein `read`-Aufruf für eine Datei, die einem Terminal zugeordnet ist, genau eine Eingabezeile liefern.

Wenn ein `read`-Aufruf von einem Signal unterbrochen wird, bevor er Daten lesen kann, liefert er den Wert -1, wobei `errno` gleich `EINTR` gesetzt wird.

Wenn ein `read`-Aufruf von einem Signal unterbrochen wird, nachdem er erfolgreich einige Daten lesen konnte, liefert er die Anzahl der gelesenen Bytes.

Werden Threads verwendet, so wirkt sich die Funktion auf den Prozess oder auf einen Thread wie folgt aus: Beim Versuch, von einer leeren Pipe oder FIFO zu lesen, geschieht Folgendes: ... Wenn ein Prozess die Pipe zum Schreiben geöffnet hat und O_NONBLOCK nicht gesetzt ist, blockiert `read()` den aufrufenden Thread solange, bis Daten geschrieben wurden oder bis die Pipe von allen Prozessen geschlossen wird, die diese zum Schreiben geöffnet hatten. Beim Versuch, aus einer Datei zu lesen, die keine Pipe oder FIFO ist, die nichtblockierendes Lesen unterstützt und für die zurzeit keine Daten verfügbar sind, geschieht Folgendes: ... Wenn O_NONBLOCK nicht gesetzt ist, blockiert `read()` den aufrufenden Thread solange, bis Daten verfügbar werden.

Returnwert Anzahl der tatsächlich gelesenen Bytes
bei erfolgreicher Beendigung.

0	bei Dateiende.
-1	bei Fehler. Der Inhalt des Puffers, auf den *buf* zeigt, ist unbestimmt. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `read()` schlägt fehl, wenn gilt:

`EAGAIN`	Das Flag `O_NONBLOCK` ist für den Dateideskriptor gesetzt und der Prozess würde durch die Leseoperation angehalten werden.

Erweiterung

EAGAIN	Der Systemspeicher, der für „raw“-Ein-/Ausgabe zur Verfügung steht, ist vorübergehend nicht ausreichend, oder in einer Terminal-Gerätedatei warten keine Daten darauf, gelesen zu werden, und O_NONBLOCK ist gesetzt, oder in einem Datenstrom wartet keine Nachricht darauf, gelesen zu werden, und O_NONBLOCK ist gesetzt. ❑
EBADF	*fildes* ist kein gültiger, zum Lesen geöffneter Dateideskriptor.
EFAULT	*buf* weist über den zugewiesenen Adressraum des Prozesses hinaus.
EINTR	Die Leseoperation wurde durch ein Signal unterbrochen und es wurden keine Daten übertragen.
EINVAL	Es wurde versucht, von einem Datenstrom zu lesen, der mit einem Multiplexer verbunden ist.
EIO	Ein physikalischer Ein-/Ausgabefehler ist aufgetreten, oder der Prozess ist Mitglied einer Hintergrund-Prozessgruppe und versucht, von seinem steuernden Terminal zu lesen. Der Prozess ignoriert oder blockiert das Signal SIGTTIN, oder die Prozessgruppe ist verwaist.
ENXIO	Eine Anforderung für ein nicht existierendes Gerät wurde gemacht, oder die Anforderung lag außerhalb der Fähigkeiten des Geräts.

Hinweis

Die Anzahl der tatsächlich gelesenen Bytes kann kleiner sein als die Angabe in *nbytes*, wenn vorher das Zeilenende erreicht wird (nur bei Textdateien), sowie bei Dateiende oder Fehler.

Um sicherzugehen, dass nicht mehr Bytes gelesen werden, als der Puffer aufnehmen kann, sollte sizeof() verwendet werden.

Ob read() für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

Siehe auch

fcntl(), lseek(), open(), pipe(), unistd.h, Abschnitt „Allgemeine Terminalschnittstelle“ auf Seite 96.

readdir - aus Dateiverzeichnis lesen

Name **readdir, readdir64**

Syntax
```
#include <dirent.h>
#include <sys/types.h>

struct dirent *readdir (DIR *dirp);
struct dirent64 *readdir64 (DIR *dirp);
```

Beschreibung

Der Datentyp `DIR`, der in der Include-Datei `dirent.h` definiert ist, repräsentiert einen Dateiverzeichnisstrom, der eine geordnete Folge aller Einträge eines speziellen Dateiverzeichnisses ist. Dateiverzeichniseinträge repräsentieren Dateien; Dateien können asynchron zur Ausführung von `readdir()` aus einem Dateiverzeichnis entfernt bzw. zu einem Dateiverzeichnis hinzugefügt werden.

`readdir()` liefert einen Zeiger auf eine Struktur, die den nächsten, nichtleeren Dateiverzeichniseintrag in dem Dateiverzeichnisstrom enthält, auf den *dirp* zeigt, und positioniert den Dateiverzeichnisstrom auf den nächsten Eintrag. Sobald sie das Ende des Dateiverzeichnisstroms erreicht, liefert sie den Nullzeiger. Die Struktur `dirent` beschreibt einen Dateiverzeichniseintrag (siehe `dirent.h`).

`readdir()` liefert keine Dateiverzeichniseinträge, die leere Namen enthalten. Wenn Einträge für . (aktuelles Dateiverzeichnis) und .. (übergeordnetes Dateiverzeichnis) existieren, wird genau ein Eintrag für . und einer für .. zurückgeliefert.

Der von `readdir()` zurückgelieferte Zeiger zeigt auf Daten, die von einem weiteren Aufruf von `readdir()` für denselben Dateiverzeichnisstrom überschrieben werden können. Diese Daten werden von einem weiteren Aufruf von `readdir()` für einen anderen Dateiverzeichnisstrom nicht überschrieben.

Wenn nach dem letzten Aufruf von `opendir()` oder `rewinddir()` eine Datei aus dem Dateiverzeichnis entfernt oder zu diesem hinzugefügt wurde, ist es unbestimmt, ob ein nachfolgender Aufruf von `readdir()` einen Eintrag für diese Datei zurückliefert.

`readdir()` kann mehrere Dateiverzeichniseinträge bei einer einzelnen Leseoperation zwischenspeichern; `readdir()` aktualisiert die Strukturkomponente `st_atime` des Dateiverzeichnisses jedes Mal, wenn das Dateiverzeichnis wirklich gelesen wird (siehe auch `sys/stat.h`).

Nach einem `fork`-Aufruf können entweder der Vater- oder der Sohnprozess (aber nicht beide) die Ausführung fortsetzen, indem `readdir()`, `rewinddir()` oder `seekdir()` verwendet werden. Wenn sowohl Vater- als auch Sohnprozess diese Funktionen aufrufen, ist das Ergebnis nicht definiert.

Es besteht kein funktionaler Unterschied zwischen `readdir()` und `readdir64()`, außer dass `readdir64()` eine `dirent64`-Struktur verwendet.

Die Struktur `dirent64` entspricht der von `dirent`, mit Ausnahme folgender Komponente:

```
ino64_t d_ino
```

`readdir()` und `readdir64()` sind nicht threadsicher. Verwenden Sie anstelle von `readdir()` bei Bedarf die reentrante Funktion `readdir_r()`. Für die Funktion `readdir64()` existiert derzeit noch kein reentrantes Pendant.

Returnwert `readdir()` und `readdir64()`:

Zeiger auf ein Objekt vom Typ `struct dirent`
bei erfolgreicher Beendigung.

Nullzeiger — wenn das Ende des Dateiverzeichnisses erreicht wird. `errno` wird nicht verändert.

Nullzeiger — wenn ein Fehler auftritt. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `readdir()` und `readdir64()` schlagen fehl, wenn gilt:

`EBADF` — Das Argument *dirp* zeigt nicht auf einen offenen Dateiverzeichnisstrom.

`ENOENT` — Die aktuelle Position des Verzeichnisstroms ist ungültig.

`EOVERFLOW` — Ein Wert in der zurückgegebenen Struktur kann nicht korrekt dargestellt werden.

Hinweis `readdir()` sollte in Verbindung mit `opendir()`, `closedir()` und `rewinddir()` verwendet werden, um den Inhalt des Dateiverzeichnisses zu untersuchen. Da `readdir()` den Nullzeiger sowohl am Ende des Dateiverzeichnisses als auch bei Fehler liefert, sollte eine Anwendung, die Fehlersituationen überprüfen will, `errno` gleich 0 setzen. Danach sollte die Anwendung `readdir()` aufrufen, den Wert von `errno` prüfen und, wenn dieser ungleich 0 ist, das Auftreten eines Fehlers annehmen.

`pclose()` wird nur für POSIX-Dateien ausgeführt.

Siehe auch `closedir()`, `opendir()`, `readdir_r()`, `rewinddir()`, `dirent.h`, `sys/stat.h`, `sys/types.h`.

readdir_r - aus Dateiverzeichnis threadsicher lesen

Syntax

#include <sys/types.h>

#include <dirent.h>

int readdir_r(DIR **dirp*, struct dirent **entry*, struct dirent ***result*);

Beschreibung

Die Funktion `readdir_r()` ist die tread-sichere Version der Funktion `readdir()`.

Die Funktion `readdir_r()` initialisiert die Struktur `dirent`, auf die *entry* verweist, mit dem nächsten, nichtleeren Dateiverzeichniseintrag im Dateiverzeichnisstrom, auf den *dirp* zeigt, speichert einen Zeiger auf diese Struktur an der Stelle, auf die *result* zeigt, ab und positioniert den Dateiverzeichnisstrom auf den nächsten Eintrag.

Der Speicher, auf den *entry* zeigt, muss groß genug sein, um für das char-Feld `d_name` aus der Struktur `dirent` schlimmstenfalls `{NAME_MAX}` plus ein Zeichen aufnehmen zu können.

Bei erfolgreicher Rückkehr besitzt der Zeiger, der für **result* zurückgegeben wurde, denselben Wert wie das Argument *entry*. Wenn das Ende des Dateiverzeichnisstroms erreicht ist, hat dieser Zeiger den Wert NULL.

Die Funktion `readdir_r()` liefert keine Dateiverzeichniseinträge zurück, die leere Namen enthalten.

`readdir_r()` kann mehrere Dateiverzeichniseinträge bei einer einzelnen Leseoperation zwischenspeichern; `readdir_r()` aktualisiert die Strukturkomponente `st_atime` des Dateiverzeichnisses jedes Mal, wenn das Dateiverzeichnis wirklich gelesen wird.

Returnwert

0 Bei Erfolg.

Fehlernummer

sonst, um den Fehler anzuzeigen. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler

Die Funktion `readdir_r()` schlägt fehl, wenn gilt:

`EBADF` Das Argument *dirp* verweist nicht auf einen offenen Dateiverzeichnisstrom.

Siehe auch `readdir(), dirent(), types()`.

readlink - Inhalt eines symbolischen Verweises lesen

Syntax #include <unistd.h>

int readlink(const char **path*, char **buf*, size_t *bufsiz*);

Beschreibung

`readlink()` schreibt den Inhalt des symbolischen Verweises, auf den *path* weist, in den Puffer *buf*, welcher die Länge *bufsiz* hat. Der Inhalt des Verweises ist bei der Rückgabe nicht mit einem Nullbyte abgeschlossen.

Returnwert Anzahl von Zeichen, die in den Puffer geschrieben wurden bei erfolgreicher Beendigung.

-1 bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen. Der Inhalt des Puffers bleibt unverändert

Fehler `readlink()` schlägt fehl, wenn gilt:

`EACCES` Für eine der Pfadpräfix-Komponenten in *path* besteht kein Suchrecht.

`EFAULT` *path* oder *buf* befinden sich außerhalb des allokierten Adressbereichs des Prozesses.

`EINVAL` *path* ist kein symbolischer Verweis.

Erweiterung
`EINVAL` Es wurde versucht, auf eine BS2000-Datei zuzugreifen. ❑

`EIO` Ein Ein-/Ausgabefehler ist beim Lesen oder Schreiben des Dateisystems aufgetreten.

`ELOOP` Bei der Übersetzung von *path* treten zu viele symbolische Verweise auf.

`ENAMETOOLONG`
Die Länge des *path*-Arguments überschreitet `{PATH_MAX}` oder die Länge einer *path*-Komponente überschreitet `{NAME_MAX}`.

`ENOENT` Die genannte Datei existiert nicht.

`ENOSYS` Das Dateisystem unterstützt keine symbolischen Verweise.

`ENOTDIR` Eine der Pfadpräfix-Komponenten in *path* ist kein Verzeichnis.

Hinweis `readlink()` greift nur auf POSIX-Dateien zu.

Siehe auch `stat()`, `symlink()`, `unistd.h`.

readv - vektorielles Lesen aus einer Datei

Syntax #include <sys/uio.h>

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

Beschreibung

siehe `read()`.

`readv()` verhält sich wie `read()`, liest jedoch die Eingabedaten aus der zu *fildes* gehörenden Datei in die *iovcnt*-Puffer, die als Elemente des Feldes *iov* spezifiziert sind:
iov[0], *iov*[1], ..., *iov*[*iovcnt*-1].
Es muss gelten $0 < iovcnt \leq$ {IOV_MAX}

Die Struktur `iovec` enthält folgende Elemente:

```
addr_t    iov_base;
size_t    iov_len;
```

Jeder `iovec`-Eintrag gibt die Basisadresse und Länge eines Speicherbereichs (Puffer) an, in den Daten gebracht werden sollen. `readv()` füllt einen Puffer immer vollständig, bevor es mit dem nächsten weitermacht.

Bei Erfolg gibt `readv()` die Anzahl der tatsächlich gelesenen und in den Puffer geschriebenen Bytes zurück. Bei Erreichen des Dateiendes wird 0 zurückgegeben.

Returnwert

ganze Zahl >0	bei Erfolg. Die Zahl gibt die Anzahl der tatsächlich gelesenen Bytes an.
0	wenn beim Lesen das Dateiende (EOF) erreicht wurde.
-1	bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen. Der Inhalt der Puffer ist unbestimmt.

Fehler `readv()` schlägt fehl, wenn gilt:

`EAGAIN`	Das Flag `O_NONBLOCK` ist für den Dateideskriptor gesetzt und der Prozess würde durch die Leseoperation angehalten werden.
Erweiterung	
`EAGAIN`	Der Systemspeicher, der für „raw“-Ein-/Ausgabe zur Verfügung steht, ist vorübergehend nicht ausreichend, oder in einer Terminal-Gerätedatei warten keine Daten darauf, gelesen zu werden, und `O_NONBLOCK` ist gesetzt, oder in einen Datenstrom wartet keine Nachricht darauf, gelesen zu werden, und `O_NONBLOCK` ist gesetzt. ❑
`EBADF`	*fildes* ist kein gültiger, zum Lesen geöffneter Dateideskriptor.

EBADMSG	Die Datei ist eine STREAM-Datei im control-normal-mode, aber die Nachricht, die darauf wartet, gelesen zu werden, enthält einen Steuerteil.
EFAULT	*iov* weist über den zugewiesenen Adressraum des Prozesses hinaus.
EINTR	Die Leseoperation wurde durch ein Signal unterbrochen. Es wurden keine Daten übertragen.
EINVAL	Es wurde versucht, von einem Datenstrom zu lesen, der mit einem Multiplexer verbunden ist oder die Summe der *iov-len*-Werte im Feld *iov* bewirkte einen ssize_t-Überlauf oder *iovcnt* ≤ 0 oder *iovcnt* >16.
EIO	Ein physikalischer Ein-/Ausgabefehler ist aufgetreten, oder der Prozess ist Mitglied einer Hintergrund-Prozessgruppe und versucht, von seinem steuernden Terminal zu lesen. Der Prozess ignoriert oder blockiert das Signal SIGTTIN, oder die Prozessgruppe ist verwaist.
EISDIR	*fildes* beschreibt ein Verzeichnis, das nicht mit readv() gelesen werden darf. Stattdessen sollte readdir() verwendet werden.
ENXIO	Eine Anforderung für ein nicht existierendes Gerät wurde gemacht, oder die Anforderung lag außerhalb der Fähigkeiten des Geräts.
ENOLINK	*fildes* liegt auf einem fernen Rechner, und die Verbindung zu diesem Rechner ist nicht mehr aktiv.

Ein readv() von einer STREAMS-Datei ist auch dann erfolglos, wenn am Stream-Kopf eine Fehlermeldung empfangen wird. In diesem Fall wird errno auf den Wert gesetzt, der in der Fehlermeldung zurückgegeben wird. Bei Auftreten eines Hangups im Stream, der gerade gelesen wird, läuft readv() normal weiter, bis die Lesewarteschlange des Stream-Kopfes entleert ist. Danach wird 0 zurückgegeben.

Siehe auch fcntl(), ioctl(), lseek(), open(), pipe(), stropts.h, sys/uio.h, unistd.h.

realloc - Speicherbereich verändern

Syntax #include <stdlib.h>

void *realloc(void **ptr*, size_t *size*);

Beschreibung

`realloc()` verändert die Größe des Speicherbereiches, auf den *ptr* zeigt, in *size* Bytes.

`realloc()` ist Teil des C-spezifischen Speicherverwaltungspaketes, das angeforderte und wieder freigegebene Speicherbereiche intern verwaltet. Neue Anforderungen werden zuerst aus bereits verwalteten Bereichen zu erfüllen versucht, dann erst vom Betriebssystem.

ptr ist ein Zeiger auf den Anfang des zu verändernden Speicherplatzes. *ptr* muss zuvor von `malloc()` oder `calloc()` zurückgeliefert worden sein.

size ist ein ganzzahliger Wert, der die neue Größe in Byte angibt.

Returnwert Zeiger auf den Anfang des geänderten Speicherbereiches bei Erfolg.

Nullzeiger falls `realloc()` den Speicherplatz nicht verändern konnte, z.B. weil der noch vorhandene Speicherplatz nicht ausreicht oder ein Fehler auftrat. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `realloc()` schlägt fehl, wenn gilt:

`ENOMEM` Es ist nicht genügend Speicherplatz verfügbar.

Hinweis Wenn `realloc()` die Größe eines Speicherbereiches ändert, kann u.U. der zugewiesene Block verschoben sein. In solchen Fällen ist der Inhalt des als Argument übergebenen Zeigers nicht identisch mit dem Returnwert.
Der Inhalt des Blocks bleibt bis zum Minimum der alten (beim Vergrößern) bzw. neuen Größe (beim Verkleinern) erhalten.

Liefert `realloc()` den Nullzeiger, kann evtl. der Block, auf den *ptr* zeigt, zerstört worden sein!

Ist *ptr* ein Nullzeiger, funktioniert `realloc()` wie ein `malloc`–Aufruf für die angegebene Größe.

Siehe auch `calloc()`, `free()`, `malloc()`, `stdlib.h`.

realpath - echten Dateinamen/Pfadnamen ausgeben

Syntax #include <stdlib.h>

char *realpath (const char **file_name*, char **resolved_name*);

Beschreibung

`realpath()` leitet aus dem in *file_name* angegebenen Pfadnamen einen absoluten Pfadnamen ab, in dem alle symbolischen Verweise und Referenzen auf '.' und '..' aufgelöst sind. Dieser „echte“ Pfadname wird bis zu `{MAX_PATH}` Bytes in *resolved_name* gespeichert.

Es können sowohl relative als auch absolute Pfadnamen verarbeitet werden. Bei absoluten Pfadnamen und relativen Pfadnamen, deren aufgelöster Name nicht relativ ausgedrückt werden kann (z.B. `../../reldir`), wird der aufgelöste absolute Name zurückgegeben. Für die anderen relativen Pfadnamen wird der aufgelöste relative Name zurückgegeben.

resolved_name muss groß genug sein, um den aufgelösten Pfadnamen aufzunehmen.

Returnwert Zeiger auf *resolved_name*
bei Erfolg.

Nullzeiger sonst. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `realpath()` schlägt fehl, wenn gilt:

`EACCES` Für eine Komponente von *file_name* besteht keine Lese- oder Suchberechtigung.

`EINVAL` Das Argument *file_name* oder *resolved_name* ist ein Nullzeiger.

`EIO` Es trat ein Ein-/Ausgabefehler auf beim Lesen aus dem Dateisystem.

`ENAMETOOLONG`
Die Länge des *file_name*-Arguments überschreitet `{PATH_MAX}`, oder die Länge einer Komponente von *file_name* überschreitet `{NAME_MAX}`.

Bei der Auflösung eines symbolischen Verweises in *path* kam es zu einem Zwischenergebnis, dessen Länge `{PATH_MAX}` überschreitet.

`ENOENT` Eine Komponente des Pfadpräfixes existiert nicht oder *file_name* ist ein leerer String.

`ENOTDIR` Eine Komponente des Pfadpräfixes ist kein Verzeichnis.

`ENOMEM` Es steht nicht mehr genügend Speicherplatz zur Verfügung.

Hinweis `realpath()` behandelt nullterminierte Zeichenketten.

Sie sollten Ausführungsrechte für alle Verzeichnisse besitzen, die sich im gegebenen und aufgelösten Pfad befinden.

`realpath()` kehrt unter Umständen nicht zum aktuellen Verzeichnis zurück, falls ein Fehler auftritt.

Siehe auch `getcwd()`, `sysconf()`, `stdlib.h`.

re_comp, re_exec - Übersetzen und Ausführen regulärer Ausdrücke

Syntax #include <re_comp.h>

char *re_comp(const char **string*);
int re_exec(const char **string*);

Beschreibung

`re_comp()` kompiliert eine Zeichenkette in ein internes Format, das für einen Mustervergleich geeignet ist. `re_exec` vergleicht die Zeichenkette, auf die *string* zeigt, mit dem letzten regulären Ausdruck, der `re_comp()` übergeben wurde.

Wird `re_comp()` mit dem Wert 0 oder einem Nullzeiger aufgerufen, bleibt der aktuelle reguläre Ausdruck unverändert.

Die Zeichenketten, die an `re_comp()` und an `re_exec()` übergeben werden, müssen mit dem NULL-Zeichen abgeschlossen werden. Die Zeichenketten können abschließende oder eingebettete Zeilenvorschubzeichen (NEWLINE) enthalten.

`re_comp()` und `re_exec()` unterstützen einfache reguläre Ausdrücke. Die für den Mustervergleich geltenden Regeln sind im Folgenden beschrieben.

1. Reguläre Ein-Zeichen-Ausdrücke passen nach folgenden Regeln zu einem Zeichen:

1.1 Ein gewöhnliches Zeichen (keines der unter 1.2 aufgeführten Sonderzeichen) ist ein regulärer Ausdruck, der zu sich selbst passt.

1.2 Ein Gegenschrägstrich (Backslash: \ )gefolgt von einem Sonderzeichen ist ein regulärer Ein-Zeichen-Ausdruck, der zu diesem Sonderzeichen passt. Definiert sind folgende Sonderzeichen:

- Punkt (.), Stern (*), öffnende eckige Klammer ([) und Gegenschrägstrich (\). Diese Zeichen sind Sonderzeichen, ausgenommen, sie stehen in eckigen Klammern [ ] (siehe 1.4).
- Circumflex (^) ist ein Sonderzeichen, wenn es am Anfang eines regulären Ausdrucks steht oder wenn es in eckigen Klammern steht und unmittelbar auf die öffnende Klammer folgt ([^]) (siehe 1.4).
- Dollar ($) ist ein Sonderzeichen, wenn es am Ende eines regulären Ausdrucks steht (siehe 3.2).
- Das Begrenzungszeichen, das verwendet wird, um einen regulären Ausdruck zu begrenzen (delimit), ist für diesen regulären Ausdruck ein Sonderzeichen.

1.3 Ein Punkt (.) ist ein regulärer Ein-Zeichen-Ausdruck, der mit Ausnahme von NEWLINE zu allen Zeichen passt.

1.4 Eine nichtleere Zeichenkette, die in eckige Klammern eingeschlossen ist, ist ein regulärer Ein-Zeichen-Ausdruck, der zu jedem einzelnen Zeichen in dieser Zeichenkette passt. Ist allerdings das erste Zeichen in der Zeichenkette der Circumflex (^), passt der reguläre Ausdruck zu allen Zeichen mit Ausnahme der verbleibenden Zeichen in der Zeichenkette und NEWLINE. Das Zeichen ^ hat diese „Ausschlussbedeutung“ aber nur, wenn es das erste Zeichen nach der öffnenden eckigen Klammer ist.
Das Minuszeichen (-) kann verwendet werden, um einen Bereich aufeinander folgender ASCII-Zeichen darzustellen, z.B. sind [0-9] und [0123456789] gleichbedeutend. Das Minuszeichen ist kein Sonderzeichen, wenn es das erste (evtl. nach einem ^) oder letzte Zeichen in der Zeichenkette ist.
Die schließende eckige Klammer beendet eine solche Zeichenkette nicht, wenn sie das erste Zeichen (evtl. nach einem ^) in der Zeichenkette ist. Zum Beispiel passt [ ]a-f zu einer schließenden eckigen Klammer] oder zu einem der Zeichen a, b, c, d, e oder f.
Die vier Zeichen Punkt (.), Stern (*), öffnende eckige Klammer ([) und Gegenschrägstrich (\) stehen innerhalb einer solchen Zeichenkette für sich selbst.

2. Mit Hilfe der folgenden Regeln können reguläre Ausdrücke aus regulären Ein-Zeichen-Ausdrücken konstruiert werden:

2.1 Ein regulärer Ein-Zeichen-Ausdruck ist ein regulärer Ausdruck, der zu allem passt, was zu dem regulären Ein-Zeichen-Ausdruck passt.

2.2 Ein Stern (*), gefolgt von einem regulären Ein-Zeichen-Ausdruck, ist ein regulärer Ausdruck, der zu 0 oder mehreren Vorkommen des Ein-Zeichen-Ausdrucks passt.
Falls es mehrere Möglichkeiten gibt, wird die längste, am weitesten links gelegene Teilkette gewählt, die passt.

2.3 Ein regulärer Ein-Zeichen-Ausdruck, gefolgt von \{*m*\}, \{*m*,\} oder \{*m*,*n*\} ist ein regulärer Ausdruck, der zu einem mehrfachen Vorkommen des Ein-Zeichen-Ausdrucks passt. *m* und *n* müssen nichtnegative Ganzzahlen kleiner 256 sein.
{*m*\} passt zu genau *m* Vorkommen, \{*m*,\ passt zu mindestens *m* Vorkommen und \{*m*,*n*\} passt zu Vorkommen zwischen *m* und *n* (inklusive).
Falls es mehrere Möglichkeiten gibt, wird die größte Zahl von Vorkommen gewählt, die passt.

2.4 Die Konkatenation von regulären Ausdrücken ist ein regulärer Ausdruck, der zu einer Zeichenkette passt, die durch Konkatenation derjenigen Zeichenketten entsteht, die zu den entsprechenden Komponenten des regulären Ausdrucks passen.

2.5 Ein regulärer Ausdruck, der zwischen den Zeichenfolgen \(und \) steht, passt zu allem, was zu dem zwischen diesen Zeichenfolgen stehenden regulären Ausdruck passt.

2.6 Der Ausdruck *n* passt zu der gleichen Folge von Zeichen, die weiter vorne in demselben regulären Ausdruck zu einem zwischen \(und \) eingeschlossenem Ausdruck passte. *n* ist eine Ziffer; der betreffende Teilausdruck beginnt mit dem *n*-ten Vorkommen von \, gezählt wird von links. Zum Beispiel passt ^\(.\)\1$ zu einer Zeile, die aus einer Zeichenkette und ihrer Wiederholung besteht.

3. Zusätzlich kann ein regulärer Ausdruck so eingeschränkt werden, dass er nur zu einem Zeilenanfang, einem Zeilenende (oder beidem) passt:

3.1 Ein Circumflex (^) am Anfang eines vollständigen regulären Ausdrucks bedeutet, dass dieser Ausdruck nur zu einer Zeichenkette am Zeilenanfang passt.

3.2 Ein Dollarzeichen ($) am Ende eines vollständigen regulären Ausdrucks bedeutet, dass dieser Ausdruck nur zu einer Zeichenkette am Zeilenende passt.
Zum Beispiel bedeutet ^*vollständigerAusdruck*$, dass der vollständige reguläre Ausdruck zu der gesamten Zeile passen muss. Der leere reguläre Ausdruck, d.h. //, ist äquivalent zu dem letzten aufgetretenen regulären Ausdruck.

Returnwert für `re_comp()`:

Nullzeiger wenn `re_comp()` die übergebene Zeichenkette erfolgreich kompiliert hat

Zeichenkette mit Fehlermeldung
sonst.

für `re_exec()`:

1 wenn *string* mit dem letzten kompilierten Ausdruck übereinstimmt.

0 wenn *string* nicht mit dem letzten kompilierten Ausdruck übereinstimmt.

-1 wenn der kompilierte Ausdruck ungültig ist (bei einem internen Fehler).

Fehler `re_comp()` gibt bei einem Fehler eine der folgenden Zeichenketten zurück:

No previous regular expression
Regular expression too long
unmatched \(
missing]
too many \(\)

Hinweis Zu einem Bereich gehören alle Zahlen, die zwischen der internen Darstellung der beiden Bereichsgrenzen liegen. Dies kann in EBCDIC- und ASCII-Umgebung unterschiedlich sein.

Aus Gründen der Portabilität zu Implementierungen, die sich an frühere Versionen des X/Open-Standards halten, werden die Funktionen `regcomp()` und `regexec()` statt der hier beschriebenen empfohlen.

Siehe auch `regcmp()`, `regexec()`, `re_comp.h`.

regcmp, regex - regulären Ausdruck übersetzen und ausführen

Syntax

```
#include <libgen.h>

char *regcmp (const char *string1 [, char *string2, ...]  /* , (char *) 0) */;
char *regex (const char *re, const char *subject [ , char *ret0, ... ] );
extern char *__loc1;
```

Beschreibung

`regcmp()` kompiliert den regulären Ausdruck, der durch die Konkatenation der Argumente entsteht. Das Ende der Argumentkette ist ein Nullzeiger. Als Ergebnis gibt `regcmp()` einen Zeiger auf den in ein internes Format übersetzten Ausdruck zurück. Der Speicherplatz für den kompilierten Ausdruck wird mit `malloc()` bereitgestellt. Der Benutzer ist für die Freigabe des so zugewiesenen Speicherplatzes verantwortlich, wenn der Platz nicht mehr benötigt wird.
Die Rückgabe eines Nullzeigers durch `regcmp()` zeigt an, dass ein Argument einen ungültigen Wert hat..

`regex()` sucht ein durch `regcmp()` kompiliertes Muster *re* in der Zeichenkette *subject*. Zusätzliche Argumente werden an `regex()` übergeben, um übereinstimmende Teilausdrücke zurückzuerhalten. Werden nicht genügend Argumente für alle zurückgelieferten Treffer angegeben, ist das Verhalten von `regex()` undefiniert.

Der globale Zeichenzeiger `_loc1` weist auf das erste übereinstimmende Byte in *subject*.

`regcmp()` und `regex()` wurden weitgehend vom Editor `ed()` übernommen, wobei Syntax und Semantik jedoch leicht verändert wurden. Die gültigen Symbole und ihre jeweiligen Bedeutungen sind wie folgt:

`[]*.^` Diese Symbole haben dieselbe Bedeutung wie in `ed()`.

`$` Dieses Symbol entspricht dem Ende der Zeichenkette (`\n` entspricht einem NEWLINE-Zeichen).

`-` Das von Klammern umschlossene Minuszeichen bedeutet *einschließlich*. So ist beispielsweise `[a-z]` gleichbedeutend mit `[abcd...xyz]`. Das `-` kann nur dann für sich selbst stehen, wenn es als das erste oder letzte Zeichen verwendet wird. So passt beispielsweise der Ausdruck `[]-]` zu den Zeichen `]` und `-`.

`+` Ein regulärer Ausdruck mit nachfolgendem `+` bedeutet *einmal oder mehrere Male*. So ist zum Beispiel `[0-9]+` gleichbedeutend mit `[0-9] [0-9]*`.

{m} {m,} {m,u}
Mit `{}` umschlossene ganzzahlige Werte zeigen die Häufigkeit an, mit der der vorangehende reguläre Ausdruck angewendet werden soll. Der Wert *m* ist die Mindestanzahl und *u* das Maximum. *u* muss kleiner als 256 sein. Wenn nur *m* vorhanden ist (z.B. *{m}*), wird damit genau angegeben, wie oft

der reguläre Ausdruck angewendet werden soll. Der Wert {*m,*} ist analog zu {*m,Unendlich*}. Die Operationen mit dem Plus-Zeichen + und dem Stern * sind gleichbedeutend mit {1,} bzw. {0,}.

`( ... )$`*n* Der Wert des geklammerten regulären Ausdrucks soll zurückgegeben werden. Der Wert wird im (*n*+1)ten Argument nach dem Argument *subject* gespeichert. Es sind höchstens zehn geklammerte reguläre Ausdrücke zulässig. `regex()` führt die Zuweisungen auf jeden Fall aus.

`( ... )` Für Gruppierungen werden Klammern verwendet. Ein Operator, z.B. *, +, {}, kann auf Einzelzeichen oder auf einen von Klammern umschlossenen regulären Ausdruck angewendet werden. Beispiel: `(a*(cb+)*)$0`.

Alle oben definierten Symbole sind Sonderzeichen. Daher müssen sie mit einem Gegenschrägstrich \ gekennzeichnet werden, wenn sie für sich stehen sollen.

Returnwert für `regcmp()`:

Zeiger auf den kompilierten regulären Ausdruck
bei Erfolg.

Nullzeiger bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

für `regex()`:

Zeiger auf das nächste Zeichen in *subject*, das nicht zum Muster passt
bei Erfolg.

Nullzeiger bei Fehler.

Fehler `regcmp()` schlägt fehl, wenn gilt:

`ENOMEM` Es steht nicht mehr genügend Speicherplatz zur Verfügung.

Hinweis Das Benutzerprogramm kann möglicherweise keinen Speicherplatz mehr zur Verfügung stellen, wenn `regcmp()` iterativ ohne Freigabe der nicht mehr benötigten Vektoren aufgerufen wird.

Wenn Sie eine dieser Funktionen verwenden, müssen Sie bei der Übersetzung die Bibliothek `libgen` dazubinden (`cc -lgen`).

Beispiel 1 Das folgende Beispiel sucht ein führendes NEWLINE-Zeichen in der Zeichenkette `subject`, auf die `cursor` zeigt.

```
char *cursor, *newcursor, *ptr;
 ...
newcursor = regex((ptr = regcmp("^\n", (char *)0)), cursor);
free(ptr);
```

Beispiel 2 Das folgende Beispiel sucht nach der Zeichenkette `Testing3` und gibt die Adresse des Zeichens hinter dem letzten passenden Zeichen (dem Zeichen `4`) zurück. Die Zeichenkette `Testing3` wird in das Zeichenfeld `ret0` kopiert.

```
char ret0[9];
char *newcursor, *name;
 ...
name = regcmp("([A-Za-z][A-za-z0-9]{0,7})$0", (char *)0);
newcursor = regex(name, "012Testing345", ret0);
```

Beispiel 3 Bei diesem Beispiel wird ein vorübersetzter regulärer Ausdruck in `file.i` (siehe `regcmp()` gegen *string* geprüft.

```
#include "file.i"
char *string, *newcursor;
 ...
newcursor = regex(name, string);
```

Siehe auch `re_comp()`, `re_exec()`, `malloc()`.
`ed()` im Handbuch "SINIX V5.41 Kommandos".

regexp: advance, compile, step, loc1, loc2, locs - reguläre Ausdrücke bearbeiten

Syntax

```
#define INIT declarations
#define GETC () getc code
#define PEEKC() peekc code
#define UNGETC() ungetc code
#define RETURN(ptr) return code
#define ERROR(val) error code

#include <regexp.h>

char *compile(char *instring, char *expbuf, const char *endbuf, int eof);
int step(const char *string, const char *expbuf);
int advance(const char *string, const char *expbuf);
extern char *loc1, *loc2, *locs;
```

Beschreibung

Diese Funktionen sind allgemeine Funktionen zur Behandlung von regulären Ausdrücken in Programmen, die Mustervergleiche von regulären Ausdrücken durchführen. Diese Funktionen werden in der Include-Datei `regexp.h` definiert.

In einem Programm müssen vor der Anweisung `#include <regexp.h>` die folgenden Makros vom Benutzer definiert werden . Diese Makros werden von der Funktion `compile()` benutzt. Die Makros `GETC()`, `PEEKC()` und `UNGETC()` arbeiten mit dem regulären Ausdruck, der als Eingabe an `compile()` übergeben wurde.

`GETC()` liefert den Wert des nächsten Zeichens im regulären Ausdruck. Der Benutzer muss darauf achten, dass wiederholte, aufeinander folgende Aufrufe von `GETC()` aufeinander folgende Zeichen des regulären Ausdrucks ausgeben.

`PEEKC()` liefert das nächste Zeichen im regulären Ausdruck. Der Benutzer muss darauf achten, dass wiederholte, unmittelbar aufeinander folgende Aufrufe von `PEEKC()` immer dasselbe Zeichen liefern, das zudem mit dem nächsten, von `GETC()` gelieferten Zeichen identisch sein sollte.

`UNGETC(`*c*`)` bewirkt, dass beim nächsten Aufruf von `GETC()` und `PEEKC()` das Argument *c* ausgegeben wird. Es ist nur ein Zeichen notwendig, das in die Eingabe zurückgeschoben wird, und dieses Zeichen ist in jedem Fall das letzte von `GETC()` eingelesene Zeichen. Der Wert des Makros `UNGETC(`*c*`)` wird immer ignoriert.

`RETURN(`*ptr*`)` wird bei einer normalen Beendigung der Funktion `compile()` benutzt. Der Wert des Arguments *ptr* ist ein Zeiger auf das Zeichen, das auf das letzte Zeichen des übersetzten regulären Ausdrucks folgt. Dieses Makro ist bei Programmen hilfreich, die Speicherbereiche verwalten.

`ERROR(`*val*`)` entspricht der abnormalen Beendigung der Funktion `compile()`. Das Argument *val* ist eine Fehlernummer (Bedeutung der einzelnen Returnwerte siehe unter Fehler). Der Benutzer muss darauf achten, dass dieser Aufruf nicht zurückkehrt.

Die Funktionen `step()` und `advance()` führen Mustervergleiche durch, bei denen eine Zeichenkette und ein übersetzter regulärer Ausdruck als Eingabe verwendet werden.

`compile()` nimmt als Eingabe einen regulären Ausdruck und erzeugt einen übersetzten Ausdruck, der mit `step()` oder `advance()` verwendet werden kann.

Die Syntax der Funktion `compile()` ist die folgende:

```
char *compile(char *instring, char *expbuf, const char *endbuf, int eof);
```

– Der erste Parameter *instring* wird niemals direkt von der Funktion `compile()` benutzt, ist aber nützlich für Programme, die verschiedene Zeiger auf Eingabezeichen übergeben. Er wird manchmal in den Deklarationen zu `INIT` verwendet (siehe auch unten). Programme, die Funktionen aufrufen, um Zeichen einzugeben, oder die Zeichen aus einem externen Vektor verarbeiten, können hier den Wert `(char*)0` übergeben.

– Der nächste Parameter *expbuf* ist ein Zeiger auf `char`. Er zeigt auf die Stelle, an der der übersetzte reguläre Ausdruck abgelegt werden soll.

– Der Parameter *endbuf* ist um eins höher als die höchste Adresse, in die der übersetzte reguläre Ausdruck eingetragen werden soll. Wenn der übersetzte Ausdruck nicht in (*endbuf*-*expbuf*)-Bytes passt, dann wird `ERROR(50)` aufgerufen.

– Der Parameter *eof* ist das Zeichen, das das Ende eines regulären Ausdruckes kennzeichnet.

Jedes Programm, das die `#include`–Anweisung für `regexp.h` enthält, muss auch eine `#define`–Anweisung für das Makro `INIT` enthalten. Dieses Makro wird für abhängige Vereinbarungen und Initialisierungen verwendet. In den meisten Fällen wird es dazu verwendet, eine Registervariable so zu setzen, dass sie auf den Anfang des regulären Ausdrucks zeigt, so dass diese Registervariable in den Vereinbarungen von `GETC()`, `PEEKC()` und `UNGETC()` verwendet werden kann. Ansonsten kann es benutzt werden, um externe Variablen zu vereinbaren, die von `GETC()`, `PEEKC()` und `UNGETC()` benutzt werden könnten.

Die Funktionen `step()` und `advance()` haben jeweils zwei Parameter:

– *string*, der erste Parameter, ist ein Zeiger auf eine Zeichenkette, die gegen einen regulären Ausdruck geprüft werden soll. Diese Zeichenkette muss mit dem Nullbyte abgeschlossen sein.

– *expbuf*, der zweite Parameter, ist der übersetzte reguläre Ausdruck, der von einem Aufruf der Funktion `compile()` geliefert wurde.

Die Funktion `step()` liefert einen Wert ungleich null, wenn eine Teilfolge von *string* zu dem regulären Ausdruck *expbuf* passt; sie liefert den Wert Null, wenn es keine Übereinstimmung gibt. Solange keine Übereinstimmung vorliegt, werden zwei externe Zeiger als Seiteneffekt des Aufrufs von `step()` gesetzt. Die Variable *loc1* zeigt dann auf das erste, zum regulären Ausdruck passende Zeichen; die Variable *loc2* zeigt auf das Zeichen nach dem letzten Zeichen, das zum regulären Ausdruck passt. Wenn also die gesamte Eingabezeichenkette zum regulären Ausdruck passt, so zeigt *loc1* auf das erste Zeichen von *string,* und *loc2* zeigt auf das Nullbyte am Ende von *string.*

`advance()` liefert einen Wert ungleich null, wenn die erste Teilfolge von *string* zum regulären Ausdruck in *expbuf* passt. Gibt es eine Übereinstimmung, dann wird als Seiteneffekt ein externer Zeiger *loc2* auf *char* gesetzt. Die Variable *loc2* zeigt auf das nächste Zeichen in *string*, das sich hinter dem letzten passenden Zeichen befindet.

Trifft die Funktion `advance()` auf ein Zeichen * oder auf die Zeichenkette `\{\}` im regulären Ausdruck, so setzt sie ihren Zeiger hinter die größtmögliche dazu passende Zeichenkette und ruft sich selbst rekursiv auf, um den Rest der Zeichenkette mit dem Rest des regulären Ausdrucks zu vergleichen. Solange keine Übereinstimmung vorliegt, wird geprüft, ob das gesuchte Muster bereits im vorher erkannten Teilstring enthalten ist. Dabei wird jeweils um eine Stelle zurückgerückt, bis eine Übereinstimmung festgestellt wird oder bis die Stelle in der Zeichenkette erreicht ist, die anfangs zu dem * oder der Zeichenkette `\{\}` passte. In manchen Fällen ist es wünschenswert, dass das Zurückgehen abgebrochen wird, bevor diese Stelle erreicht ist. Ist der externe Zeiger *locs* zu irgendeinem Zeitpunkt während des Zurückgehens identisch mit dieser Stelle in der Zeichenkette, so beendet `advance()` die Schleife und gibt den Wert Null zurück.

Die externen Variablen *circf*, *sed* und *nbra* sind reserviert.

Einfache reguläre Ausdrücke (historische Version)

Ein einfacher regulärer Ausdruck vereinbart eine Menge von Zeichenketten. Wenn eine Zeichenkette in dieser Menge liegt, wird gesagt, dass es auf den einfachen regulären Ausdruck passt.

Ein Muster wird von einem einfachen regulären Ausdruck oder von mehreren einfachen regulären Ausdrücken gebildet. Ein einfacher regulärer Ausdruck besteht aus normalen Zeichen oder aus Metazeichen.

Syntaxelemente zur Bildung von Mustern:

regulärer Ausdruck	**Bedeutung**	**Beispiel**	**passende Zeichenkette**
r+	Einmal oder mehrmals der reguläre Ausdruck *r*. *r* muss von einer der folgenden Formen sein: *r*, *r*, beliebiges Zeichen, [*r*], [*r1*−*r2*], [^*s*], [^*r1*−*r2*], (*r*), (*r1*|*r2*)	u+	u, uu, uuu, ...
r?	Null- oder einmal der reguläre Ausdruck *r*. *r* muss von einer der folgenden Formen sein: *r*, *r*, beliebiges Zeichen, [*r*], [*r1*−*r2*], [^*s*], [^*r1*−*r2*], (*r*), (*r1*|*r2*)	u?	nichts oder u
(*r*)	Zeichenketten, die zu dem regelären Ausdruck *r* passen. *r* kann ein beliebiger Ausdruck sein.	(ok(abc)) (au)*	okabc nichts oder aus, auau, ...
(*r1*|*r2*)	Zeichenketten, die zu dem regulären Ausdruck *r1* oder zu dem regulären Ausdruck *r2* passen.	(ok|ko)	ok oder ko

Innerhalb eines Musters passen alle alphanumerischen Zeichen, die nicht Teil eines Klammerausdrucks, Rückbezugs oder eines Duplikats sind, auf sich selber. Das bedeutet, das Muster *abc* des regulären Ausdrucks passt, wenn es auf eine Menge von Zeichenketten angewendet wird, auf die Zeichenketten verglichen, die auch die Zeichenfolge *abc* enthalten.

Nur einige Zeichen, die Metazeichen, haben eine besondere Bedeutung, wenn sie in einem regulären Ausdruck verwendet werden; andere Zeichen stehen für sich selbst.

Die regulären Ausdrücke, die mit den regexp-Funktionen verfügbar sind, werden folgendermaßen erzeugt:

Ausdruck	**Bedeutung**
c	Das Zeichen *c*, wobei *c* kein Sonderzeichen sein darf.
c	Das Zeichen *c*, wobei *c* irgendein Zeichen ist, außer einer Ziffer im Bereich 1-9.
^	Der Anfang der Zeile, auf der der Vergleich durchgeführt wird.
$	Das Ende der Zeile, auf der der Vergleich durchgeführt wird.
.	Irgendein Zeichen in der Eingabe.

[*s*] Irgendein Zeichen in der Menge *s*, wobei *s* eine Folge von Zeichen ist. Bereiche können als `[`*c*−*c*`]` angegeben werden. In dieser Menge kann das Zeichen `]` nur an erster Stelle stehen, das Zeichen − kann an erster oder letzter Stelle stehen, das Zeichen ^ kann an jeder Stelle stehen, nur nicht an der ersten. Bereiche in einfachen regulären Ausdrücken sind nur gültig, wenn die Kategorie `LC_COLLATE` auf die C-Lokalität gesetzt wird.

[^*s*] Irgendein Zeichen, das nicht in der Menge *s* liegt, wobei *s* wie oben definiert ist.

*r** Null oder mehrere aufeinander folgende Vorkommen des regulären Ausdrucks *r*. Die längste, am weitesten links liegende, passende Zeichenkette wird verwendet.

rx Das Vorkommen des regulären Ausdrucks *r*, gefolgt vom Vorkommen des regulären Ausdrucks *x* (Verkettung).

r`\{`*m*`,`*n*`\}` Irgendeine Anzahl zwischen *m* und *n* aufeinander folgender Vorkommen des regulären Ausdrucks *r*. Der reguläre Ausdruck *r*`\{`*m*`\}` passt bei genau *m* Vorkommen; *r*`\{`*m*`,\}` passt bei mindestens *m* Vorkommen. Die maximale Anzahl der Vorkommen wird geprüft.

`\(`*r*`\)` Der reguläre Ausdruck *r*. Die Folgen `\(` und `\)` werden ignoriert.

`\`*n* Wenn `\`*n* eine Ziffer im Bereich von 1-9 ist und in einem verketteten regulären Ausdruck vorkommt, steht es für den regulären Ausdruck *x*. Dabei ist *x* der *n*-te reguläre Ausdruck, eingeschlossen in `\(` und `\)`, der in dem vorher geketteten regulären Ausdruck vorkam. In dem Muster `\(`*r*`\)`*x*`\(`*y* vergleicht `\`*2* den regulären Ausdruck *y* und ergibt *rxyzy*.

Folgende Zeichen haben eine besondere Bedeutung, wenn sie nicht innerhalb von eckigen Klammern `[ ]` auftreten oder ihnen ein `\` vorangeht: `.`, `*`, `[`, `\`. Andere Sonderzeichen, wie z.B. `$` haben in noch weiter eingeschränkten Umgebungen eine besondere Bedeutung.

Steht das Zeichen ^ am Anfang eines Ausdrucks, können nur Zeichenfolgen passen, die unmittelbar nach einem Zeilenendezeichen stehen oder am Anfang einer jeden Zeichenkette, bei der der Vergleich angewendet wird. Das Zeichen `$` am Ende eines Ausdrucks verlangt ein abschließendes Zeilenendezeichen.

Zwei Zeichen haben nur dann eine besondere Bedeutung, wenn sie innerhalb von eckigen Klammern verwendet werden. Das Zeichen − gibt einen Bereich an, `[`*c*−*c*`]`, außer wenn es direkt nach einer öffnenden oder direkt vor einer schließenden Klammer auftritt, `[`−*c*`]` oder `[`*c*−`]`. In diesem Fall hat es keine besondere Bedeutung. Bei der Verwendung innerhalb von eckigen Klammern hat das Zeichen ^ die Bedeutung „Komplement von“, wenn es unmittelbar auf die öffnende eckige Klammer folgt (`[^`*c*`]`); sonst steht es zwischen eckigen Klammern (`[`*c*`^]`) für das normale Zeichen ^. Die schließende eckige Klammer hat keine besondere Bedeutung mehr, wenn sie direkt dem ersten Zeichen ^ folgt. Sie steht dann als normale schließende Klammer in einem Klammerausdruck.

Die besondere Bedeutung des Operators \ kann nur durch das Voran Stellen eines weiteren \, d.h \\, ausgeschaltet werden.

Rangfolge der Operatoren für einfache reguläre Ausdrücke

[...]	hohe Rangfolge
Verkettung	niedrige Rangfolge

Internationalisierte einfache reguläre Ausdrücke

Zeichenausdrücke in eckigen Klammern werden wie folgt gebildet:

c	ein einzelnes Zeichen *c*, wobei *c* kein Sonderzeichen ist.
[[:*class*:]]	Ein char-Klassenausdruck. Jedes Zeichen vom Typ class, wie durch die Kategorie LC_CTYPE in der Programmlokalität definiert (siehe Handbuch „POSIX Kommandos (BS2000/OSD)“).

Anstelle von *class* kann Folgendes angegeben werden:

alpha	ein Buchstabe
upper	ein Großbuchstabe
lower	ein Kleinbuchstabe
digit	eine Ziffer
xdigit	eine hexadezimale Ziffer
alnum	ein alphanumerisches Zeichen (Buchstabe oder Ziffer)
space	ein Leerzeichen
punct	ein Interpunktionszeichen
print	ein abdruckbares Zeichen
graph	ein sichtbares Zeichen
cntrl	ein Steuerzeichen
[[=*c*=]]	Eine Äquivalenzklasse. Jede Zeicheneinheit, die so definiert ist, als hätte sie dieselbe relative Reihenfolge in der aktuellen Sortierreihenfolge wie *c*. Beispiel: wenn A und a derselben Äquivalenzklasse angehören, dann entsprechen [[=*A*=]*b*] und [[=*a*=]*b*] beide [*Aab*].

[[.*cc*.]] Ein Zeicheneinheits-Symbol. Mehrzeichen-Zeicheneinheiten müssen als Zeicheneinheits-Symbole dargestellt werden, um sie von Einzelzeichen-Zeicheneinheiten unterscheiden zu können. Wenn zum Beispiel *ch* eine gültige Zeicheneinheit ist, dann wird die Zeichenkette `[[.`*ch*`.]]` als ein Element betrachtet, das genau auf dieselbe Zeichenkette passt, während *ch* als einfaches *c* und *h* betrachtet wird. Ist *ch* keine gültige Zeicheneinheit in der aktuellen Definition der Sortierreihenfolge, wird das Symbol als ungültiger Ausdruck behandelt.

[*c*-*c*] Jede Vergleichseinheit im Bereich des Zeichenausdrucks *c*-*c*, wobei *c* ein Zeicheneinheits-Symbol oder eine Äquivalenzklasse sein kann. Befindet sich das Zeichen − unmittelbar nach einer öffnenden eckigen Klammer, zum Beispiel `[-`*c*`]` oder vor einer schließenden, zum Beispiel `[`*c*`-]`, so hat dies keine besondere Bedeutung.

^ Folgt das Zeichen *c* unmittelbar einer öffnenden eckigen Klammer, so ist es das Komplement von zum Beispiel `[^`*c*`]`. Ansonsten hat dies keine besondere Bedeutung.

Bei innerhalb von eckigen Klammern stehenden Ausdrücken ist *a* . nicht Teil der Folge *a* `[[.`*cc*`.]]` oder *a* : nicht Teil der Folge *a* `[[:`*class*`:]]` oder *an* = nicht Teil der Folge *a* `[[=`*c*`=]]`. Diese Folgen stimmen nur mit Zeichenketten überein, die dieselben Folgen haben.

Beispiele für reguläre Ausdrücke

`ab.d`	*ab* beliebiges Zeichen *d*
`ab.*d`	*ab* jede Folge des Zeichens (inkl. kein Zeichen) *d*
`ab[xyz]d`	*ab* eines der Zeichen *x y* oder *z d*
`ab[^c]d`	*ab* jedes Zeichen, ausgenommen *c d*
`^abcd$`	eine Zeile, die nur *abcd* enthält
`a-d`	jedes der Zeichen *a b c* oder *d*

Returnwert

`RETURN()`	bei Erfolg von `compile()`.
≠ 0	bei Erfolg von `step()` und `advance()`.
`ERROR`	bei Fehler von `compile()`.
0	bei Fehler von `step()` und `advance()`.

Fehler

11	zu großer Endpunkt des Bereichs
16	ungültige Zahl
25	`\digit` außerhalb des Bereichs
36	ungültiger oder fehlender Begrenzer
41	keine Suchfolge im Speicher
42	`\(\)` Ungleichgewicht
43	zuviele `\(`
44	mehr als 2 Zahlen in `\{\}`
45	nach `\` wird `}` erwartet
46	in `\{\}` ist die erste Zahl größer als die zweite
49	`[ ]` nicht ausgeglichen
50	regulärer Ausdruck zu umfangreich

Siehe auch `fnmatch()`, `glob()`, `regcomp()`, `regexec()`, `stlocale()`, `regex.h`, `regexp.h`, Handbuch „POSIX Kommandos (BS2000/OSD)“.

remainder - Rest bei Division

Syntax

```
#include <math.h>

double remainder (double x, double y);
```

Beschreibung

`remainder()` gibt den Gleitkommarest der Division x durch y zurück. Genauer gesagt, gibt es den Wert $r = x - yn$ zurück , falls gilt $y \neq 0$. Dabei ist n die ganze Zahl, die am dichtesten beim exakten Wert x/y liegt. Wenn gilt $|n - x/y| = {}^1/_2$, wird für n der gerade Wert gewählt.

Returnwert

Gleitkommarest $r = x-ny$
wenn gilt $y \neq 0$.

`HUGE_VAL` wenn gilt $y = 0$. `errno` wird auf `EDOM` gesetzt.

Fehler

`remainder()` schlägt fehl, wenn gilt:

`EDOM` $y = 0$.

Siehe auch `abs()`, `math.h`.

remove - Datei löschen

Syntax

#include <stdio.h>

int remove(const char **path*);

Beschreibung

`remove()` hat zur Folge, dass die durch *path* angegebene Datei oder das leere Verzeichnis nicht länger unter dem Namen verfügbar ist. Ein weiterer Versuch, die Datei unter dem Namen zu öffnen, wird fehlschlagen, es sei denn, die Datei wird neu angelegt.

Für Dateien ist `remove()` identisch mit `unlink()`. Für Verzeichnisse ist `remove()` identisch mit `rmdir()`.

BS2000
`remove()` ist auch auf Dateien mit Satz-Ein-/Ausgabe anwendbar ❑

Returnwert

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

Fehler

Siehe `unlink()` und `rmdir()`.

Hinweis

Ob `remove()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

BS2000
path kann ein voll- oder teilqualifizierter Dateiname sein. Wird ein teilqualifizierter Dateiname angegeben, löscht `remove()` alle entsprechenden Dateien ohne vorherige Abfrage (`Y/N`). Es wird von der Antwort „`Y`“ ausgegangen.

`remove()` löscht die Dateien nur logisch, d.h. der Katalogeintrag wird gelöscht und der zugewiesene Speicherplatz freigegeben.

Wenn eine Datei durch irgendein Programm geöffnet ist, wird sie nicht gelöscht. ❑

Siehe auch

`rmdir()`, `unlink()`, `stdio.h`.

remque - Element aus Queue entfernen

Syntax #include <search.h>

void remque(void *element);

Beschreibung

siehe `insque()`.

`insque()` und `remque()` ändern Queues, die aus doppelt verketteten Elementen erzeugt werden.
`insque()` fügt *element* in einer Queue ein. `remque()` entfernt den Eintrag *element* aus einer Queue.

rename - Dateiname ändern

Syntax #include <stdio.h>

int rename(const char *_old_, const char *_new_);

Beschreibung

`rename()` ändert den Namen einer Datei. *old* zeigt auf den Pfadnamen der Datei, die umbenannt werden soll. *new* zeigt auf den neuen Pfadnamen der Datei.

Wenn sowohl *old* als auch *new* auf dieselbe existierende Datei verweisen, kehrt die Funktion `rename()` erfolgreich zurück und führt keine weitere Aktion aus.

Wenn *old* auf den Pfadnamen einer Datei zeigt, die kein Dateiverzeichnis ist, darf *new* nicht auf den Pfadnamen eines Dateiverzeichnisses zeigen. Wenn der Verweis existiert, der durch das Argument *new* angegeben wird, wird er entfernt und *old* wird in *new* umbenannt. In diesem Fall bleibt ein Verweis *new* während der umbenannten Operation sichtbar für andere Prozesse und bezieht sich auf die Datei, auf die sich entweder *new* oder *old* bezogen hat, bevor die Operation begann. Sowohl für das Dateiverzeichnis, das *old* enthält, als auch für das Dateiverzeichnis, das *new* enthält, wird das Schreibrecht benötigt.

Wenn *old* auf den Pfadnamen eines Dateiverzeichnisses zeigt, dann darf *new* nicht auf den Pfadnamen einer Datei zeigen, die kein Dateiverzeichnis ist. Wenn das Dateiverzeichnis existiert, das durch *new* angegeben wird, dann wird es entfernt und *old* wird in *new* umbenannt. In diesem Fall existiert ein Verweis *new* während der umbenannten Operation und bezieht sich auf die Datei, auf die sich entweder *new* oder *old* bezogen hat, bevor die Operation begann. Wenn daher *new* ein existierendes Dateiverzeichnis angibt, muss dieses ein leeres Dateiverzeichnis sein.

Der Pfadnamen-Anfang von *new* darf nicht identisch sein mit *old*. Das Schreibrecht wird sowohl für das Dateiverzeichnis, das *old* enthält, als auch für das Dateiverzeichnis, das *new* enthält, benötigt.

Wenn *old* auf den Pfadnamen eines Dateiverzeichnisses zeigt, kann das Schreibrecht für das durch *old* angegebene Dateiverzeichnis benötigt werden und, falls es existiert, für das Dateiverzeichnis, das durch *new* angegeben wird.

Wenn der Verweis existiert, der durch *new* angegeben wird, und der Verweiszähler der Datei durch das Entfernen dieser Datei gleich 0 wird und falls außerdem kein Prozess diese Datei geöffnet hat, wird der Platz freigegeben, der durch diese Datei belegt wird, und auf die Datei kann nicht länger zugegriffen werden. Falls einer oder mehrere Prozesse die Datei geöffnet haben, während der letzte Verweis entfernt wird, wird der Verweis entfernt, bevor `rename()` zurückkehrt, aber die Entfernung der Datei wird aufgeschoben, bis alle Referenzen auf diese Datei geschlossen sind.

Bei erfolgreicher Beendigung kennzeichnet `rename()` die Felder `st_ctime` und `st_mtime` des übergeordneten Dateiverzeichnisses jeder der beiden Dateien zum Aktualisieren.

BS2000
`rename()` ist auch auf Dateien mit Satz-Ein-/Ausgabe unverändert anwendbar. ❑

Returnwert 0 bei Erfolg.

-1 bei Fehler, `errno` wird gesetzt, um den Fehler anzuzeigen. Keine der durch *old* oder *new* benannten Dateien wird geändert oder erzeugt.

BS2000
`errno` wird auf `EMACRO` gesetzt.

Wenn *old* und *new* auf Dateien aus verschiedenen Dateisystemen zeigen, wird nichts verändert. `errno` wird auf `EXDEV` gesetzt. ❑

Fehler `rename()` schlägt fehl, wenn gilt:

`EACCES` Für eine Komponente eines Pfades existiert kein Suchrecht oder für eines der Dateiverzeichnisse, die *old* oder *new* enthalten, existiert kein Schreibrecht; oder das Schreibrecht für eines der Dateiverzeichnisse, auf die *old* oder *new* zeigen, wird benötigt, existiert aber nicht.

`EBUSY` Eines der Dateiverzeichnisse die durch *old* oder *new* angegeben werden, wird zurzeit durch das System oder einen anderen Prozess verwendet, und die Implementierung nimmt dies als einen Fehler an.

Erweiterung
`EDQUOT` Das Verzeichnis, in dem sich der Eintrag mit dem neuen Namen befindet, kann nicht erweitert werden, da der Benutzer die Anzahl der zulässigen Blöcke im Dateisystem, in dem sich das Verzeichnis befindet, überschritten hat. ❑

`EEXIST` oder `ENOTEMPTY`
Der Verweis, der durch *new* angegeben wird, ist ein Dateiverzeichnis, das nicht leer ist.

Erweiterung
`EFAULT` *old* oder *new* zeigen außerhalb des allokierten Adressbereichs des Prozesses.

`EINTR` Während der Ausführung des Systemaufrufs `rename()` wurde ein Signal empfangen. ❑

`EINVAL` Der Dateiverzeichnis-Pfadname *new* enthält einen Pfadnamen-Anfang, der das Dateiverzeichnis *old* bezeichnet (siehe auch „Hinweis“).

Erweiterung
`EIO` Beim Anlegen und Aktualisieren eines Verzeichniseintrags trat ein Ein-/Ausgabe-Fehler auf. ❑

`EISDIR` Das Argument *new* zeigt auf ein Dateiverzeichnis, und das Argument *old* zeigt auf eine Datei, die kein Dateiverzeichnis ist.

Erweiterung

`ELOOP` Zu viele symbolische Verweise traten bei der Übersetzung von *old* oder *new* auf. ❑

BS2000

`EMACRO` Es existiert keine Datei mit dem Namen *old*.
Es ist bereits eine Datei unter dem Namen *old* katalogisiert oder die umzubenennende Datei ist durch ein Programm geöffnet. ❑

`EMLINK` *old* zeigt auf ein Dateiverzeichnis, und der Verweiszähler des Dateiverzeichnisses, das *new* übergeordneten ist, ist größer als `{LINK_MAX}`.

`ENAMETOOLONG`
Die Länge von *old* oder *new* überschreitet `{PATH_MAX}`, oder eine Komponente des Pfades ist länger als `{NAME_MAX}`.

`ENOENT` Der Verweis, der durch *old* bezeichnet wird, existiert nicht, oder *old* bzw. *new* zeigt auf eine leere Zeichenkette.

`ENOSPC` Das Dateiverzeichnis, das *new* enthalten würde, kann nicht erweitert werden.

`ENOTDIR` Eine Komponente des Pfades ist kein Dateiverzeichnis; oder das Argument *old* bezeichnet ein Dateiverzeichnis, und das Argument *new* bezeichnet eine Datei, die kein Dateiverzeichnis ist.

`EROFS` Die angeforderte Operation fordert das Schreiben in ein Dateiverzeichnis, das sich in einem nur zum Lesen eingehängten Dateisystem befindet.

`EXDEV` Die durch *new* und *old* bezeichneten Verweise befinden sich in verschiedenen Dateisystemen.

Hinweis Mit `rename()` kann keine Datei aus POSIX in das BS2000 verlagert werden oder umgekehrt. Zum Beispiel führt die nachstehende Anweisung zum Fehler `EINVAL`:

```
rename(/BS2/hugo, *POSIX(hugo))
```

Ob `rename()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

Siehe auch `link()`, `rmdir()`, `unlink()`, `stdio.h`.

rewind - Lese-/Schreibzeiger auf Datenstrom-Anfang positionieren

Syntax

#include <stdio.h>

void rewind(FILE **stream*);

Beschreibung

Der Aufruf `rewind(`*stream*`)` entspricht dem nachfolgenden Aufruf, außer dass `rewind()` auch die Fehleranzeige von *stream* löscht:

```
(void) fseek(stream, 0L, SEEK_SET);
```

Fehler

Siehe `fseek()` - ausgenommen `EINVAL`.

Hinweis

Da `rewind()` kein Ergebnis liefert, muss eine Anwendung, die Fehler erkennen will, zuerst `errno` gleich 0 setzen, dann `rewind()` aufrufen und dann, wenn `errno` ungleich 0 ist, annehmen, dass ein Fehler aufgetreten ist.

Ob `rewind()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

BS2000
`rewind()` ist auch auf Dateien mit Satz-Ein-/Ausgabe unverändert anwendbar. ❑

Siehe auch

`fseek()`, `fsetpos()`, `stdio.h`.

rewinddir - Lese-/Schreibzeiger auf Dateiverzeichnisstrom-Anfang positionieren

Syntax

```
#include <dirent.h>

Optional
#include <sys/types.h> ❑

void rewinddir(DIR *dirp);
```

Beschreibung

`rewinddir()` setzt die Position des Dateiverzeichnisstroms, auf den *dirp* zeigt, auf den Anfang des Dateiverzeichnisses. Es veranlasst den Dateiverzeichnisstrom auch, den aktuellen Zustand des entsprechenden Dateiverzeichnisses zu berücksichtigen, so wie dies ein Aufruf von `opendir()` machen würde. Wenn *dirp* nicht auf einen Dateiverzeichnisstrom zeigt, ist das Verhalten undefiniert.

Bei `rewinddir()` kann nach einem Aufruf von `fork()` entweder der Vater- oder der Sohnprozess (aber nicht beide) den Dateiverzeichnisstrom unter Verwendung von `readdir()`, `rewinddir()` oder `seekdir()` fortführen. Wenn beide Prozesse diese Funktionen verwenden, ist das Verhalten undefiniert.

Hinweis

`rewinddir()` sollte zusammen mit `opendir()`, `readdir()` und `closedir()` verwendet werden, um den Inhalt eines Dateiverzeichnisses zu untersuchen. Diese Methode wird aus Portabilitätsgründen empfohlen.

`rewinddir()` wird nur für POSIX-Dateien ausgeführt.

Siehe auch `closedir()`, `opendir()`, `readdir()`, `dirent.h`, `sys/types.h`.

rindex - letztes Vorkommen eines Zeichens in Zeichenkette ermitteln

Syntax #include <string.h>

char *rindex(const char *s, int c);

Beschreibung

siehe `strrchr()`.

`rindex()` sucht die letzte Stelle, an der das Zeichen *c* in der Zeichenkette *s* vorkommt, und liefert bei Erfolg einen Zeiger auf die gesuchte Position in *s*.

Das abschließende Nullbyte (`\0`) wird als Zeichen mitberücksichtigt.

Returnwert Zeiger auf die (letzte) Position von *c* in der Zeichenkette *s*, bei Erfolg.

Nullzeiger, wenn *c* in der Zeichenkette *s* nicht enthalten ist.

Hinweis `index()` und `strrchr()` sind äquivalent.

Im BS2000, wie auch in vielen anderen Betriebssystemen, können Sie den Nullzeiger nicht verwenden, um eine NULL-Zeichenkette zu bezeichnen. Ein Nullzeiger ist hier ein Fehler und führt zu einem Abbruch des Programms. Wenn Sie eine NULL-Zeichenkette angeben möchten, müssen Sie einen Zeiger verwenden, der auf eine explizite NULL-Zeichenkette zeigt. Bei einigen Implementierungen der Programmiersprache C auf manchen Rechnern, würde ein Nullzeiger, wenn dereferenziert, eine NULL-Zeichenkette ergeben; dieser nur in den allerseltensten Fällen portierbare Trick wurde in einigen Programmen verwendet. Programmierer, die einen Nullzeiger verwenden, um auf eine leere Zeichenkette zu verweisen, sollten sich dieser Portabilitätsfrage bewußt sein; auch bei Rechnern, bei denen eine Dereferenzierung eines Nullzeigers nicht zum Abbruch des Programms führt, muss sie nicht unbedingt eine NULL-Zeichenkette ergeben.

Das Bewegen von Zeichen wird bei unterschiedlichen Implementierungen auch unterschiedlich ausgeführt. Überlappungen können daher zu unvorhergesehenen Ergebnissen führen.

Siehe auch `index()`, `strchr()`, `strrchr()`.

rint, rintf, rintl - auf nächste ganze Zahl runden

Syntax

```
#include <math.h>

double rint(double x);

float rintf(float x);

long double rintl(long double x);
```

Beschreibung

Die Funktionen geben in Gleitpunktdarstellung jeweils die ganze Zahl zurück, die *x* am nächsten liegt.
`rint()` stellt das Ergebnis dar als Zahl vom Typ `double`, `rintf()` als Zahl vom Typ `float` und `rintl()` als Zahl vom Typ `long double`.

Der zurückgegebene Wert ist entsprechend dem aktuell gesetzten Rundungsmodus des Rechners gerundet. Wenn der Rundungsmodus 'round-to-nearest' gesetzt ist und die Differenz zwischen *x* und dem gerundeten Ergebnis genau 0.5 ist, wird die nächste gerade Ganzzahl zurückgegeben.

Wenn der aktuell eingestellte Rundungsmodus in Richtung positiv unendlich rundet, ist `rint()` identisch zu `ceil()`. Wenn der aktuell eingestellte Rundungsmodus in Richtung negativ unendlich rundet, ist `rint()` identisch zu `floor()`.
In dieser Version ist der Rundungsmodus fest auf Richtung positiv unendlich eingestellt.

Returnwert

ganze Zahl	dargestellt als Zahl vom Typ `double`, `float` bzw. `long double` bei Erfolg.
`HUGE_VAL`	bei Überlauf. `errno` wird auf `ERANGE` gesetzt, um den Fehler anzuzeigen.

Hinweis

In dieser Version ist der Rundungsmodus fest auf Richtung positiv unendlich eingestellt.

Siehe auch

`abs()`, `ceil()`, `floor()`, `llrint()`, `llround()`, `lrint()`, `lround()`, `round()`

rmdir - Dateiverzeichnis löschen

Syntax #include <unistd.h>

int rmdir(const char **path*);

Beschreibung

`rmdir()` löscht ein Dateiverzeichnis, dessen Name durch *path* angegeben wird. Das Dateiverzeichnis wird nur dann gelöscht, wenn es ein leeres Dateiverzeichnis ist.

Wenn *path* ein symbolischer Verweis ist, wird ihm nicht gefolgt.

Wenn *path* das Root-Verzeichnis ist, wird *path* auf `EBUSY` gesetzt. Wenn *path* das aktuelle Dateiverzeichnis eines aktiven Prozesses ist, ist das Verhalten von `rmdir()` nicht spezifiziert.

Wenn der Verweiszähler des Dateiverzeichnisses gleich 0 wird und kein Prozess das Dateiverzeichnis geöffnet hat, wird der vom Dateiverzeichnis belegte Speicher freigegeben. Auf das Dateiverzeichnis kann nicht länger zugegriffen werden. Wenn ein oder mehrere Prozesse das Dateiverzeichnis geöffnet haben, während der letzte Verweis entfernt wird, werden die Einträge `.` und `..` entfernt, bevor `rmdir()` zurückkehrt. Es können keine neuen Einträge mehr in diesem Dateiverzeichnis vorgenommen werden; das Dateiverzeichnis wird jedoch erst dann entfernt, wenn alle Verweise auf das Dateiverzeichnis geschlossen worden sind.

Bei erfolgreicher Beendigung kennzeichnet `rmdir()` die Felder `st_ctime` und `st_mtime` des übergeordneten Dateiverzeichnisses zur Aktualisierung.

Returnwert

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

Fehler `rmdir()` schlägt fehl, wenn gilt:

`EACCES`	Für eine Komponente des Pfades ist kein Suchrecht vorhanden, oder das Schreibrecht für das übergeordnete Dateiverzeichnis des zu löschenden Dateiverzeichnisses ist nicht vorhanden.
`EBUSY`	Das zu entfernende Dateiverzeichnis ist das aktuelle Dateiverzeichnis des Systems.
`EEXIST` oder `ENOTEMPTY`	*path* bezeichnet ein Dateiverzeichnis, das nicht leer ist.
Erweiterung `EFAULT`	*path* weist über den zugewiesenen Adressraum des Prozesses hinaus.
`EINVAL`	Das Verzeichnis, das entfernt werden soll, ist das aktuelle Dateiverzeichnis.

`EIO` Ein Ein-/Ausgabe-Fehler ist während des Zugriffs auf das Dateisystem aufgetreten.

`ELOOP` Bei der Übersetzung von *path* wurden zuviele symbolische Verweise angetroffen. ❑

`ENAMETOOLONG`
Die Länge von *path* überschreitet `{PATH_MAX}` oder eine Pfadnamenkomponente ist länger als `{NAME_MAX}`, und `{_POSIX_NO_TRUNC}` ist aktiv.

`ENOENT` *path* bezeichnet ein nicht-existierendes Dateiverzeichnis oder zeigt auf eine leere Zeichenkette.

`ENOTDIR` Eine Komponente des Pfades ist kein Dateiverzeichnis. ❑

`EROFS` Der zu löschende Dateiverzeichniseintrag befindet sich in einem schreibgeschützten Dateisystem.

Hinweis `rmdir()` wird nur für POSIX-Dateien ausgeführt.

Siehe auch `mkdir()`, `remove()`, `unlink()`, `unistd.h`.

round, roundf, roundl - auf nächste ganze Zahl runden

Syntax #include <math.h>

double round(double *x*);

float roundf (float *x*);

long double roundl (long double *x*);

Beschreibung

Die Funktionen geben in Gleitpunktdarstellung jeweils die ganze Zahl zurück, die *x* am nächsten liegt.
`round()` stellt das Ergebnis dar als Zahl vom Typ `double`, `roundf()` als Zahl vom Typ `float` und `roundl()` als Zahl vom Typ `long double`.

Der zurückgegebene Wert ist unabhängig vom eingestellten Rundungsmodus. Wenn die Differenz zwischen *x* und dem gerundeten Ergebnis genau 0.5 ist, wird die betragsmäßig größere ganze Zahl zurückgegeben.

Returnwert

ganze Zahl	dargestellt als Zahl vom Typ `double`, `float` bzw. `long double` bei Erfolg.
undefiniert	bei Über- oder Unterlauf. `errno` wird auf `ERANGE` gesetzt, um den Fehler anzuzeigen.

Siehe auch `abs()`, `ceil()`, `floor()`, `llrint()`, `llround()`, `lrint()`, `lround()`, `rint()`

sbrk - Größe des Datensegments verändern

Syntax #include <unistd.h>

void *sbrk(int *incr*);

Beschreibung

Siehe `brk()`.

scanf - formatiert aus Standard-Eingabestrom lesen

Syntax #include <stdio.h>

int scanf(const char **format*[, *arglist*]);

Beschreibung

Siehe `fscanf()`.

scalb - laden Exponent einer basisunabhängigen Gleitpunktzahl

Syntax #include <math.h>

double scalb (double *x*, double *n*);

Beschreibung

`scalb()` berechnet $x*r^n$, wobei *r* die Basis der maschinenabhängigen Gleitpunkt-Arithmetik ist. Für *r*=2 ist `scalb()` äquivalent mit `ldexp()`.

Returnwert

$x*r^n$	bei erfolgreicher Ausführung von `scalb()`.
`+-HUGE_VAL`	je nach Vorzeichen von *x*, wenn `scalb()` einen Überlauf verursacht. `errno` wird auf `ERANGE` gesetzt
`0`	wenn `scalb()` einen Unterlauf verursacht. `errno` wird auf `ERANGE` gesetzt.

Fehler `scalb()` schlägt fehl, wenn gilt:

`ERANGE`	`scalb()` verursacht einen Über- oder Unterlauf.

Hinweis Eine Anwendung, die die Fehlersituation abprüfen möchte, sollte `errno` auf 0 setzen, bevor die Funktion `scalb()` aufgerufen wird. Wenn dann bei der Rückkehr `errno` ungleich null ist, wird damit ein Fehler signalisiert.

Für BS2000 ist die Basis *r=16*

Siehe auch `ldexp()`, `math.h`

seed48 - Startwert (int) für Pseudo-Zufallszahlen setzen

Syntax #include <stdlib.h>

unsigned short int *seed48 (unsigned short int *seed16v*[3]);

Beschreibung

Siehe `drand48()`.

seekdir - Lese-/Schreibzeiger in Dateiverzeichnisstrom positionieren

Syntax #include <dirent.h>

Optional
#include <sys/types.h> ❑

void seekdir(DIR **dirp*, long int *loc*);

Beschreibung

`seekdir()` setzt die Position für die nächste Operation `readdir()` im Dateiverzeichnisstrom, auf den *dirp* zeigt, auf die durch *loc* angegebene Position. Der Wert von *loc* sollte von einem vorangegangenen Aufruf von `telldir()` zurückgeliefert worden sein. Die neue Position geht an die Position des Dateiverzeichnisstroms zurück, die diesem zu dem Zeitpunkt zugeordnet war, als die Operation `telldir()` ausgeführt wurde.

Erweiterung
Die von `telldir()` zurückgegebenen Werte sind nur dann richtig, wenn das Dateiverzeichnis nicht infolge von Verdichtung oder Erweiterung verändert wurde. Dies ist kein Problem bei System V, kann jedoch bei einigen Dateisystemen problematisch sein. ❑

Fehler `seekdir()` schlägt fehl, wenn gilt:

Erweiterung

`EBADF` Der dem Dateiverzeichnis zugeordnete Strom ist nicht mehr gültig. Dieser Fehler entsteht, wenn das Dateiverzeichnis geschlossen wurde. ❑

Hinweis `seekdir()` wird nur für POSIX-Dateien ausgeführt.

Siehe auch `opendir()`, `readdir()`, `telldir()`, `dirent.h`, `sys/types.h`

select - synchrones I/O Multiplexen

Syntax

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

int select ( int nfds, fd_set *readfds, fd_set *writefds,
             fd_set *execptfds, struct timeval *timeout);

void FD_CLR(int fd, fd_set *fdset);

int FD_ISSET(int fd, fd_set *fdset);

void FD_SET(int fd, fd_set *fdset);

void FD_ZERO(fd_set *fdset);
```

Beschreibung

`select` überprüft die E/A-Deskriptormengen, die in *readfds*, *writefds* und *execptfds* übergeben werden, um zu sehen, ob einer ihrer Deskriptoren bereit fürs Lesen oder fürs Schreiben ist, oder eine noch nicht ausgewertete Ausnahmebedingung besitzt. *nfds* ist die Anzahl der Bits, die in jeder Bitmaske überprüft werden sollen, die eine Dateideskriptormenge darstellt. Die Deskriptoren der Deskriptormengen werden von 0 bis *nfds*-1 überprüft. Beim Rücksprung ersetzt `select` die gegebene Deskriptormenge durch Untermengen, die aus den Deskriptoren bestehen, die für die gewünschte Operation bereit sind. Der Rückgabewert von dem `select()`-Aufruf ist die Anzahl der Deskriptoren, die bereit sind.

Die Deskriptormengen werden als Bitfelder in aufsteigender Reihenfolge abgespeichert. Für die Manipulation solcher Deskriptormengen stehen folgende Makros zur Verfügung:

`FD_ZERO`(&*fdset*)	initialisiert eine Deskriptormenge *fdset* mit der Nullmenge.
`FD_SET`(*fd*,&*fdset*)	fügt einen Deskriptor *fd* in *fdset* ein.
`FD_CLR`(*fd*,&*fdset*)	entfernt *fd* aus *fdset*
`FD_ISSET`(*fd*,&*fdset*)	ist ungleich null, wenn *fd* ein Element aus *fdset* ist, ansonsten ist es null.

Das Verhalten dieser Makros ist nicht definiert, falls ein Deskriptorwert kleiner als null oder größer/gleich `FD_SETSIZE` ist. `FD_SETSIZE` ist eine Konstante, die in `sys/select.h` definiert ist, und normalerweise mindestens genauso groß ist wie die maximale Anzahl der Deskriptoren, die vom System zur Verfügung stehen.

Falls *timeout* kein Nullzeiger ist, gibt es eine maximale Zeit an, die gewartet werden soll, bis die Auswahl vollständig ist. Falls *timeout* ein Nullzeiger ist, blockiert das `select` bis eines der abgefragten Ereignisse eintritt. `select` blockiert dann nicht, wenn eine Struktur übergeben wird, die nur Null-Werte enthält. *readfds*, *writefds* und *execptfds* können als Nullzeiger gegeben sein, wenn keine der Deskriptoren von Interesse ist.

Returnwert

Anzahl	bereite Deskriptoren in den Deskriptorenmengen
-1	bei Fehler
0	falls die Zeitgrenze überschritten wurde

Fehler Eine Fehlerrückgabe von `select` kann sein:

`EBADF`	Eine der E/A-Deskriptormengen besitzt einen ungültigen E/A-Deskriptor.
`EINTR`	Es wurde ein Signal gegeben, bevor eines der gewünschten Ereignisse eingetreten ist, oder die Zeitgrenze überschritten wurde.
`EINVAL`	Eine Komponente der Zeitgrenze, die referenziert wird, liegt außerhalb des erlaubten Bereichs: *t_sec* muss zwischen 0 und 10, inklusive liegen. *t_usec* muss größer oder gleich 0 und kleiner als 10 sein.

Hinweis Der Standardwert für `FD_SETSIZE` (augenblicklich 2048) ist gleich der Standardgrenze der Anzahl geöffneter Dateien. Um Programme anzupassen, die eine größere Anzahl geöffneter Dateien mit `select` verwenden, ist es möglich, diese Größe innerhalb eines Programms zu erhöhen, indem man einen größeren Wert für `FD_SETSIZE` definiert, bevor man `<sys/types.h>` einschließt.

In zukünftigen Versionen des Systems könnte `select` die verbliebene Zeit des ursprünglichen Zeitlimits (wenn einer existiert) zurückliefern, indem der Zeitwert an der richtigen Stelle geändert wird. Es ist deshalb nicht ratsam vorauszusetzen, dass der Wert des Zeitlimits durch den `select` -Aufruf unverändert bleibt.

Die Deskriptormengen sind bei der Rückkehr immer verändert, sogar wenn der Aufruf als Ergebnis eines Zeitlimits zurückkehrt.

Siehe auch `poll()`, `read()`, `write()`.

semctl - Semaphor-Steueroperationen anwenden

Syntax #include <sys/sem.h>

int semctl(int *semid*, int *semnum*, int *cmd*, ...);

Beschreibung

`semctl()` bietet eine Vielzahl von Operationen für die Semaphor-Steuerung.

Mit *cmd* werden die im Folgenden aufgeführten Semaphor-Steueroperationen angegeben, mit *semid* und *semnum* das Semaphor, für das die angegebene Operation ausgeführt werden soll. Die für die jeweilige Operation erforderlichen Zugriffsrechte werden bei den entsprechenden Kommandos angegeben (siehe auch Abschnitt „Interprozesskommunikation“ auf Seite 116). Die symbolischen Namen für die Werte von *cmd* sind in der Include-Datei `sys/sem.h` definiert:

`GETVAL` Wert von `semval` liefern (siehe auch `sys/sem.h`). Leserecht erforderlich.

`SETVAL` Wert von `semval` auf den Wert des vierten Arguments vom Typ `int` setzen. Nach erfolgreicher Durchführung von *cmd* ist der dem angegebenen Semaphor entsprechende `semadj`-Wert in allen Prozessen gelöscht. Änderungsberechtigung erforderlich (siehe auch Abschnitt „Interprozesskommunikation“ auf Seite 116).

`GETPID` Wert von `sempid` liefern. Leserecht erforderlich.

`GETNCNT` Wert von `semncnt` liefern. Leserecht erforderlich.

`GETZCNT` Wert von `semzcnt` liefern. Leserecht erforderlich.

Folgende Kommandos wirken auf jeden `semval` aus der Menge der zulässigen Semaphore:

`GETALL` Wert von `semval` zurückgeben und in den Vektor eintragen, auf den *arg.array* zeigt. Leserecht erforderlich.

`SETALL` `semval` auf den Wert des Vektors vom Typ unsigned short setzen, auf das das vierte Argument von `semctl()` zeigt. Nach erfolgreicher Ausführung dieses Kommandos sind die den angegebenen Semaphoren entsprechenden `semadj`-Werte in allen Prozessen gelöscht. Änderungsberechtigung erforderlich.

Folgende Kommandos sind außerdem verfügbar:

`IPC_STAT` den aktuellen Wert jedes Elements der `semid_ds` Datenstruktur, die zu *semid* gehört, in die `semid_ds` Struktur schreiben, auf die das vierte Argument von `semctl()` zeigt.

`IPC_SET` den Wert der folgenden Elemente der `semid_ds` Datenstruktur, die zu *semid* gehört, auf den entsprechenden Wert setzen, der in der `semid_ds` Struktur gefunden wurde, auf die das vierte Argument von `semctl()` zeigt:

```
sem_perm.uid
sem_perm.gid
sem_perm.mode /* nur 9 niederwertige Bits */
```

Dieses Kommando kann nur von einem Prozess ausgeführt werden, dessen effektive Benutzernummer die eines Prozesses mit Sonderrechten ist oder mit `sem_perm.cuid` oder `sem_perm.uid` in der *semid* zugeordneten Datenstruktur übereinstimmt.

`IPC_RMID` Die mit *semid* angegebene Semaphorkennzahl im System sowie die Semaphorenmenge samt zugeordneter Datenstruktur löschen. Dieses Kommando kann nur von einem Prozess ausgeführt werden, dessen effektive Benutzernummer die eines Prozesses mit Sonderrechten ist oder mit `sem_perm.cuid` oder `sem_perm.uid` in der *semid* zugeordneten Datenstruktur übereinstimmt.

Returnwert Bei Erfolg liefert `semctl()` je nach *cmd*-Wert einen der folgenden Returnwerte:

Wert von `semval`
: wenn für *cmd* `GETVAL` angegeben wurde.

Wert von `sempid`
: wenn für *cmd* `GETVAL` angegeben wurde.

Wert von `semncnt`
: wenn für *cmd* `GETVAL` angegeben wurde.

Wert von `semzcnt`
: wenn für *cmd* `GETVAL` angegeben wurde.

0 wenn andere *cmd*-Werte angegeben wurden.

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

Fehler `semctl()` schlägt fehl, wenn gilt:

`EACCES` Der aufrufende Prozess hat für das auszuführende Kommando nicht die erforderliche Zugriffsrecht (siehe Abschnitt „Interprozesskommunikation“ auf Seite 116).

Erweiterung
`EFAULT` *msgp* verweist auf eine unzulässige Adresse. ❑

`EINVAL` *semid* ist keine gültige Semaphorkennzahl, *semnum* hat einen Wert kleiner 0 oder größer `sem_nsems` oder *cmd* ist kein gültiges Kommando.

EPERM — *cmd* ist gleich IPC_RMID oder IPC_SET, die effektive Benutzernummer des aufrufenden Prozesses ist nicht die eines Prozesses mit Sonderrechten und stimmt nicht mit sem_perm.cuid oder sem_perm.uid in der *semid* zugeordneten Datenstruktur überein.

ERANGE — *cmd* ist gleich SETVAL oder SETALL und der Wert, auf den semval gesetzt werden soll, ist größer als der im System zulässige Höchstwert.

Hinweis

Das vierte Argument im Abschnitt „Syntax“ ist im XPG4 mit ... gekennzeichnet, um einen Widerspruch zum ISO C-Standard zu vermeiden. Der vierte Parameter kann vom Anwendungsprogrammierer wie folgt definiert werden:

```
union semun
{  int val;
   struct semid_ds *buf;
   unsigned short *array;
} arg;
```

Das Kommittee des IEEE 1003.4-Standards entwickelt gerade eine alternative Schnittstelle für die Interprozeßkommunikation. Anwendungsprogrammierer, die Interprozeßkommunikation einsetzen, sollten die Anwendungen so konzipieren, daß Module, die derzeit beschriebene Funktionen für Interprozeßkommunikation benutzen, einfach geändert werden können.

Siehe auch semget(), semop(), sys/sem.h, Abschnitt „Interprozesskommunikation“ auf Seite 116.

semget - Semaphorkennzahl ermitteln

Syntax #include <sys/sem.h>

int semget(key_t *key*, int *nsems*, int *semflg*);

Beschreibung

`semget()` richtet eine Semaphorkennzahl mit der dazugehörigen Datenstruktur `semid_ds` und der dazugehörigen Menge von *nsems* Semaphoren (siehe `sys/sem.h`) für das Argument *key* ein, wenn eine der folgenden Bedingungen zutrifft:

– *key* hat den Wert `IPC_PRIVATE`.

– Für *key* wurde noch keine Semaphorkennzahl eingerichtet und (*semflg* & `IPC_CREAT`) ist ungleich 0.

Beim Einrichten der neuen Semaphorkennzahl *key* wird die dazugehörige Datenstruktur `semid_ds` wie folgt initialisiert:

– Für die Strukturkomponenten `sem_perm.cuid`, `sem_perm.uid`, `sem_perm.cgid` und `sem_perm.gid` werden die effektive Benutzernummer und die effektive Gruppennummer des aufrufenden Prozesses eingetragen.

– In die 9 niederwertigen Bits von `sem_perm.mode` werden die 9 niederwertigen Bits von *semflg* eingetragen.

– `sem_nsems` gleich dem Wert von *nsems* gesetzt.

– `sem_otime` wird gleich 0 und `sem_ctime` gleich der aktuellen Zeit gesetzt.

– Die den einzelnen Semaphoren zugeordneten Datenstrukturen werden nicht initialisiert. `setctl()` kann mit den Kommandos `SETVAL` oder `SETALL` dazu verwendet werden, die einzelnen Semaphoren zu initialisieren.

Returnwert

Semaphorkennzahl	bei Erfolg. Die Semaphorkennzahl ist eine nichtnegative ganze Zahl.
-1	bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `semget()` schlägt fehl, wenn gilt:

`EACCES`	Für *key* existiert bereits eine Semaphorkennzahl, aber die in den 9 niederwertigen Bits von *semflg* angegebene Berechtigung wurde nicht erteilt.
`EEXIST`	Für *key* existiert eine Semaphorkennzahl, aber `((`*semflg* & `IPC_CREAT) && (`*semflg* & `IPC_EXCL))` ist ungleich 0.

`EINVAL`	Der Wert von *nsems* ist entweder kleiner gleich 0 oder größer als der vom System festgelegte Maximalwert, oder es existiert bereits eine Semaphorkennzahl *key*, deren zugehörige Semaphorenmenge weniger als *nsems* Semaphore enthält und *nsems* ist ungleich 0.
`ENOENT`	Für *key* existiert keine Semaphorkennzahl und `(`*semflg* `& IPC_CREAT)` ist gleich 0.
`ENOSPC`	Es soll eine Semaphorkennzahl eingerichtet werden; aber dadurch wird die Maximalzahl der Semaphoren überschritten, die im System zulässig sind.

Hinweis Das Kommittee des IEEE 1003.4-Standards entwickelt gerade eine alternative Schnittstelle für die Interprozeßkommunikation. Anwendungsprogrammierer, die Interprozeßkommunikation einsetzen, sollten die Anwendungen so konzipieren, daß Module, die derzeit beschriebene Funktionen für Interprozeßkommunikation benutzen, einfach geändert werden können.

Siehe auch `semctl()`, `semop()`, `sys/sem.h`, Abschnitt „Interprozesskommunikation“ auf Seite 116.

semop - Semaphor-Operationen durchführen

Syntax #include <sys/sem.h>

int semop(int *semid*, struct sembuf **sops*, size_t *nsops*);

Beschreibung

`semop()` ermöglicht die automatische Ausführung einer Liste mit vom Benutzer definierten Semaphoroperationen bezogen auf die Semaphormenge mit der im Argument *semid* angegebenen Semaphorkennzahl.

sops zeigt auf einen benutzerdefinierten Vektor von Strukturen für Semaphoroperationen.

nsops gibt die Anzahl der Strukturen im Vektor an.

Jede `sembuf`-Struktur enthält folgende Komponenten:

Datentyp	Komponentennamen	Beschreibung
`short`	`sem_num`	Semaphornummer
`short`	`sem_op`	Semaphoroperation
`short`	`sem_flg`	Operationsschalter

Jede mit `sem_op` definierte Semaphoroperation wird für das mit *semid* und *sem_num* angegebene Semaphor ausgeführt.

`sem_op` definiert eine der folgenden drei Semaphoroperationen:

1. Wenn `sem_op` eine negative ganze Zahl ist und der aufrufende Prozess das Schreibrecht besitzt, so tritt einer der folgenden Fälle ein:
   - Wenn `semval` größer oder gleich dem Absolutbetrag von `sem_op` ist, wird der Absolutbetrag von `sem_op` von `semval` subtrahiert.
   - Wenn (`sem_flg & SEM_UNDO`) ungleich 0 ist, wird der Absolutbetrag von `sem_op` zum `semadj`-Wert des aufrufenden Prozesses für das angegebene Semaphor addiert (siehe `exit()`).
   - Wenn `semval` kleiner als der Absolutbetrag von `sem_op` und `(sem_flg & IPC_NOWAIT)` ungleich 0 ist, kehrt `semop()` sofort zurück.
   - Wenn `semval` kleiner als der Absolutbetrag von `sem_op` und `(sem_flg & IPC_NOWAIT)` gleich 0 ist, erhöht `semop()` den Wert von `semncnt` des angegebenen Semaphors um 1 und der aufrufende Prozess wird angehalten, bis eine der folgenden Bedingungen eintritt:
     - Der Wert von `semval` wird größer oder gleich dem Absolutbetrag von `sem_op`. Wenn dies eintritt, wird der `semncnt`-Wert des angegebenen Semaphors um 1 vermindert, der Absolutbetrag von `sem_op` wird von `semval` subtrahiert und,

wenn (sem_flg & SEM_UNDO) ungleich 0 ist, wird der Absolutbetrag von sem_op zum semadj-Wert des aufrufenden Prozesses für das angegebene Semaphor addiert.

- Die Kennzahl semid, für die der aufrufende Prozess auf eine Operation wartet, wird im System gelöscht (siehe semctl()). In diesem Fall wird errno auf EIDRM gesetzt und der Wert -1 zurückgeliefert.
- Der aufrufende Prozess empfängt ein abzufangendes Signal. In diesem Fall wird der semncnt-Wert des angegebenen Semaphors um 1 vermindert und der aufrufende Prozess setzt seine Ausführung in der Weise fort, wie dies bei der Funktion sigaction() beschrieben ist.

2. Wenn sem_op eine positive ganze Zahl ist und der aufrufende Prozess das Schreibrecht besitzt, wird der Wert von sem_op zum semval-Wert addiert, und, wenn (sem_flg & SEM_UNDO) ungleich 0 ist, vom semadj-Wert des aufrufenden Prozesses für das angegebene Semaphor subtrahiert.
3. Wenn sem_op gleich 0 ist und der aufrufende Prozess das Leserecht besitzt, so tritt einer der folgenden Fälle ein:
   - Wenn semval gleich 0 ist, kehrt semop() sofort zurück.
   - Wenn sowohl semval als auch (sem_flg & IPC_NOWAIT) ungleich 0 sind, kehrt semop() sofort zurück.
   - Wenn semval ungleich 0 und (sem_flg & IPC_NOWAIT) gleich 0 sind, erhöht semop() den Wert von semzcnt des angegebenen Semaphors um 1 und der aufrufende Prozess wird angehalten, bis eines der folgenden Ereignisse eintritt:
     - semval nimmt den Wert 0 an. Dann wird der Wert von semzcnt des angegebenen Semaphors um 1 vermindert.
     - Die Kennzahl semid des Semaphors, für das der aufrufende Prozess auf eine Operation wartet, wird im System gelöscht. In diesem Fall wird errno auf EIDRM gesetzt und der Wert -1 zurückgeliefert.
     - Der aufrufende Prozess empfängt ein abzufangendes Signal. In diesem Fall wird der Wert von semzcnt des angegebenen Semaphors um 1 vermindert und der aufrufende Prozess setzt seine Ausführung in der Weise fort, wie dies bei sigaction() beschrieben ist.

Bei erfolgreicher Ausführung wird der Wert von sempid für alle in dem Vektor, auf den sops zeigt, enthaltenen Semaphore gleich der Prozessnummer des aufrufenden Prozesses gesetzt.

Bei der Verwendung von Threads kommt ändert sich die Funktionalität von semop in folgenden Punkten:

Durchführen von Semaphor-Operationen

zu 1. Wenn *semval* kleiner als der Absolutbetrag von *sem_op* und (*sem_flg* & IPC_NOWAIT) gleich 0 ist, erhöht semop() den Wert von *semncnt* des angegebenen Semaphors um 1 und der aufrufende Thread wird angehalten, bis eine der folgenden Bedingungen eintritt:

- Der Wert von *semval* wird größer oder gleich dem Absolutbetrag von *sem_op*. Wenn dies eintritt, wird der *semncnt*-Wert des angegebenen Semaphors um 1 vermindert, der Absolutbetrag von *sem_op* wird von *semval* subtrahiert und wenn (*sem_flg* & SEM_UNDO) ungleich 0 ist, wird der Absolutbetrag von *sem_op* zum *semadj*-Wert des aufrufenden Prozesses für das angegebene Semaphor addiert.
- Die Kennzahl *semid*, für die der aufrufende Thread auf eine Operation wartet, wird im System gelöscht. In diesem Fall wird errno auf EIDRM gesetzt und der Wert -1 zurückgeliefert.
- Der aufrufende Thread empfängt ein abzufangendes Signal. In diesem Fall wird der *semncnt*-Wert des angegebenen Semaphors um 1 vermindert und der aufrufende Thread setzt seine Ausführung in der Weise fort, wie dies bei der Funktion sigaction() beschrieben ist.

zu 3. Wenn *semval* ungleich 0 und (*sem_flg* & IPC_NOWAIT) gleich 0 sind, erhöht semop() den Wert von *semzcnt* des angegebenen Semaphors um 1 und der aufrufende Thread wird angehalten, bis eines der folgenden Ereignisse eintritt:

- *semval* nimmt den Wert 0 an. Dann wird der Wert von *semzcnt* des angegebenen Semaphors um 1 vermindert.
- Die Kennzahl *semid* des Semaphors, für das der aufrufende Thread auf eine Operation wartet, wird im System gelöscht. In diesem Fall wird errno auf EIDRM gesetzt und der Wert -1 zurückgeliefert.

Der aufrufende Thread empfängt ein abzufangendes Signal. In diesem Fall wird der Wert von *semzcnt* des angegebenen Semaphors um 1 vermindert und der aufrufende Thread setzt seine Ausführung in der Weise fort, wie dies bei sigaction() beschrieben ist.

Returnwert

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

Fehler semop() schlägt fehl, wenn gilt:

E2BIG	Der Wert von *nsops* ist größer als der systemspezifische Maximalwert.
EACCES	Der Prozess hat für das auszuführende Kommando nicht die erforderlichen Zugriffsrechte (siehe Abschnitt „Fehlerbehandlung“ auf Seite 130).

EAGAIN	Diese Operation würde dazu führen, dass der aufrufende Prozess angehalten wird, obwohl (sem_flg & IPC_NOWAIT) ungleich 0 ist.
EFBIG	Der Wert von sem_num ist kleiner als 0 oder größer gleich der Anzahl von Semaphoren in der mit *semid* bezeichneten Semaphorenmenge.
EIDRM	Die Semaphorkennzahl *semid* wurde im System gelöscht.
EINTR	semop() wurde durch ein Signal unterbrochen.
EINVAL	Der Wert von *semid* ist keine gültige Semaphorkennzahl oder die Anzahl der einzelnen Semaphore, für die der aufrufende Prozess ein SEM_UNDO anfordert, würde den systemspezifischen Maximalwert überschreiten.
ENOSPC	Die systemspezifische Maximalanzahl von Prozessen, die SEM_UNDO anfordern dürfen, würde überschritten.
ERANGE	Eine Operation würde dazu führen, dass semval oder semadj den systemspezifischen Maximalwert überschreitet.

Hinweis Das Kommittee des IEEE 1003.4-Standards entwickelt gerade eine alternative Schnittstelle für die Interprozeßkommunikation. Anwendungsprogrammierer, die Interprozeßkommunikation einsetzen, sollten die Anwendungen so konzipieren, daß Module, die derzeit beschriebene Funktionen für Interprozeßkommunikation benutzen, einfach geändert werden können.

Siehe auch exec, exit(), fork(), semctl(), semget(), sys/sem.h, Abschnitt „Interprozesskommunikation“ auf Seite 116.

setbuf - Puffer einem Datenstrom zuweisen

Syntax

```
#include <stdio.h>

void setbuf(FILE *stream, char *buf);
```

Beschreibung

`setbuf()` kann verwendet werden, nachdem der Datenstrom, auf den *stream* zeigt, einer offenen Datei zugeordnet wurde, aber bevor eine andere Operation auf dem Datenstrom ausgeführt wird. `setbuf()` bewirkt, dass das Feld, auf das *buf* zeigt, an Stelle eines automatisch zugewiesenen Puffers verwendet wird.

Die Puffergröße ist nicht begrenzt; die Konstante `BUFSIZ` (siehe `stdio.h`) bezeichnet jedoch eine geeignete Puffergröße:

```
char buf[BUFSIZ];
```

Wenn *buf* kein Nullzeiger ist, sind folgende Funktionsaufrufe äquivalent:

```
setbuf(stream, buf)
setvbuf(stream, buf, _IOFBF, BUFSIZ)
```

Wenn *buf* ein Nullzeiger ist, sind Eingabe und Ausgabe ungepuffert und folgende Funktionsaufrufe äquivalent:

```
setbuf(stream, buf)
setvbuf(stream, buf, _IONBF, BUFSIZ)
```

BS2000
Wenn *buf* ein Nullzeiger ist, wird der vom System zugewiesene Puffer verwendet. ❑

Im Unterschied zu `setvbuf()` hat `setbuf()` keinen Returnwert.

Hinweis

Eine häufige Fehlerquelle besteht darin, dass als Puffer in einem Programmblock eine Variable der Speicherklasse `auto` verwendet und die Datei in diesem Block dann nicht geschlossen wird.

Teile von *buf* werden für interne Verwaltungsinformationen des Streams benötigt; deswegen enthält *buf* weniger als *size* Bytes, wenn er voll ist. Sie sollten bei der Verwendung von `setvbuf()` automatisch zugewiesene Puffer verwenden.

`setbuf()` wird für die Datei ausgeführt, die *stream* zugeordnet ist. Dies kann eine POSIX- oder BS2000-Datei sein.

BS2000
Wird der Blockungsfaktor mit dem `BUFFER-LENGTH`-Parameter des `ADD-FILE-LINK`-Kommandos explizit vereinbart, muss die Größe des Bereichs dieser vereinbarten Blockungsgröße entsprechen. ❑

Siehe auch `fopen()`, `setvbuf()`, `stdio.h`, Abschnitt „Datenströme“ auf Seite 76.

setcontext - Benutzerkontext ändern

Syntax #include <ucontext.h>

int setcontext(const ucontext_t **ucp*);

Beschreibung

siehe `getcontext()`

setgid - Gruppennummer eines Prozesses setzen

Syntax #include <unistd.h>

Optional
#include <sys/types.h> ❑

int setgid(gid_t *gid*);

Beschreibung

Wenn der Prozess Sonderrechte hat, setzt `setgid()` die reale, die effektive und die gesicherte Gruppennummer gleich *gid*.

Wenn der Prozess keine Sonderrechte hat, aber *gid* gleich der realen oder der gesicherten Gruppennummer ist, dann setzt `setgid()` die effektive Gruppennummer gleich *gid*, während die reale und gesicherte Gruppenummer unverändert bleiben.
Vorhandene zusätzliche Gruppennummern des aufrufenden Prozesses bleiben unverändert.

Returnwert

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

Fehler `setgid()` schlägt fehl, wenn gilt:

`EINVAL`	Der Wert *gid* ist ungültig und wird nicht unterstützt.
`EPERM`	Der Prozess besitzt keine Sonderrechte und *gid* entspricht weder der realen noch der gesicherten Gruppennummer.

Hinweis Beim Login werden die reale Benutzernummer, die effektive Benutzernummer und die gesicherte Benutzernummer des Login-Prozesses auf die Benutzernummer des Benutzers gesetzt, der für die Erzeugung des Prozesses verantwortlich ist. Dasselbe gilt für die reale, effektive und gesicherte Gruppennummer; sie werden auf die Gruppennummer des Benutzers gesetzt, der für die Erzeugung des Prozesses verantwortlich ist.

Wenn ein Prozess `exec()` aufruft, um eine Datei auszuführen, können sich die Benutzer- und/oder Gruppennummern, die mit dem Prozess verbunden sind, ändern. Wenn die ausgeführte Datei eine 'set-user-ID'-Datei ist, werden die effektive und gesicherte Benutzernummer des Prozesses auf den Benutzer der ausgeführten Datei gesetzt. Wenn die ausgeführte Datei eine 'set-group-ID'-Datei ist, werden die effektive und gesicherte Gruppennummer des Prozesses auf die Gruppe der ausgeführten Datei gesetzt. Wenn die Datei keine 'set-user-ID'- oder 'set-group-ID'-Datei ist, werden die effektive Benutzernummer, die gesicherte Benutzernummer, die effektive Gruppennummer und die gesicherte Gruppennummer nicht verändert.

Siehe auch `exec`, `getgid()`, `setuid()`, `sys/types.h`, `unistd.h`.

setgrent - Schreib-/Lesezeiger auf den Anfang der Gruppendatei zurücksetzen

Syntax #include <grp.h>

void setgrent (void);

Beschreibung

siehe `endgrent()`.

setitimer - Intervall-Timer setzen

Syntax #include <sys/time.h>

int setitimer(int *which*, const struct itimerval **value*, struct itimerval **ovalue*);

Beschreibung

siehe `getitimer()`.

_setjmp - Marke für nichtlokalen Sprung setzen (ohne Signalmaske)

Syntax #include <setjmp.h>

int _setjmp(jmp_buf *env*);

Beschreibung

siehe `_longjmp()`.

setjmp - Marke für nichtlokalen Sprung setzen

Syntax #include <setjmp.h>

int setjmp(jmp_buf *env*);

Beschreibung

`setjmp()` sichert die aktuelle Aufrufumgebung (Adresse im C-Laufzeitstack, Befehlszähler, Registerinhalte) im Argument *env* für eine spätere Verwendung durch die Funktion `longjmp()`. Im POSIX-Subsystem ist `setjmp()` als Makro implementiert. In anderen X/Open-konformen Systemen kann `setjmp()` als Funktion implementiert sein.

Wenn eine Makrodefinition unterdrückt wird, um auf eine vorhandene Funktion zugreifen zu können oder ein Programm oder einen externen Bezeichner mit dem Namen `setjmp` definiert, ist das Verhalten undefiniert.

`setjmp()` ist nur zusammen mit der Funktion `longjmp()` sinnvoll: Mit diesen beiden Funktionen lassen sich nichtlokale Sprünge realisieren, d.h. Sprünge von einer beliebigen Funktion in eine andere, noch aktive Funktion. Ein `longjmp`-Aufruf richtet die von `setjmp()` gespeicherte Aufrufumgebung wieder ein und setzt die Programmausführung anschließend fort (siehe auch `longjmp()`).

env ist das Feld, in das `setjmp()` den aktuellen Programmzustand speichert. Der Typ `jmp_buf` ist in `setjmp.h` definiert.

Alle zugreifbaren Objekte besitzen die Werte, die sie zum Zeitpunkt des Aufrufs von `longjmp()` besaßen, mit Ausnahme der Werte von automatischen Objekten. Diese sind unter folgenden Bedingungen undefiniert:

– sie sind lokal zu der Funktion, die den entsprechenden `setjmp`-Aufruf enthält.

– sie sind nicht vom Typ `volatile`.

– sie wurden zwischen dem `setjmp-` und dem `longjmp`-Aufruf geändert.

`setjmp()` sollte nur in folgenden Zusammenhängen aufgerufen werden:

– als vollständiger Bedingungsausdruck einer Auswahl- oder Schleifenanweisung, z.B.

```
if (setjmp(env)) ...
```

– als Operand eines Vergleichsoperators, wobei der andere Operand ein konstanter ganzzahliger Ausdruck und der Gesamtausdruck der vollständige Bedingungsausdruck einer Auswahl- oder Schleifenanweisung ist, z.B.

```
if (setjmp(env) == 0) ...
```

– als Operand des einstelligen Operators !, wobei der Gesamtausdruck der vollständige Bedingungsausdruck einer Auswahl- oder Schleifenanweisung ist, z.B.

```
if (!setjmp(env)) ...
```

– als vollständiger Ausdruck einer Ausdrucksanweisung, ggf. umgewandelt in den Typ

```
void: (void)setjmp(env);
```

Returnwert

0	bei erfolgreicher Rückkehr eines unmittelbaren `sigset`-Aufrufs.
≠ 0	wenn die Rückkehr von einem `longjmp`-Aufruf erfolgt. Der Returnwert entspricht in diesem Falle dem Wert des Arguments *val* des `longjmp`-Aufrufs.

Hinweis Im Allgemeinen ist `sigsetjmp()` geeigneter als `setjmp()` zur Behandlung von Fehlern und Signalen, die in Low-Level-Unterprogrammen auftreten.

Siehe auch `longjmp()`, `sigsetjmp()`, `setjmp.h`

setkey - Codierschlüssel setzen

Syntax #include <stdlib.h>

void setkey(const char **key*);

Beschreibung

`setkey()` ermöglicht den Zugriff auf einen Verschlüsselungsalgorithmus.
key ist ein Zeichenfeld mit einer Länge von 64 Bytes, das nur Zeichen mit den numerischen Werten 0 und 1 enthält. Diese Zeichenkette wird in Gruppen von je acht Bits aufgeteilt; dabei wird das niederwertige Bit in jeder Gruppe ignoriert. Hieraus ergibt sich ein Schlüssel mit 56 Bits, der eingetragen wird. Dies ist dann der Schlüssel, der von dem Algorithmus zum Verschlüsseln der Zeichenkette *block* von der Funktion `encrypt()` verwendet wird.

Hinweis Da `setkey()` keinen Returnwert zurückgibt, können Fehler nur wie folgt festgestellt werden: `errno` wird auf 0 gesetzt; anschließend wird die Funktion aufgerufen und `errno` geprüft. Wenn `errno` ungleich 0 ist, muss ein Fehler aufgetreten sein.

Siehe auch `crypt()`, `encrypt()`, `stdlib.h`.

setlocale - Lokalität ändern oder ermitteln

Syntax #include <locale.h>

char *setlocale(int *category*, const char **locale*);

Beschreibung

`setlocale()` kann den Teil der Lokalität, der durch *category* und *locale* angegeben wird, ändern oder die aktuelle Lokalität ganz oder teilweise ermitteln.

Für *category* können folgende Konstantennamen angegeben werden, die einer Datenbank zugeordnet sind:

`LC_ALL` beeinflusst die gesamte Lokalität (siehe Abschnitt „Lokalität“ auf Seite 52).

BS2000
Die Lokalitätskomponente `LC_MESSAGES` wird bei BS2000-Funktionalität nicht unterstützt (siehe Abschnitt „Umfang der unterstützten C-Bibliothek“ auf Seite 18). ❑

`LC_COLLATE` beeinflusst das Verhalten von regulären Ausdrücken und der Vergleichsfunktionen für Zeichenketten.

`LC_CTYPE` beeinflusst das Verhalten von regulären Ausdrücken und der Funktionen zur Zeichenbearbeitung und der Multibyte-Funktionen.

`LC_MESSAGES` beeinflusst das Format von Meldungs-Zeichenketten.

BS2000
Diese Lokalitätskomponente wird bei BS2000-Funktionalität nicht unterstützt (siehe Abschnitt „Umfang der unterstützten C-Bibliothek“ auf Seite 18). ❑

`LC_MONETARY` beeinflusst die monetären Formatierungsinformationen, die von `localeconv()` zurückgegeben werden.

`LC_NUMERIC` beeinflusst den Dezimalpunkt für die formatierte Ein- und Ausgabe und die Zeichenketten-Umwandlungsfunktionen und nichtmonetäre Formatierungsinformationen, die von `localeconv()` zurückgegeben werden.

`LC_TIME` beeinflusst das Verhalten der Zeit-Umwandlungsfunktionen.

Das Verhalten von `nl_langinfo()` wird ebenfalls durch die Einstellungen von *category* beeinflusst.

locale ist ein Zeiger auf eine Zeichenkette, die die benötigten Einstellungen für *category* enthält. Zusätzlich sind folgende, für *locale* vordefinierten Werte für alle Einstellungen von *category* definiert:

`"POSIX"` spezifiziert die minimale Umgebung für die Programmiersprache C; sie wird **POSIX-Lokalität** genannt. Wenn `setlocale()` nicht aufgerufen wird, ist die POSIX-Lokalität voreingestellt.

`"C"` wird **C-Lokalität** genannt und entspricht `"POSIX"`.

`""` spezifiziert eine sprachabhängige Umgebung, die dem Wert der *category* entsprechenden Umgebungsvariablen `LC_*` bzw. der Umgebungsvariablen `LANG` entspricht.

Nullzeiger wird verwendet, um `setlocale()` anzuweisen, die aktuelle Lokalität abzufragen und deren Namen zurückzugeben.

Werden Threads verwendet, so wirkt sich die Funktion auf den Prozess oder auf einen Thread wie folgt aus: Wenn der Prozess ”multithreaded” ist, wirkt sich die Änderung der Lokalität auf alle Threads des Prozesses aus.

BS2000

`"V1CTYPE"` Im Unterschied zur C-Lokalität gelten die Zeichen `X'8B'`, `X'8C'`, `X'8D'` als Kleinbuchstaben, die Zeichen `X'AB'`, `X'AC'`, `X'AD'` als Großbuchstaben und die Zeichen `X'C0'` und `X'D0'` als Sonderzeichen. In der Lokalität `"C"` gelten all diese Zeichen als Steuerzeichen.

`"V2CTYPE"` Im Unterschied zur C-Lokalität ist die Sortierreihenfolge so eingestellt, dass sie den Werten des EBCDIC-Zeichensatzes entspricht.

`"GERMANY"` Die im deutschen Sprachraum üblichen Konventionen sind festgelegt.

`"De.EDF04F"` Länderspezifische Lokalität, deren Konvertierungstabellen auf ASCII-Code ISO 8859-15 bzw. EBCDIC-Code EDF04F basieren und die in der Kategorie LC_MONETARY die Währung “DM“ unterstützt.

`"De.EDF04F@euro"`
Länderspezifische Lokalität, deren Konvertierungstabellen auf ASCII-Code ISO 8859-15 bzw. EBCDIC-Code EDF04F basieren und die in der Kategorie LC_MONETARY die Währung “Euro“ unterstützt.

Die Zeichenketten sind in der Include-Datei `locale.h` folgendermaßen vordefiniert:

symbolische Konstante	**voreingestellter Wert**
LC_C_C	"POSIX"
LC_C_C	"C"
LC_C_DEFAULT	""
LC_C_V1CTYPE	"V1CTYPE"
LC_C_V2TYPE	"V2CTYPE"
LC_C_GERMANY	"GERMANY"
LC_C_DeEDF04F	"De.EDF04F"
LC_C_DeEDF04F@euro	"De.EDF04F@euro"

❑

Returnwert Zeichenkette, die die aktuelle Lokalität für *category* angibt
wenn *locale* kein Nullzeiger ist und `setlocale()` erfolgreich beendet wurde oder
wenn *locale* ein Nullzeiger ist. Die Lokalität wird dabei nicht verändert.

Nullzeiger wenn `setlocale()` nicht erfolgreich beendet wurde. Die Lokalität wird nicht verändert.

Wenn ein nachfolgender `setlocale`-Aufruf mit der zurückgegebenen Zeichenkette und der zugehörigen Kategorie aufgerufen wird, wird dieser Teil der Lokalität wiederherstellt. Die zurückgelieferte Zeichenkette darf nicht durch das Programm verändert werden, kann aber durch einen nachfolgenden `setlocale`-Aufruf überschrieben werden.

Hinweis Die folgenden Programmanweisungen zeigen, wie ein Programm die Lokalität initialisieren kann, während die Lokalität teilweise verändert wird, so dass reguläre Ausdrücke und Zeichenketten-Operationen auf einen fremdsprachigen Text angewendet werden können:

```
setlocale(LC_ALL, "De");
setlocale(LC_COLLATE, "Fr@dict");
```

Internationalisierte Programme müssen `setlocale()` aufrufen, um Sprachspezifika zu berücksichtigen. Dies kann durch einen Aufruf von `setlocale()` geschehen, der wie folgt aussieht:

```
setlocale (LC_ALL, "");
```

Dieser Aufruf verwendet die Einstellungen der Umgebungsvariablen, um die Lokalität zu initialisieren.

Wenn `LC_MESSAGES` geändert wird, hat dies keine Auswirkung auf Meldungskataloge, die schon von `catopen`-Aufrufen geöffnet worden sind.

BS2000
Beim Start eines Programms wird aus den in SYSPOSIX.*name* hinterlegten Variablen der Zeigervektor environ aufgebaut. Falls beim Aufruf von setlocale() als Lokalität eine leere Zeichenkette "" angegeben wird, sind die in diesem Vektor hinterlegten Umgebungsvariablen mit ihren Werten maßgeblich. Falls die abgefragte Umgebungsvariable nicht vorhanden ist, gilt der entsprechende Wert aus der POSIX-Lokalität. ❑

Zusätzlich zu den vordefinierten Lokalitäten lassen sich auch eigene Lokalitäten implementieren und durch setlocale() auswählen (siehe Abschnitt „Lokalität“ auf Seite 52).

Siehe auch catopen(), ctime(), ctype(), environ, exec, getdate(), gettxt(), isalnum(), isalpha(), iscntrl(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), iswalnum(), iswalpha(), iswcntrl(), iswgraph(), iswlower(), iswprint(), iswpunct(), iswspace(), iswupper(), localeconv(), mblen(), mbstowcs(), mbtowc(), nl_langinfo(), printf(), scanf(), strcoll(), strerror(), strfmon(), strtime(), strtod(), strxrfm(), tolower(), toupper(), towlower(), towupper(), wcscoll(), wcstod(), wcstombs(), wcsxfrm(), wctomb(), langinfo.h, locale.h, Abschnitt „Lokalität“ auf Seite 52.

setlogmask - Log Priority Mask setzen

Syntax

#include <syslog.h>

int setlogmask(int *maskpri*);

Beschreibung

siehe closelog()

setpgid - Prozessgruppennummer für Auftragssteuerung setzen

Syntax

```
#include <unistd.h>

Optional
#include <sys/types.h> ❑

int setpgid(pid_t pid, pid_t pgid);
```

Beschreibung

`setpgid()` wird benutzt, um sich entweder einer existierenden Prozessgruppe anzuschließen oder um eine neue Prozessgruppe innerhalb der Sitzung des aufrufenden Prozesses zu erzeugen. Wenn *pgid* gleich *pid* ist, wird der Prozess zu einem Prozessgruppenleiter. Wenn *pgid* ungleich *pid* ist, wird der Prozess Mitglied einer existierenden Gruppe. Die Prozessgruppennummer des Sitzungsleiters ändert sich nicht. Bei erfolgreicher Beendigung wird die Prozessgruppennummer des Prozesses mit der Prozessnummer, die zu *pid* passt, auf *pgid* gesetzt.

Wenn *pid* gleich 0 ist, wird die Prozessnummer des aufrufenden Prozesses verwendet.

Wenn *pgid* gleich 0 ist, wird die Prozessnummer des angegebenen Prozesses verwendet.

Returnwert

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

Fehler `setpgid()` schlägt fehl, wenn gilt:

`EACCES`	Der Wert von *pid* entspricht der Prozessnummer eines Sohnprozesses des aufrufenden Prozesses, und der Sohnprozess hat erfolgreich eine der `exec`-Funktionen aufgerufen.
`EINVAL`	Der Wert von *pgid* ist kleiner als 0 oder ein Wert, der von der Implementierung nicht unterstützt wird.
`EPERM`	Der Prozess, der durch *pid* angegeben wird, ist ein Sitzungsführer, oder der Wert von *pid* entspricht der Prozessnummer eines Sohnprozesses des aufrufenden Prozesses und der Sohnprozess ist nicht in derselben Sitzung wie der aufrufende Prozess, oder der Wert von *pgid* ist gültig, aber entspricht nicht der Prozessnummer des Prozesses, der durch *pid* angesprochen wird und es gibt keinen Prozess mit einer Prozessgruppennummer, die dem Wert von *pgid* in derselben Sitzung wie der aufrufende Prozess entspricht.
`ESRCH`	Der Wert von *pid* entspricht nicht der Prozessnummer des aufrufenden Prozesses oder eines Sohnprozesses des aufrufenden Prozesses.

Siehe auch `exec`, `getpgrp()`, `setsid()`, `tcsetpgrp()`, `sys/types.h`, `unistd.h`.

setpgrp - Prozessgruppennummer einstellen

Syntax #include <unistd.h>

pid_t setpgrp (void);

Beschreibung

Wenn der aufrufende Prozess nicht schon ein Sitzungsleiter (session leader) ist, so stellt `setpgrp()` die Prozessgruppennummer und die Sitzungsnummer des aufrufenden Prozesses auf die Prozessnummer des aufrufenden Prozesses und gibt das steuernde Terminal des aufrufenden Prozesses frei.

Die Funktion hat keine Wirkung, wenn der aufrufende Prozess ein Sitzungsleiter ist.

Returnwert `setpgrp()` gibt den Wert der neuen Prozessgruppennummer zurück.

Siehe auch `exec`, `fork()`, `getpid()`, `getsid()`, `kill()`, `setsid()`, `unistd.h`.

setpriority - Prozesspriorität setzen

Syntax #include <sys/resource.h>

int setpriority(int *which*, id_t *who*, int *priority*);

Beschreibung

siehe `getpriority()`.

setpwent - Zeiger zum Durchsuchen des Benutzerkatalogs löschen

Syntax #include <pwd.h>

void setpwent(void);

Beschreibung

Siehe `endpwent()`.

setregid - reale und effektive Gruppennummer setzen

Syntax

```
#include <unistd.h>

int setregid(gid_t rgid, grid_t egid);
```

Beschreibung

`setregid()` wird verwendet, um die reale und die effektive Gruppennummer des aufrufenden Prozesses zu setzen. Wenn *rgid* gleich -1 ist, wird die reale Gruppennummer (GID) nicht geändert; wenn *egid* gleich -1 ist, wird die effektive GID nicht geändert. Die reale und die effektive GID können im selben Aufruf auf verschiedene Werte gesetzt werden.

Entspricht die effektive Benutzernummer des aufrufenden Prozesses dem Superuser, können die reale GID und die effektive GID auf jeden zulässigen Wert gesetzt werden.

Entspricht die effektive Benutzernummer des aufrufenden Prozesses nicht dem Superuser, kann entweder die reale GID auf die gesicherte „set-GID“ aus `execv` gesetzt werden, oder die effektive GID kann entweder auf die gesicherte „set-GID“ oder auf die reale GID gesetzt werden.

Wenn ein Prozess zum Setzen der GID seine effektive GID auf seine reale GID setzt, kann er seine effektive GID immer noch auf die gesicherte „set-GID“ zurücksetzen.

Sowohl bei einer Änderung der realen GID (d. h. wenn *rgid* nicht gleich -1 ist) als auch bei der Änderung der effektiven GID in einen Wert, der nicht der realen GID entspricht, wird die gesicherte „set-GID“ mit der neuen effektiven GID gleichgesetzt.

Wird der aktuelle Wert der realen GID geändert, wird der alte Wert aus der Gruppenzugriffsliste gelöscht (siehe `getgroups()`), sofern er in dieser Liste eingetragen ist, und der neue Wert wird in die Gruppenzugriffsliste aufgenommen, wenn er nicht bereits existiert und wenn hierdurch nicht die Anzahl der Gruppen in dieser Liste NGROUPS überschreitet, wie in der Datei `/usr/include/sys/param.h` definiert.

Returnwert

0	bei erfolgreicher Ausführung
-1	bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `setregid()` schlägt fehl, wenn gilt:

EINVAL	Der Wert von *rgid* oder *egid* ist ungültig oder außerhalb des Wertebereiches.
EPERM	Die effektive Benutzernummer des aufrufenden Prozesses entspricht nicht der des Superuser, und es wurde eine andere Änderung als die Änderung der realen GID in die gesicherte „set-GID“ oder der effektiven GID in die reale oder die gesicherte GID angegeben.

Siehe auch `exec()`, `getuid()`, `setuid()`, `setreuid()`, `unistd.h`.

setreuid - reale und effektive Benutzernummer setzen

Syntax #include <unistd.h>

int setreuid(uid_t *ruid*, uid_t *euid)*

Beschreibung

`setreuid()` wird verwendet, um die reale und die effektive Benutzernummer des aufrufenden Prozesses zu setzen. Wenn *ruid* gleich -1 ist, wird die reale Benutzernummer nicht geändert; wenn *euid* gleich -1 ist, wird die effektive Benutzernummer nicht geändert. Die reale und die effektive Benutzernummer können im selben Aufruf auf verschiedene Werte gesetzt werden.

Entspricht die effektive Benutzernummer des aufrufenden Prozesses dem Superuser, können die reale Benutzernummer und die effektive Benutzernummer auf jeden zulässigen Wert gesetzt werden.

Entspricht die effektive Benutzernummer des aufrufenden Prozesses nicht dem Superuser, kann entweder die reale Benutzernummer auf die effektive Benutzernummer, oder die effektive Benutzernummer kann entweder auf die gesicherte „set-user-ID“ aus `execv` oder die reale Benutzernummer gesetzt werden.

Wenn ein Prozess zum Setzen der Benutzernummer (UID) seine effektive Benutzernummer auf seine reale Benutzernummer setzt, kann er seine effektive Benutzernummer immer noch auf die gesicherte „set-user-ID“ zurücksetzen.

Sowohl bei einer Änderung der realen Benutzernummer (d. h. wenn *ruid* nicht gleich -1 ist) als auch bei der Änderung der effektiven Benutzernummer in einen Wert, der nicht der realen Benutzernummer entspricht, wird die gesicherte „set-user-ID“ mit der neuen effektiven Benutzernummer gleichgesetzt.

Returnwert

0	bei erfolgreicher Ausführung
-1	bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `setreuid()` schlägt fehl, wenn gilt:

EINVAL	Der Wert des Arguments *ruid* oder *euid* ist ungültig oder außerhalb des Wertebereiches.
EPERM	Die effektive Benutzernummer des aufrufenden Prozesses ist nicht der Superuser, und es wurde eine andere Änderung als die Änderung der realen Benutzernummer in die effektive Benutzernummer oder der effektiven Benutzernummer in die reale oder die gesicherte „set-user-ID“ angegeben.

Siehe auch `getuid()`, `setuid()`, `unistd.h`

setrlimit - Grenzwert für ein Betriebsmittel setzen

Name **setrlimit, setrlimit64**

Syntax #include <sys/resource.h>

int setrlimit (int *resource*, const struct rlimit **rlp*);
int setrlimit64 (int *resource*, const struct rlimit64 **rlp*);

Beschreibung
siehe `getrlimit()`.

setsid - Sitzung erzeugen und Prozessgruppennummer setzen

Syntax #include <unistd.h>

Optional #include <sys/types.h> ❑

pid_t setsid(void);

Beschreibung

Wenn der aufrufende Prozess kein Prozessgruppenleiter ist, erzeugt `setsid()` eine neue Sitzung. Nach der Rückkehr dieser Funktion ist der aufrufende Prozess der Sitzungsleiter dieser neuen Sitzung und der Prozessgruppenleiter einer neuen Prozessgruppe. Außerdem besitzt der Prozess kein steuerndes Terminal. Die Prozessgruppennummer des aufrufenden Prozesses wird gleich der Prozessnummer des aufrufenden Prozesses gesetzt. Der aufrufende Prozess ist der einzige Prozess in der neuen Prozessgruppe und der einzige Prozess in der neuen Sitzung.

Returnwert Prozessgruppennummer des aufrufenden Prozesses bei Erfolg.

(`pid_t`) -1 bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `setsid()` schlägt fehl, wenn gilt:

`EPERM` Der aufrufende Prozess ist bereits Prozessgruppenleiter oder die Prozessgruppennummer eines anderen Prozesses als des aufrufenden stimmt mit der Prozessnummer des aufrufenden Prozesses überein.

Hinweis Wenn der aufrufende Prozess die letzte Komponente einer Pipe ist, die von einer Job-Kontroll-Shell gestartet worden ist, kann die Shell den aufrufenden Prozess zum Prozessgruppenleiter machen. Die anderen Prozesse der Pipeline werden Mitglieder der Prozessgruppe. In diesem Fall schlägt der Aufruf von `setsid()` fehl. Aus diesem Grund sollte ein Prozess, der `setsid()` aufruft und davon ausgeht, Teil einer Pipe zu sein, vorher immer ein `fork()` ausführen; der Vaterprozess sollte beendet werden, und der Sohnprozess sollte `setsid()` aufrufen und dadurch versichern, dass der Prozess zuverlässig funktioniert, ob er nun von Job-Kontroll-Shells aufgerufen wird oder nicht (siehe Handbuch „POSIX Grundlagen (BS2000/OSD)“ und Handbuch „POSIX Kommandos (BS2000/OSD)“).

Siehe auch `setpgid()`, `sys/types.h`, `unistd.h`.

setstate - Pseudozufallszahlen

Syntax #include <stdlib.h>

char *setstate(const char **state*);

Beschreibung

siehe `initstate()`.

setuid - Benutzernummer setzen

Syntax #include <unistd.h>

Optional
#include <sys/types.h> ❑

int setuid(uid_t *uid*);

Beschreibung

Wenn der Prozess Sonderrechte hat, setzt die Funktion `setuid()` die reale, die effektive und die gesicherte Benutzernummer gleich *uid*.

Wenn der Prozess keine Sonderrechte hat, aber *uid* gleich der realen oder der gesicherten Benutzernummer ist, setzt `setuid()` die effektive Benutzernummer gleich *uid*. Die reale und die gesicherte Benutzernummer bleiben unverändert.

Returnwert

0	bei Erfolg.
–1	bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `setuid()` schlägt fehl, wenn gilt:

`EPERM`	Der Prozess hat keine Sonderrechte und *uid* entspricht auch nicht seiner realen oder gesicherten Benutzernummer.

Hinweis Eine häufige Anwendung von `setuid()` ist die Aufgabe von nicht mehr benötigten Rechten in Programmen mit gesetztem s-Bit für den Eigentümer (insbesondere `root`). Solche Programme benötigen die durch das s-Bit gewährten Rechte oft nur für ganz bestimmte Aufgaben. Werden die Rechte nicht mehr benötigt, so können sie durch einen Aufruf der folgenden Form wieder aufgegeben werden:

```
erg = setuid(getuid());
```

Siehe auch `setpgid()`, `sys/types.h`, `unistd.h`.

setutxent - Zeiger auf utmpx-Datei zurücksetzen

Syntax #include <utmpx.h>

void setutxent (void);

Beschreibung

Siehe `endutxent()`.

setvbuf - Puffer einem Datenstrom zuweisen

Syntax

```
#include <stdio.h>

int setvbuf(FILE *stream, char *buf, int type, size_t size);
```

Beschreibung

`setvbuf()` kann verwendet werden, nachdem der Datenstrom, auf den *stream* zeigt, einer offenen Datei zugeordnet wurde, aber bevor eine andere Operation auf dem Datenstrom ausgeführt wird. `setvbuf()` bewirkt, dass das Feld, auf das *buf* zeigt, an Stelle eines automatisch zugewiesenen Puffers verwendet wird. Wenn *buf* der Nullzeiger ist, sind Ein- und Ausgaben völlig ungepuffert.

type bestimmt folgendermaßen, wie *stream* gepuffert werden soll:

`_IOFBF`	sorgt für vollständige Pufferung der Ein- und Ausgaben.
`_IOLBF`	sorgt für zeilenweise Pufferung.
`_IONBF`	sorgt für ungepufferte Ein- und Ausgaben.

Wenn *buf* nicht der Nullzeiger ist, kann der Vektor, auf den *buf* zeigt, an Stelle eines von `setvbuf()` reservierten Puffers verwendet werden.

size gibt die Größe des *buf*-Vektors an.

Der Inhalt des *buf*-Vektors ist zu jeder Zeit unbestimmt.

Returnwert

0	bei Erfolg.
≠ 0	wenn ein ungültiger Wert für *type* angegeben wurde oder wenn die Anforderung nicht ausgeführt werden kann. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `setvbuf()` schlägt fehl, wenn gilt:

`EBADF`	Der *stream* zu Grunde liegende Dateideskriptor ist ungültig.

Hinweis

Eine häufige Fehlerquelle besteht darin, dass als Puffer in einem Programmblock eine Variable der Speicherklasse `auto` verwendet und die Datei in diesem Block dann nicht geschlossen wird.

Teile von *buf* werden für interne Verwaltungsinformationen des Datenstroms benötigt, deswegen enthält *buf* weniger als *size* Bytes, wenn er voll ist. `setvbuf()` sollte automatisch zugewiesene Puffer verwenden.

Wenn man mit `setvbuf()` Speicherbereich der Größe *size* zuweist, bedeutet das nicht automatisch, dass alle *size* Bytes für den Speicherbereich gebraucht werden.

Anwendungen sollten beachten, dass viele Implementierungen nur zeilenweises Puffern von Terminal-Gerätedateien unterstützen.

`setvbuf()` wird für die Datei ausgeführt, die *stream* zugeordnet ist. Die kann eine POSIX- oder BS2000-Datei sein.

BS2000
Wird der Blockungsfaktor mit dem `BUFFER-LENGTH`-Parameter des `ADD-FILE-LINK`-Kommandos explizit vereinbart, muss die Größe des Bereichs dieser vereinbarten Blockungsgröße entsprechen. ❑

Siehe auch `fopen()`, `setbuf()`, `stdio.h`, Abschnitt „Datenströme“ auf Seite 76.

shmat - gemeinsam nutzbaren Speicherbereich anhängen

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

void *shmat(int shmid, const void*shmaddr, int shmflg);
```

Beschreibung

`shmat()` hängt das mit der Kennzahl für gemeinsam nutzbaren Speicherbereich *shmid* bezeichnete, gemeinsam benutzte Speichersegment an das Datensegment des aufrufenden Prozesses an. An welcher Stelle das Segment angehängt wird, richtet sich nach folgenden Kriterien:

– Wenn *shmaddr* gleich 0 ist, wird das Segment an der ersten vom System festgestellten freien Adresse angehängt.

– Wenn *shmaddr* und (*shmflg* `& SHM_RND)` ungleich 0 sind, wird das Segment an der mit (*shmaddr*−`(ptrdiff_t)`*shmaddr* `% SHMLBA`)) angegebenen Adresse angehängt. (Das Zeichen % ist der Modulo-Operator der Sprache C.)

– Wenn *shmaddr* ungleich 0 und (*shmflg* `& SHM_RND`) gleich 0 sind, wird das Segment an der mit *shmaddr* angegebenen Adresse angehängt.

– Wenn (*shmflg* `& SHM_RDONLY`) ungleich 0 ist und der aufrufende Prozess das Leserecht hat, wird das Segment zum Lesen angehängt.

– Wenn (*shmflg* `& SHM_RDONLY`) gleich 0 ist und der aufrufende Prozess Schreib- und Leserecht hat, wird das Segment zum Lesen und Schreiben angehängt.

Folgende symbolische Namen sind in der Include-Datei `sys/shm.h` definiert:

Name	Beschreibung
`SHMLBA`	Vielfaches der Adresse der Segmentuntergrenze
`SHM_RDONLY`	Anfügen nur zum Lesen
`SHM_RND`	Anhängeadresse aufrunden

Returnwert

Startadresse des Datensegments für den gemeinsam nutzbaren Speicherbereich bei Erfolg. Der Wert von `shm_nattach` wird in der Datenstruktur inkrementiert, die mit der Kennzahl für den gemeinsam nutzbaren Speicherbereich verbunden ist.

-1 bei Fehler. Das gemeinsame Speichersegment wird nicht angehängt. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler

`shmat()` schlägt fehl, wenn gilt:

`EACCES` Dem aufrufenden Prozess werden die für die Operation benötigten Zugriffsrechte verweigert.

EINVAL	Der Wert von *shmid* ist keine gültige Kennzahl für gemeinsam nutzbaren Speicherbereich, oder der Wert von *shmaddr* ist ungleich 0 und der Wert von (*shmaddr* −((ptrdiff_t) *shmaddr* % SHMLBA)) ist eine unzulässige Adresse für das Anfügen von gemeinsam nutzbarem Speicher, oder der Wert von *shmaddr* ist ungleich 0, (*shmflg* & SHM_RND) ist gleich 0 und der Wert von *shmaddr* ist eine unzulässige Adresse für das Anfügen von gemeinsamem Speicher.
EMFILE	Die Anzahl der beim aufrufenden Prozess angehängten gemeinsamen Speichersegmente würde die systemspezifische Grenze überschreiten.
ENOMEM	Der verfügbare Datenspeicher ist nicht groß genug, um das gemeinsame Speichersegment unterzubringen.

Hinweis Das Kommittee des IEEE 1003.4-Standards entwickelt gerade eine alternative Schnittstelle für die Interprozeßkommunikation. Anwendungsprogrammierer, die Interprozeßkommunikation einsetzen, sollten die Anwendungen so konzipieren, daß Module, die derzeit beschriebene Funktionen für Interprozeßkommunikation benutzen, einfach geändert werden können.

Siehe auch exec, exit(), fork(), shmctl(), shmdt(), shmget(), sys/shm.h, Abschnitt „Interprozesskommunikation“ auf Seite 116.

shmctl - gemeinsam nutzbaren Speicherbereich steuern

Syntax

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

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Beschreibung

`shmctl()` bietet eine Vielzahl von Steuerungsoperationen für gemeinsam nutzbare Speicherbereiche („shared memory“), die mit *cmd* angegeben werden. Die folgenden Werte für *cmd* sind verfügbar:

`IPC_STAT` Aktuelle Werte aller Komponenten der *shmid* zugeordneten Datenstruktur `shmid_ds` in die Struktur eintragen, auf die *buf* zeigt. Der Aufbau der Struktur wird in `sys/shm.h` definiert.

`IPC_SET` Die Werte folgender Komponenten der *shmid* zugeordneten Datenstruktur `shmid_ds` auf die entsprechenden Werte aus der Struktur setzen, auf die *buf* zeigt:

```
shm_perm.uid
shm_perm.gid
shm_perm.mode    /* nur 9 niederwertige Bits */
```

`IPC_SET` kann nur von einem Prozess ausgeführt werden, dessen effektive Benutzernummer gleich der eines Prozesses mit Sonderrechten oder gleich dem Wert von `shm_perm.cuid` bzw. `shm_perm.uid` in der *shmid* zugeordneten Datenstruktur `shmid_ds` ist.

`IPC_RMID` Die mit *shmid* angegebene shared-memory-Speicherkennzahl im System sowie das dazugehörige Speichersegment und die dazugehörige Datenstruktur `shmid_ds` löschen. `IPC_RMID` kann nur von einem Prozess ausgeführt werden, dessen effektive Benutzernummer gleich der eines Prozesses mit besonderen Rechten oder gleich dem Wert von `shm_perm.cuid` bzw. `shm_perm.uid` in der *shmid* zugeordneten Datenstruktur `shmid_ds` ist.

Returnwert

0 bei Erfolg.

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

Fehler `shmctl()` schlägt fehl, wenn gilt:

`EACCES` *cmd* ist gleich `IPC_STAT` und der aufrufende Prozess hat kein Leserecht.

Erweiterung
`EFAULT` *msgp* verweist auf eine unzulässige Adresse. ❑

`EINVAL` Der Wert von *shmid* ist keine gültige Kennzahl für gemeinsam nutzbare Speicherbereiche,
oder der Wert von *cmd* ist kein gültiges Kommando,
oder *cmd* ist `IPC_SET` und `shm_perm.uid` oder `shm_perm.gid` sind ungültig.

Erweiterung

`ENOMEM` Es steht nicht genügend Speicher zur Verfügung. ❑

`EPERM` *cmd* ist gleich `IPC_RMID` oder `IPC_SET`, die effektive Benutzernummer des aufrufenden Prozesses ist nicht die eines Prozesses mit Sonderrechten und stimmt nicht mit `shm_perm.cuid` oder `shm_perm.uid` in der *shmid* zugeordneten Datenstruktur überein.

Hinweis Das Kommittee des IEEE 1003.4-Standards entwickelt gerade eine alternative Schnittstelle für die Interprozeßkommunikation. Anwendungsprogrammierer, die Interprozeßkommunikation einsetzen, sollten die Anwendungen so konzipieren, daß Module, die derzeit beschriebene Funktionen für Interprozeßkommunikation benutzen, einfach geändert werden können.

Siehe auch `shmat()`, `shmdt()`, `shmget()`, `sys/shm.h`, Abschnitt „Interprozesskommunikation“ auf Seite 116.

shmdt - gemeinsam nutzbaren Speicherbereich abhängen

Syntax

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

int shmdt(const void*shmaddr);
```

Beschreibung

`shmdt()` hängt das gemeinsam nutzbare Speichersegment vom Datensegment des aufrufenden Prozesses ab, das sich an der durch *shmaddr* angegebenen Adresse befindet.

Einschränkung
Da in dieser Version des POSIX-Subsystems ein gemeinsam nutzbarer Speicherbereich nur existieren kann, wenn er an einen Prozess gebunden ist, weicht das Verhalten von `shmdt()` von XPG4 in folgendem Punkt ab: Wenn sich der letzte Prozess von einem gemeinsam nutzbaren Speicherbereich abgehängt hat, wird dieser Speicherbereich freigegeben. Der POSIX-Kernel behält jedoch die Verwaltungsinformationen über diesen Speicherbereich. Hängt sich nun ein anderer Prozess wieder an das gemeinsam nutzbare Speichersegment an, ist dessen früherer Inhalt verloren. ❑

Returnwert

0	bei Erfolg. `shmdt()` dekrementiert den Wert von `shm_attach` in der Datenstruktur, die mit der Kennzahl des gemeinsam nutzbaren Speicherbereichs verbunden ist.
-1	bei Fehler. Das gemeinsam nutzbare Speichersegment wird nicht abgehängt. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `shmdt()` schlägt fehl, wenn gilt:

`EINVAL`	Der Wert von *shmaddr* ist nicht die Datensegment-Startadresse eines gemeinsamen Speichersegments.

Hinweis Das Kommittee des IEEE 1003.4-Standards entwickelt gerade eine alternative Schnittstelle für die Interprozeßkommunikation. Anwendungsprogrammierer, die Interprozeßkommunikation einsetzen, sollten die Anwendungen so konzipieren, daß Module, die derzeit beschriebene Funktionen für Interprozeßkommunikation benutzen, einfach geändert werden können.

Siehe auch `exec`, `exit()`, `fork()`, `shmat()`, `shmctl()`, `shmget()`, `sys/shm.h`, Abschnitt „Interprozesskommunikation“ auf Seite 116.

shmget - gemeinsam nutzbaren Speicherbereich anlegen

Syntax #include <sys/shm.h>

int shmget(key_t *key*, int *size*, int *shmflg*);

Beschreibung

`shmget()` liefert die *key* zugeordnete Kennzahl für gemeinsam nutzbaren Speicherbereich.

Es wird eine Kennzahl für gemeinsam nutzbaren Speicherbereich mit dazugehöriger Datenstruktur und das dazugehörige Speichersegment in einer Größe von mindestens *size* Bytes (siehe `sys/shm.h`) für *key* eingerichtet, wenn eine der folgenden Bedingungen zutrifft:

– Das Argument *key* hat den Wert `IPC_PRIVATE`.

– Das Argument *key* besitzt noch keine ihm zugeordnete Kennzahl für gemeinsam nutzbaren Speicherbereich und (*shmflg* `&` `IPC_CREAT`) ist ungleich 0.

Beim Einrichten der neuen Kennzahl für gemeinsam nutzbaren Speicherbereich wird die dazugehörige Datenstruktur wie folgt initialisiert:

– Die Werte von `shm_perm.cuid`, `shm_perm.uid`, `shm_perm.cgid` und `shm_perm.gid` werden gleich der effektiven Benutzer- bzw. Gruppennummer des aufrufenden Prozesses gesetzt.

– Die 9 niederwertigen Bits von `shm_perm.mode` werden gleich den 9 niederwertigen Bits von *shmflg* gesetzt. Das Argument `shm_segsz` wird auf den Wert von *size* gesetzt.

– Die Werte von `shm_lpid`, `shm_nattch`, `shm_atime` und `shm_dtime` werden auf 0 gesetzt.

– Für `shm_ctime` wird die aktuelle Zeit eingetragen.

Returnwert Kennzahl für gemeinsam nutzbaren Speicherbereich
bei Erfolg. Die Kennzahl ist eine nichtnegative ganze Zahl.

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

Fehler `shmget()` schlägt fehl, wenn gilt:

`EACCES` Es existiert eine Kennzahl für gemeinsam nutzbaren Speicherbereich für das Argument *key*, aber die in den 9 niederwertigen Bits von *shmflg* angegebene Berechtigung wurde nicht erteilt.

`EEXIST` Für *key* existiert eine Kennzahl für gemeinsam nutzbaren Speicherbereich, aber ((*shmflg* `&` `IPC_CREAT`) `&&` (*shmflg* `&` `IPC_EXCL`)) ist ungleich 0.

EINVAL	Der Wert von *size* ist kleiner als der systemspezifische Minimalwert oder größer als der systemspezifische Maximalwert, oder für *key* existiert bereits eine Kennzahl für gemeinsam nutzbaren Speicherbereich, aber die Größe des ihr zugeordneten Segments ist kleiner als *size* und *size* ist ungleich 0.
ENOENT	Für *key* existiert keine Kennzahl für gemeinsam nutzbaren Speicherbereich und (*shmflg* & IPC_CREAT) ist gleich 0.
ENOMEM	Der vorhandene physikalische Speicherplatz würde überschritten werden.
ENOSPC	Der systemspezifische Maximalwert für Kennzahlen für gemeinsam nutzbaren Speicherbereich würde überschritten.

Hinweis Das Kommittee des IEEE 1003.4-Standards entwickelt gerade eine alternative Schnittstelle für die Interprozeßkommunikation. Anwendungsprogrammierer, die Interprozeßkommunikation einsetzen, sollten die Anwendungen so konzipieren, daß Module, die derzeit beschriebene Funktionen für Interprozeßkommunikation benutzen, einfach geändert werden können.

BS2000
Es wird nicht verhindert, dass eine Task, die nur Leserecht hat, mit BS2000-Mitteln auch schreibend auf den gemeinsam nutzbaren Speicherbereich zugreift. ❑

Siehe auch shmat(), shmctl(), shmdt(), sys/shm.h, Abschnitt „Interprozesskommunikation“ auf Seite 116.

sigaction - Signalbehandlung ermitteln oder ändern

Syntax #include <signal.h>

int sigaction(int *sig*, const struct sigaction **act*, struct sigaction **oact*);

Beschreibung

`sigaction()` erlaubt es dem aufrufenden Prozess, die mit dem Signal *sig* verbundene Signalbehandlung zu ermitteln oder zu ändern. Mögliche Werte für *sig* sind in der Datei `signal.h` definiert (siehe `signal.h`).

Die Struktur `sigaction`, die zur Beschreibung von Signalbehandlungen benutzt wird, ist in der Datei `signal.h` definiert und beinhaltet zumindest folgende Komponenten:

Komponententyp	Komponentenname	Beschreibung
`void(*)(int)`	`sa_handler`	`SIG_DFL`, `SIG_IGN` oder ein Zeiger auf eine Signalbehandlungsfunktion.
`sigset_t`	`sa_mask`	Zusätzliche Signalmenge, die während der Abarbeitung der Signalbehandlungsfunktion blockiert werden soll.
`int`	`sa_flags`	Spezielle Flags, mit denen das Verhalten von *sig* beeinflusst werden kann.

Wenn *act* kein Nullzeiger ist, zeigt es auf eine Struktur, welche die neue Signalbehandlung für *sig* beschreibt; d.h. die aktuelle Signalbehandlung wird geändert. In diesem Fall sollte *oact* auf eine Struktur zeigen, in der die aktuelle Signalbehandlung nach der Rückkehr von `sigaction()` abgespeichert wird.

Wenn *act* ein Nullzeiger ist, bleibt die aktuelle Signalbehandlung unverändert; sie kann mit diesem `sigaction`-Aufruf für ein gegebenes Signal ermittelt werden. In diesem Fall kann *oact* der Nullzeiger sein.

`sa_handler` identifiziert die Signalaktion für *sig* und kann die Werte annehmen, die in `signal.h` als Signalaktionen definiert sind (siehe `signal.h`).

Wenn `sa_handler` eine Signalbehandlungsfunktion festlegt, gibt die Komponente `sa_mask` eine Signalmenge an, die vor Aufruf der Signalbehandlungsfunktion der Signalmaske des Prozesses hinzugefügt wird Die Signale `SIGKILL` und `SIGSTOP` können nicht blockiert werden; diese Einschränkung wird vom System erzwungen, ohne dass ein Fehler angezeigt wird.

sa_flags kann verwendet werden, um das Verhalten des angegebenen Signals zu ändern. Die folgenden, in der Datei signal.h definierten Flag-Bits können in sa_flags gesetzt werden:

SA_NOCLDSTOP Verhindert, dass SIGCHLD erzeugt wird, wenn ein Sohnprozess anhält.

Erweiterung
SA_NOCLDWAIT
: Wenn dieses Flag-Bit gesetzt und *sig* gleich SIGCHLD ist, erzeugt das System keine Zombie-Prozesse, wenn Sohnprozesse des aufrufenden Prozesses beendet werden. Führt der aufrufende Prozess aufeinander folgende wait-Aufrufe aus, wird blockiert, bis alle Sohnprozesse des aufrufenden Prozesses beendet sind; danach wird der Wert -1 zurückgegeben und errno enthält ECHILD.

SA_NODEFER
: Das Signal wird vom System nicht automatisch blockiert, während es von der Signalbehandlungsfunktion bearbeitet wird.

SA_RESETHAND
: Wenn diese Option gesetzt ist und das Signal behandelt wird, wird die Disposition des Signals auf SIG_DFL zurückgesetzt und das Signal bei Einsprung in die Signalbehandlungsroutine blockiert (SIGILL, SIGTRAP und SIGPWR können nicht automatisch zurückgesetzt werden, wenn sie empfangen werden; das System erzwingt diese Beschränkung stillschweigend).

SA_RESTART
: Wenn dieses Flag-Bit gesetzt ist und das Signal behandelt wird, wird ein Systemaufruf, der durch die Ausführung der Signalbehandlungsroutine unterbrochen wird, vom System neu gestartet. Dies geschieht transparent. Ansonsten liefert der Systemaufruf den Fehler EINTR.

SA_SIGINFO
: Wenn dieses Flag-Bit nicht gesetzt ist und das Signal behandelt wird, wird *sig* als einziges Argument an die Funktion gesendet, welche die Signale abfängt.
Wenn die Option gesetzt ist und das Signal behandelt wird, werden blockierte Signale vom Typ *sig* zuverlässig für den aufrufenden Prozess in die Warteschlange aufgenommen und zwei zusätzliche Argumente an die Funktion übergeben, die das Signal bearbeitet. Wenn das zweite Argument nicht gleich dem Nullzeiger ist, zeigt es auf eine Struktur vom Typ siginfo_t, welche den Grund für das Signal enthält; das dritte Argument zeigt auf eine Struktur vom Typ ucontext_t, welche den Kontext des empfangenden Prozesses zurzeit des Signalempfangs enthält. ❑

Wenn *sig* gleich SIGCHLD ist und SA_NOCLDSTOP nicht in sa_flags gesetzt ist, wird das Signal SIGCHLD jedes Mal an den aufrufenden Prozess gesendet, wenn einer seiner Sohnprozesse anhält. Wenn *sig* gleich SIGCHLD ist und SA_NOCLDSTOP in sa_flags gesetzt ist, wird kein SIGCHLD-Signal erzeugt.

Wenn ein Signal durch eine mit `sigaction()` festgelegte Signalbehandlungsfunktion abgefangen wird, wird für die Laufzeit der Signalbehandlungsfunktion (bzw. bis entweder `sigprocmask()` oder `sigsuspend()` aufgerufen wird) eine neue Signalmaske berechnet. Diese Maske wird aus der Vereinigung der aktuellen Signalmaske und dem Wert aus `sa_mask` für das gesendete Signal gebildet, einschließlich des gesendeten Signals selbst. Wenn die benutzerdefinierte Signalbehandlungsfunktion normal beendet wird, wird die ursprüngliche Maske wiederhergestellt.

Die aktuelle Signalbehandlung für *sig* ist solange gültig, bis erneut `sigaction()` oder eine der `exec`-Funktionen aufgerufen wird.

Wenn die vorherige Signalbehandlung *oact* für *sig* durch `signal()` festgelegt wurde, sind die Werte der Strukturkomponenten, auf die *oact* zeigt, nicht spezifiziert und in der speziellen Komponente *oact*->`sv_handler` befindet sich nicht notwendig derselbe Wert, der vorher von `signal()` übergeben wurde. Trotzdem wird, wenn ein Zeiger auf dieselbe Struktur oder eine Kopie davon über *act* an einen nachfolgenden Aufruf von `sigaction()` übergeben wird, die Signalbehandlung so sein, als ob der ursprüngliche Aufruf von `signal()` wiederholt worden wäre.

Bei einem Versuch, eine Signalaktion festzulegen, die nicht abgefangen werden kann oder von `SIG_DFL` ignoriert wird, wird `errno` auf `EINVAL` gesetzt.

Allgemeines zur Signalbehandlung

Ein Signal wird für einen Prozess **erzeugt** (oder an einen Prozess **gesendet**), wenn das Ereignis, welches das Signal auslöst, erstmalig eintritt. Beispiele für solche **Ereignisse** sind die Erkennung von Hardware-Fehlern, das Ablaufen von Zeitgebern, Bildschirmaktivitäten oder ein Aufruf von `kill()`. Unter bestimmten Umständen erzeugt ein Ereignis Signale für mehrere Prozesse.

Jeder Prozess muss dafür sorgen, dass eine Signalaktion für jedes vom System definierte Signal festgelegt ist (siehe „Signalaktionen“ auf Seite 752). Ein Signal an einen Prozess nennt man **zugestellt**, wenn die für Prozess und Signal vorgesehene Signalaktion gestartet wird.

In der Zeit zwischen Signalerzeugung und -zustellung heißt ein Signal **anstehend**. Normalerweise kann diese Zeitspanne nicht von einer Anwendung erkannt werden. Dennoch kann ein Signal von der Zustellung an einen Prozess abgehalten, es kann **blockiert** werden. Wenn die Signalaktion, die einem blockierten Signal zugeordnet ist, eine andere Signalaktion als das Ignorieren des Signals ist und das Signal für den Prozess erzeugt wurde, bleibt das Signal solange anstehend, bis entweder die Blockierung aufgehoben wird oder die diesem Signal zugeordnete Signalaktion gleich Ignorieren gesetzt wird. Wenn die einem blockierten Signal zugeordnete Signalaktion das Ignorieren des Signals ist und das Signal für den Prozess erzeugt wurde, ist nicht festgelegt, ob das Signal sofort nach der Erzeugung aufgegeben wird oder ob es anstehend bleibt.

Jeder Prozess besitzt eine **Signalmaske**, die diejenigen Signale definiert, die derzeit vor der Zustellung an diesen Prozess blockiert werden. Die Signalmaske eines Prozesses wird von dessen Vaterprozess initialisiert. `sigaction()`, `sigprocmask()` und `sigsuspend()` steuern die Manipulation dieser Signalmaske.

Die Entscheidung, welche Signalaktion als Antwort auf ein Signal ausgeführt wird, wird zu dem Zeitpunkt getroffen, zu dem das Signal zugestellt wird. Dabei können auch nach dem Zeitpunkt der Erzeugung beliebige Änderungen vorgenommen werden. Diese Entscheidung ist unabhängig von dem Weg, auf dem ein Signal ursprünglich erzeugt wurde. Wenn ein bereits anstehendes Signal erzeugt wird, ist es undefiniert, ob dieses Signal mehr als einmal zugestellt wird. Die Reihenfolge, in der mehrere gleichzeitig anstehende Signale an einen Prozess zugestellt werden, ist nicht festgelegt.

Wenn ein **Haltesignal** (`SIGSTOP`, `SIGTSTP`, `SIGTTIN`, `SIGTTOU`) für einen Prozess erzeugt wird, wird ein evtl. anstehendes Signal des Typs `SIGCONT` senden. Umgekehrt werden, sobald ein Signal des Typs `SIGCONT` für einen Prozess erzeugt wird, alle noch ausstehenden Haltesignale für diesen Prozess gesendet. Wenn `SIGCONT` für einen Prozess erzeugt wird, der angehalten ist, so wird dieser Prozess fortgesetzt, auch wenn das Signal `SIGCONT` blockiert ist oder ignoriert wird. Wenn das Signal `SIGCONT` blockiert ist und nicht ignoriert wird, bleibt es anstehend bis es entweder freigegeben wird oder bis ein Haltesignal für den Prozess erzeugt wird.

Signalaktionen

Folgende Signalaktionen, können einem Signal zugeordnet werden:

- `SIG_DFL`
- `SIG_IGN`
- ein Zeiger auf eine Signalbehandlungsfunktion

Vor dem Eintritt in `main()` sind alle Signale auf `SIG_DFL` oder `SIG_IGN` gesetzt (siehe `signal.h`). Die durch diese Werte beschriebenen Signalaktionen bewirken Folgendes:

`SIG_DFL` - voreingestellte Signalbehandlung:

- Die voreingestellte Signalbehandlung für die unterstützten Signale wird unter `signal.h` beschrieben.
- Wenn die voreingestellte Signalbehandlung das Anhalten des Prozesses ist, wird die Ausführung des Prozesses zeitweilig unterbrochen. Wenn ein Prozess anhält, wird ein Signal des Typs `SIGCHLD` für dessen Vater-Prozess erzeugt, solange dieser nicht das Flag `SA_NOCLDSTOP` gesetzt hat. Solange ein Prozess angehalten ist, werden alle weiteren Signale, die an diesen Prozess gesendet werden, nicht mehr zugestellt, bis der Prozess fortgesetzt wird. Eine Ausnahme bildet das Signal `SIGKILL`, das den empfangenden Prozess immer abbricht. Einem Prozess, der Mitglied in einer verwaisten Pro-

zessgruppe ist, ist es nicht erlaubt, als Antwort auf eines der Signale SIGTSTP, SIGTTIN oder SIGTTOU anzuhalten. In den Fällen, in denen die Zustellung eines dieser Signale einen solchen Prozess anhalten würden, wird dieses Signal aufgegeben.

– Wenn die Signalbehandlung für ein anstehendes Signal, dessen voreingestellte Signalbehandlung das Ignorieren dieses Signals ist, auf SIG_DFL gesetzt wird (z.B. SIGCHLD), so wird das anstehende Signal aufgegeben, gleichgültig ob es blockiert ist oder nicht.

SIG_IGN - Signal ignorieren:

– Die Zustellung des Signals hat keine Wirkung auf den Prozess. Das Verhalten eines Prozesses ist undefiniert, nachdem dieser eines der Signale SIGFPE, SIGILL oder SIGSEGV ignoriert hat, sofern diese Signale nicht durch kill() oder raise() gesendet werden.

– Das System erlaubt die Signalaktion SIG_IGN nicht für die Signale SIGKILL oder SIGSTOP. Wenn die Signalaktion für ein anstehendes Signal auf SIG_IGN gesetzt wird, wird das anstehende Signal aufgegeben, gleichgültig ob es blockiert ist oder nicht.

– Wenn ein Prozess die Signalaktion für das Signal SIGCHLD auf SIG_IGN setzt, wird das Signal ignoriert.

Zeiger auf Signalbehandlungsfunktion - Signal abfangen:

– Bei der Zustellung eines Signals hat der empfangende Prozess eine das Signal abfangende Funktion an einer festgelegten Adresse auszuführen. Nach der Rückkehr aus der Signalbehandlungsfunktion nimmt der Prozess die Ausführung an dem Punkt wieder auf, an dem er unterbrochen wurde.

– Die Signalbehandlungsfunktion wird wie eine C-Funktion in der folgenden Art und Weise aufgerufen:

```
void func (int signo);
```

– *func* ist die angegebene Signalbehandlungsfunktion und *signo* die Nummer des Signals, das zugestellt wird.

– Das Verhalten eines Prozesses ist nicht definiert, wenn er normal von einer Fehlerbehandlungsfunktion für eines der Signale SIGFPE, SIGILL oder SIGSEGV zurückkehrt, das nicht von kill() oder von raise() erzeugt wurde.

– Das System verbietet es einem Prozess, die Signale SIGKILL und SIGSTOP abzufangen.

– Wenn ein Prozess eine Signalbehandlungsfunktion für das Signal SIGCHLD einführt, während er einen beendeten Sohnprozess besitzt, auf den er nicht wartet, ist nicht festgelegt, ob ein Signal des Typs SIGCHLD erzeugt wird, um diesen Sohn-Prozess anzuzeigen.

– Wenn Signalbehandlungsfunktionen asynchron zur Prozessausführung aufgerufen werden, ist das Verhalten einiger in diesem Handbuch beschriebener Funktionen nicht definiert, wenn diese aus einer Signalbehandlungsfunktion heraus aufgerufen werden. Die folgende Tabelle listet eine Reihe von Funktionen auf, die entweder **simultan nutzbar** oder nicht durch Signale unterbrechbar sind. Daher können diese so genannten **sicheren** Funktionen ohne Einschränkung von Anwendungen aus Signalbehandlungsfunktionen heraus aufgerufen werden:

Einschränkung

access()	free()	raise()	sysconf()
alarm()	fstat()	read()	tcdrain()
calloc()	getegid()	rename()	tcflow()
cfgetispeed()	geteuid()	rmdir()	tcflush()
cfgetospeed()	getgid()	setgid()	tcgetattr()
cfsetispeed()	getgroups()	setpgid()	tcgetpgrp()
cfsetospeed()	getpgrp()	setsid()	tcsendbreak()
chdir()	getpid()	setuid()	tcsetattr()
chmod()	getppid()	sigaction()	tcsetpgrp()
chown()	getuid()	sigaddset()	time()
close()	kill()	sigdelset()	times()
creat()	link()	sigemptyset()	umask()
dup2()	lseek()	sigfillset()	uname()
dup()	malloc()	sigismember()	unlink()
execle()	mkdir()	signal()	utime()
execve()	mkfifo()	sigpending()	wait()
_exit()	open()	sigprocmask()	waitpid()
fcntl()	pathconf()	sigsuspend()	write()
fork()	pause()	sleep()	
fpathconf()	pipe()	stat()	

Alle Funktionen, die nicht in der obigen Tabelle aufgeführt sind, gelten als **unsicher** in Bezug auf Signale. In Gegenwart von Signalen verhalten sich alle X/Open-konformen Funktionen wie definiert, wenn sie von einer Signalbehandlungsfunktion aufgerufen oder unter-

brochen werden, mit einer einzigen Ausnahme: Wenn eine unsichere Funktion durch ein Signal unterbrochen wird und die Signalbehandlungsfunktion eine unsichere Funktion aufruft, ist das Verhalten undefiniert.

Wirkung von Signalen auf andere Funktionen

Signale beeinflussen das Verhalten der folgenden Funktionen, wenn sie an einen Prozess gesendet werden, der gerade eine dieser Funktionen ausführt:

catclose()	fgetwc()	getgrnam()	tcdrain()
catgets()	fopen()	getpass()	tcsetattr()
close()	fputc()	getpwnam()	tmpfile()
dup()	fputwc()	getpwuid()	wait()
fclose()	freopen()	open()	write()
fcntl()	fseek()	pause()	
fflush()	fsync()	read()	
fgetc()	getgrgid()	sigsuspend()	

Dies hat folgende Auswirkungen:

– Wenn die Signalbehandlung der Prozessbeendigung dient, wird der Prozess beendet, und die Funktion kehrt nicht zurück.

– Wenn die Signalbehandlung dem Anhalten des Prozesses dient, wird er solange angehalten, bis er fortgesetzt oder beendet wird.

– Wenn für einen Prozess ein SIGCONT-Signal erzeugt wird, wird der Prozess an dem Punkt fortgesetzt, an dem der Prozess angehalten wurde.

– Wenn die zugehörige Signalbehandlung dem Aufrufen einer Signalbehandlungsfunktion dient, wird die Signalbehandlungsfunktion aufgerufen; in diesem Fall wird die ursprüngliche Funktion von dem Signal unterbrochen.

– Wenn die Signalbehandlungsfunktion eine return-Anweisung ausführt, verhält sich die unterbrochene Funktion genauso, wie es für diese Funktion beschrieben ist.

– Signale, die ignoriert werden, beeinflussen das Verhalten einer Funktion nicht.

– Signale, die blockiert werden, haben solange keinen Einfluss auf das Verhalten einer Funktion, bis sie zugestellt sind.

Returnwert 0 bei Erfolg.

-1 bei Fehler. errno wird gesetzt, um den Fehler anzuzeigen. Es wird keine neue Signalbehandlungsfunktion festgelegt.

Fehler `sigaction()` schlägt fehl, wenn gilt:

Erweiterung

`EFAULT` *act* und *oact* weisen über den zugewiesenen Adressraum des Prozesses hinaus. ❑

`EINVAL` *sig* ist keine gültige Signalnummer,
oder es wurde versucht, ein Signal abzufangen bzw. zu ignorieren, das nicht abgefangen bzw. ignoriert werden kann,
oder es wurde versucht, die Signalaktion `SIG_DFL` für ein Signal zu setzen, das nicht abgefangen und/oder ignoriert werden kann.

Hinweis `sigaction()` löst `signal()` ab und sollte vorzugsweise verwendet werden. Insbesondere sollten `sigaction()` und `signal()` nicht im selben Prozess für dasselbe Signal verwendet werden.

Wenn dasselbe Signal zweimal angemeldet wird, ist nur die letzte Anmeldung wirksam. Dies gilt insbesondere für aufeinander abgebildete Signale. So ist `SIGDVZ` auf `SIGFPE` abgebildet und `SIGTIM` auf `SIGVTALRM`. Wenn erst ein Signal eines solchen Paares angemeldet wird und dann das andere, gilt dies als Wiederholung desselben Signals.

Simultan nutzbare Funktionen verhalten sich so, wie es in diesem Handbuch beschrieben ist. Sie können ohne Einschränkung in Signalbehandlungsfunktionen verwendet werden. Anwendungen müssen dennoch alle Wirkungen dieser Funktionen berücksichtigen, die sich auf Datenstrukturen, Dateien und Prozesszustände beziehen. Insbesondere müssen die Autoren von Anwendungen die Einschränkungen von Interaktionen beachten, die sich bei der Unterbrechung von `sleep()` ergeben und die Interaktionen zwischen mehreren Dateideskriptoren für eine Dateibeschreibung.

Um Fehler zu vermeiden, die sich aus der Unterbrechung von nicht simultan nutzbaren Funktionsaufrufen ergeben, sollten Anwendungen die Aufrufe solcher Funktionen entweder durch das Blockieren der entsprechenden Signale oder durch die Verwendung von Semaphoren schützen. Dieses Handbuch spricht die allgemeineren Probleme der Synchronisation des Zugriffs auf simultan genutzte Datenstrukturen nicht an. Auch die unterbrechungssicheren Funktionen können zum Beispiel die externe Variable `errno` verändern; die Signalbehandlungsfunktion wiederum kann den Wert der Variablen sichern und wiederherstellen wollen. Selbstverständlich treffen dieselben Prinzipien auch auf simultan nutzbare Anwendungs-Funktionen und asynchronen Datenzugriff zu.

`siglongjmp()` ist nicht in der Liste der simultan nutzbaren Funktionen enthalten. Denn der Code, der nach `siglongjmp()` ausgeführt wird, kann beliebige unsichere Funktionen aufrufen, mit denselben Gefahren, die beim Aufruf dieser unsicheren Funktionen direkt aus der Signalbehandlungsfunktion auftreten. Anwendungen, die `siglongjmp()` aus Signalbehandlungsfunktionen heraus verwenden, benötigen einen rigorosen Schutz, um portabel zu sein. Viele andere Funktionen, die nicht in der Liste aufgeführt sind, sind traditionell so implementiert, dass sie `malloc()`, `free()` oder Funktionen aus `stdio.h` verwenden; diese Funktionen verwenden Datenstrukturen in einer nicht simultan nutzbaren Weise. Weil

jede Kombination verschiedener Funktionen, die eine gemeinsame Datenstruktur verwenden, Probleme bei der simultanen Nutzung verursachen können, definiert dieses Handbuch nicht das Verhalten, wenn eine unsichere Funktion aus einer Signalbehandlungsfunktion heraus aufgerufen wird, die eine unsichere Funktion unterbricht.

Wenn ein Signal auftritt, ohne dass `abort()`, `kill()` oder `raise()` aufgerufen wurden, ist das Verhalten nicht definiert, wenn die Signalbehandlungsfunktion eine X/Open-konforme Bibliotheksfunktion aufruft, die nicht in der obigen Tabelle steht, oder wenn auf ein Objekt im statischen Speicher zugegriffen wird und das keine statische Variable vom Typ `volatile sig_atomic_t` ist. Wenn ein derartiger Aufruf auf einen Fehler läuft, ist der Wert von `errno` nicht definiert.

Die Zuordnung zwischen den symbolischen Namen der Signalnummern und ihren numerischen Werten ist nicht standardisiert. Eine Anwendung ist nur portabel, wenn *sig* die symbolischen Namen verwendet.

Siehe auch `kill()`, `sigaddset()`, `sigdelset()`, `sigfillset()`, `sigemptyset()`, `sigismember()`, `sigprocmask()`, `sigsuspend()`, `signal.h`, Abschnitt „Signale“ auf Seite 115.

sigaddset - Signal einer Signalmenge hinzufügen

Syntax #include <signal.h>

int sigaddset(sigset_t **set*, int *sig*);

Beschreibung

`sigaddset()` fügt das Signal *sig* der Signalmenge hinzu, auf die *set* zeigt.

Returnwert

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

Fehler `sigaddset()` schlägt fehl, wenn gilt:

`EINVAL`	Der Wert von *sig* ist eine ungültige oder nicht unterstützte Signalnummer.

Hinweis Anwendungen sollten vor jeder anderen Verwendung eines Objekts vom Typ `sigset_t` `sigemptyset()` oder `sigfillset()` für dieses Objekt aufrufen. Wenn so ein Objekt nicht auf diese Weise initialisiert wird, aber trotzdem als Argument für `sigaction()`, `sigaddset()`, `sigdelset()`, `sigismember()`, `sigpending()` oder `sigprocmask()` verwendet wird, ist das Verhalten undefiniert.

Siehe auch `sigdelset()`, `sigemptyset()`, `sigfillset()`, `sigismember()`, `signal.h`.

sigaltstack - alternativen Stack eines Signals setzen/lesen

Syntax #include <signal.h>

int sigaltstack(const stack_t *ss, stack_t *oss);

Beschreibung

Mit `sigaltstack()` wird ein alternativer Stack definiert, in dem Signale bearbeitet werden können. Wenn *ss* ungleich null ist, wird ein Zeiger auf eine `stack_t` Struktur erwartet, die einen Stack beschreibt, auf dem die Signale bearbeitet werden können. Mit `sigaction` kann man festlegen, welche Signale auf dem alternativen Signalstack behandelt werden sollen. Für die Dauer der Ausführung der Signalbehandlungsroutine schaltet das System dann auf den Signalstack um.

Die Struktur `stack_t` enthält die folgenden Komponenten:

```
int     *ss_sp
long    ss_size
int     ss_flags
```

Ist *ss* nicht `NULL`, beschreibt die Struktur `stack_t` einen alternativen Signalstack, welcher nach Rückkehr von `sigaltstack()` wirksam wird. Die Komponenten `ss_sp` und `ss_size` bestimmen die Basis und die Größe des Stacks. Die Komponente `ss_flags` gibt den Zustand des neuen Stacks an und kann die folgenden Werte aufweisen:

`SS_DISABLE` Der Stack wird deaktiviert und `ss_sp` und `ss_size` werden ignoriert. Wenn `SS_DISABLE` nicht gesetzt ist, wird der Stack aktiviert.

Ist *oss* nicht `NULL`, so enthält die Struktur nach erfolgreicher Rückkehr aus `sigaltstack` die Beschreibung des alternativen Signalstacks, der vor dem Aufruf von `sigaltstack()` aktiv war. `ss_sp` und `ss_size` geben die Basis und die Größe des Stacks an.

Die `ss_flags`-Komponente gibt den Zustand des Stacks an. Dieser Zustand kann die folgenden Werte annehmen:

`SS_ONSTACK` Der Prozess wird momentan mit dem alternativen Signalstack ausgeführt. Versuche, den alternativen Signalstack während der Ausführung des Prozesses zu ändern, schlagen fehl.

`SS_DISABLE` Der alternative Signalstack ist momentan deaktiviert.

Der Wert `SIGSTKSZ` stellt die Anzahl der Bytes dar, welche im Allgemeinen für einen alternativen Stack notwendig sind. Der Wert `MINSIGSTKSZ` definiert dabei die minimale Stackgröße für eine Signalbehandlungsroutine. Bei der Berechnung der Stackgröße sollte das Programm noch diesen Minimalwert zusätzlich anlegen, um den Eigenbedarf des Betriebssystems zu berücksichtigen. Die Konstanten `SS_ONSTACK`, `SS_DISABLE`, `SIGSTKSZ` und `MINSIGSTKSZ` sind in `<signal.h>` definiert.

Returnwert

0	bei erfolgreicher Ausführung.
-1	bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `sigaltstack()` schlägt fehl, wenn gilt:

`EPERM`	Es *wurde versucht, einen aktiven Stack zu verändern (deaktivieren).*
`EINVAL`	Das Argument *ss* ist nicht null und die `ss_flags`-Komponente, auf die *ss* zeigt, enthält andere Flags als `SS_DISABLE`.
`ENOMEM`	Die Größe des alternativen Stackbereichs ist kleiner als `MINSIGSTKSZ`.

Hinweis Der folgende Programmauszug wird dazu verwendet, um einen alternativen Stackbereich zu allokieren:

```
if ((sigstk.ss_sp = (char *)malloc(SIGSTKSZ)) == NULL)
         /* Fehlerbehandlung */;

sigstk.ss_size = SIGSTKSZ;
sigstk.ss_flags = 0;
if (sigaltstack(&sigstk, (stack_t *)0) < 0)
      perror("sigaltstack");
```

Siehe auch `sigaction()`, `sigsetjmp()`, `signal.h`

sigdelset - Signal aus Signalmenge löschen

Syntax #include <signal.h>

int sigdelset(sigset_t **set*, int *sig*);

Beschreibung

`sigdelset()` löscht das Signal *sig* aus der Signalmenge, auf die *set* zeigt.

Returnwert

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

Fehler `sigdelset()` schlägt fehl, wenn gilt:

`EINVAL`	Der Wert von *sig* ist eine ungültige oder nicht unterstützte Signalnummer.

Hinweis Anwendungen sollten vor jeder anderen Verwendung eines Objekts vom Typ `sigset_t` `sigemptyset()` oder `sigfillset()` für dieses Objekt aufrufen. Wenn so ein Objekt nicht auf diese Weise initialisiert wird, aber trotzdem als Argument für `sigaction()`, `sigaddset()`, `sigdelset()`, `sigismember()`, `sigpending()` oder `sigprocmask()` verwendet wird, ist das Verhalten undefiniert.

Siehe auch `sigdelset()`, `sigemptyset()`, `sigfillset()`, `sigismember()`, `signal.h`.

sigemptyset - leere Signalmenge initialisieren

Syntax #include <signal.h>

int sigemptyset(sigset_t **set*);

Beschreibung

`sigemptyset()` initialisiert die Signalmenge, auf die *set* zeigt so, dass keines der vom System definierten Signale enthalten ist.

Returnwert

0	bei Erfolg.
-1	bei Fehler.

Hinweis Anwendungen sollten vor jeder anderen Verwendung eines Objekts vom Typ `sigset_t` `sigemptyset()` oder `sigfillset()` für dieses Objekt aufrufen. Wenn so ein Objekt nicht auf diese Weise initialisiert wird, aber trotzdem als Argument für `sigaction()`, `sigaddset()`, `sigdelset()`, `sigismember()`, `sigpending()` oder `sigprocmask()` verwendet wird, ist das Verhalten undefiniert.

Siehe auch `sigdelset()`, `sigemptyset()`, `sigfillset()`, `sigismember()`, `signal.h`.

sigfillset - Signalmenge mit allen Signalen initialisieren

Syntax #include <signal.h>

int sigfillset(sigset_t **set*);

Beschreibung

`sigfillset()` initialisiert die Signalmenge, auf die *set* zeigt so, dass alle vom System definierten Signale enthalten sind.

Returnwert

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

Fehler `sigfillset()` schlägt fehl, wenn gilt:

Erweiterung

`EFAULT`	*set* gibt eine ungültige Adresse an. ❑

Hinweis Anwendungen sollten vor jeder anderen Verwendung eines Objekts vom Typ `sigset_t` `sigemptyset()` oder `sigfillset()` für dieses Objekt aufrufen. Wenn so ein Objekt nicht auf diese Weise initialisiert wird, aber trotzdem als Argument für `sigaction()`, `sigaddset()`, `sigdelset()`, `sigismember()`, `sigpending()` oder `sigprocmask()` verwendet wird, ist das Verhalten undefiniert.

Siehe auch `sigdelset()`, `sigemptyset()`, `sigfillset()`, `sigismember()`, `signal.h`.

sighold, sigignore - Signal in der Signalmaske hinzufügen / SIG_IGN für ein Signal anmelden

Syntax #include <signal.h>

int sighold(int *sig*);
int sigignore(int *sig*);

Beschreibung

siehe `signal()`.

siginterrupt - Verhalten von Systemaufrufen bei Unterbrechungen ändern

Syntax #include <signal.h>

int siginterrupt(int *sig*, int *flag*);

Beschreibung

`siginterrupt()` wird verwendet, um das Neustartverhalten von Systemaufrufen zu ändern, wenn der Systemaufruf durch das angegebene Signal unterbrochen wurde. Die Funktion hat die selbe Wirkung, wie in folgender Implementierung gezeigt:

```
siginterrupt(int sig, int flag) {

    int ret;
    struct sigaction act;
    (void) sigaction(sig, NULL, &act);
    if (flag)
        act.sa_flags &=~SA_RESTART;

    else
        act.sa_fags |= SA_RESTART;

    ret=sigaction(sig, &act, NULL);
    return ret;
    }
```

Returnwert

0	bei erfolgreicher Ausführung.
-1	bei Fehler. `errno` wird gesetzt.

Fehler `siginterrupt()` schlägt fehl, wenn gilt:

`EINVAL`	Das Argument *sig* gibt eine ungültige Signalnummer an.

Hinweis `siginterrupt()` unterstützt Programme, die „historische“ Systemschnittstellen benutzen. Eine portierbare Anwendung sollte, wenn sie neu- bzw. umgeschrieben wird, an Stelle von `siginterrupt()` die Funktion `sigaction()` mit dem Flag `SA_RESTART` verwenden.

Siehe auch `sigaction()`, `signal.h`.

sigismember - auf Element einer Signalmenge prüfen

Syntax #include <signal.h>

int sigismember(const sigset_t *_set_, int _sig_);

Beschreibung

`sigismember()` prüft, ob das Signal *sig* in der Signalmenge enthalten ist, auf die *set* zeigt.

Returnwert

1	wenn das angegebene Signal bei erfolgreicher Beendigung in der angegebenen Signalmenge enthalten ist.
0	wenn das Signal bei erfolgreicher Beendigung in der angegebenen Signalmenge nicht enthalten ist.
-1	bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `sigismember()` schlägt fehl, wenn gilt:

`EINVAL`	Der Wert von *sig* ist eine ungültige oder nicht unterstützte Signalnummer.

Hinweis Anwendungen sollten vor jeder anderen Verwendung eines Objekts vom Typ `sigset_t` `sigemptyset()` oder `sigfillset()` für dieses Objekt aufrufen. Wenn so ein Objekt nicht auf diese Weise initialisiert wird, aber trotzdem als Argument für `sigaction()`, `sigaddset()`, `sigdelset()`, `sigismember()`, `sigpending()` oder `sigprocmask()` verwendet wird, ist das Verhalten undefiniert.

Siehe auch `sigdelset()`, `sigemptyset()`, `sigfillset()`, `sigismember()`, `signal.h`.

siglongjmp - nichtlokalen Sprung durch Signal ausführen

Syntax #include <setjmp.h>

void siglongjmp(sigjmp_buf *env*, int *val*);

Beschreibung

`siglongjmp()` stellt die Umgebung wieder her, die vom letzten `sigsetjmp`-Aufruf mit demselben `sigjmp_buf`-Argument im selben Prozess aufgerufen wurde. Wenn vorher `sigsetjmp()` nicht aufgerufen wird oder die Funktion, in der dieses Makro aufgerufen wurde in der Zwischenzeit beendet wurde, so ist das Verhalten nicht definiert.

Alle zugreifbaren Objekte besitzen dieselben Werte wie zu dem Zeitpunkt, als `siglongjmp()` aufgerufen wurde, mit der Ausnahme, dass die Werte von automatischen Objekten, die zwischen der Ausführung von `sigsetjmp()` und dem Aufruf von `siglongjmp()` geändert wurden, unbestimmt sind.

Da `siglongjmp()` den normalen Funktionsaufruf- und -Rückkehrmechanismus verwendet, läuft diese Funktion auch im Zusammenhang mit Unterbrechungen, Signalen und den damit zusammenhängenden Funktionen korrekt ab. Trotzdem gilt, dass das Verhalten nicht definiert ist, wenn `siglongjmp()` von einer verschachtelten Signalbehandlungs-Funktion aus aufgerufen wird (d.h. von einer Funktion die auf Grund eines Signals aus einer anderen Signalbehandlungsfunktion heraus aufgerufen wurde).

`siglongjmp()` stellt die gesicherte Signalmaske nur dann wieder her, wenn und nur wenn das Argument *env* durch einen Aufruf von `sigsetjmp()` mit einem Argument *savemask* ungleich 0 initialisiert wurde.

`siglongjmp()` ist nicht threadsicher. Das Ergebnis eines Aufrufs dieser Funktion ist undefiniert, wenn die Struktur `jmp_buf` nicht im aufrufenden Thread initialisiert wurde.

Returnwert 0 Nachdem `siglongjmp()` beendet ist, setzt die Ausführung des Programms so fort, als ob die zugehörige Ausführung des Makros `sigsetjmp()` soeben mit dem durch *val* angegebenen Wert beendet worden wäre. `siglongjmp()` kann `sigsetjmp()` nicht veranlassen, den Wert 0 zurückzugeben.

Hinweis wenn *val* gleich 0 ist, dann liefert das entsprechende Makro `sigsetjmp()` den Wert 1. Der Unterschied zwischen `setjmp()` oder `longjmp()` und `sigsetjmp()` oder `siglongjmp()` ist nur für solche Programme von Bedeutung, die die Funktionen `sigaction()`, `sigprocmask()` oder `sigsuspend()` verwenden.

Siehe auch `longjmp()`, `setjmp()`, `sigprocmask()`, `sigsetjmp()`, `sigsuspend()`, `setjmp.h`.

signal - Signalbehandlung ermitteln oder ändern

Syntax

```
#include <signal.h>

void (*signal(int sig, void (* func)(int)))(int);

int sighold(int sig);

int sigignore(int sig);

int sigpause(int sig);

int sigrelse(int sig);

void (*sigset(int sig, void (*disp)(int)))(int);
```

Beschreibung

`signal()` legt fest, wie der Empfang eines Signals zukünftig behandelt werden soll.

sig kann jedes Signal sein, das vom System definiert ist, außer `SIGKILL` und `SIGSTOP` (siehe `signal.h`).

func`()` definiert die Signalaktion. Folgende Werte sind möglich:

– `SIG_DFL` (voreingestellte Signalbehandlung)

– `SIG_IGN` (Ignorieren des Signals)

– die Adresse einer Signalbehandlungsfunktion
  In diesem Fall fügt das System das Signal *sig* der Signalmaske des aufrufenden Prozesses hinzu, bevor die Signalbehandlungsfunktion ausgeführt wird. Wenn die Ausführung der Signalbehandlungsfunktion beendet ist, stellt das System die Signalmaske des aufrufenden Prozesses wieder auf den Zustand um, der vor Empfang des Signals herrschte.

Wenn ein Signal auftritt und *func*`()` auf eine Funktion zeigt, werden nacheinander folgende Schritte ausgeführt:

1. Ein Äquivalent zu folgender `signal`-Funktion wird ausgeführt:

   `signal(`*sig*`, SIG_DFL);`

   Wenn in diesem Beispiel der Wert von *sig* `SIGILL` ist, wird `SIG_DFL` zurückgesetzt.

2. Ein Äquivalent der folgenden Funktion wird ausgeführt:

   `(*`*func*`)(`*sig*`);`

Die Signalbehandlungsfunktion *func*() kann durch eine return-Anweisung, eine abort-, exit- oder longjmp-Funktion beendet werden. Wenn *func*() eine return-Anweisung ausführt und der Wert von *sig* SIGFPE, SIGILL oder SIGDVZ ist, ist das Verhalten nicht definiert. Ansonsten setzt das Programm die Ausführung an dem Punkt fort, an dem es unterbrochen wurde.

Wenn ein Signal auftritt, ohne dass abort(), kill() oder raise() aufgerufen wurden, ist das Verhalten nicht definiert, wenn die Signalbehandlungsfunktion eine X/Open-konforme Bibliotheksfunktion aufruft, die nicht in der obigen Tabelle steht, oder wenn auf ein Objekt im statischen Speicher zugegriffen wird und das keine statische Variable vom Typ volatile sig_atomic_t ist. Wenn ein derartiger Aufruf auf einen Fehler läuft, ist der Wert von errno nicht definiert.

Bei Programmstart wird ein Äquivalent der folgenden Funktion für einige Signale ausgeführt:

signal(*sig,* SIG_IGN);

Ein Äquivalent der folgenden Funktion wird für alle anderen Signale ausgeführt (siehe exec):

signal(*sig,* SIG_DFL);

Die Funktionen sigset(), sighold(), sigignore(), sigpause() und sigrelse() erlauben Applikationsprozessen das vereinfachte Verwalten von Signalen.

sigset() wird verwendet, um Signalbehandlungen zu verändern. *sig* gibt dabei das Signal an, welches jedes außer SIGKILL und SIGSTOP sein darf. *disp* definiert die Behandlung des Signals, welches SIG_DFL, SIG_IGN oder die Adresse einer Signalbehandlungsroutine sein darf. Wird sigset() verwendet und ist *disp* die Adresse einer Signalbehandlungsroutine, fügt das System das Signal *sig* der Signalmaske des aufrufenden Prozesses hinzu, bevor die Signalbehandlungsroutine ausgeführt wird. Ist die Ausführung der Signalbehandlungsroutine beendet, stellt das System die Signalmaske des aufrufenden Protesses wieder auf den Zustand, der vor dem Empfang des Signals herrschte. Wird sigset() benutzt und ist *disp* gleich SIG_HOLD, so wird *sig* zur Signalmaske des aufrufenden Prozesses hinzugefügt, und die Signalbehandlung bleibt unverändert.

sighold() fügt *sig* der Signalmaske des aufrufenden Prozesses hinzu.

sigrelse() entfernt *sig* von der Signalmaske des aufrufenden Prozesses.

sigignore() stellt die Behandlung von *sig* auf SIG_IGN.

sigpause() entfernt *sig* von der Signalmaske des aufrufenden Prozesses und deaktiviert den aufrufenden Prozess, bis ein Signal empfangen wird.

Wird eine der obigen Funktionen verwendet, um die Behandlung von SIGCHLD auf SIG_IGN zu setzen, so erzeugen die Sohnprozesse des aufrufenden Prozesses keine Zombie-Prozesse, wenn sie beendet werden. Wenn der aufrufende Prozess nacheinander auf seine

Sohnprozesse wartet, blockiert er, bis alle seine Sohnprozesse terminiert sind. Dann wird der Wert -1 zurückgeliefert und `errno` enthält die Fehlernummer `ECHILD` (siehe `wait()`, `waitid()`, `waitpid()`).

Returnwert Wert von *func*`()` bei erfolgreicher Beendigung.

`SIG_ERR` bei Fehler, z.B. wenn *sig* keine gültige Signalnummer ist oder *func*`()` auf eine unzulässige Adresse zeigt. `errno` wird gesetzt, um den Fehler anzuzeigen.

`SIG_HOLD` von `sigset()` bei Erfolg geliefert, wenn das Signal blockiert wurde. Wenn es nicht blockiert wurde, liefert `sigset()` die vorherige Behandlung zurück.

`SIG_ERR` bei Fehler von `sigset()`. `errno` enthält die entsprechende Fehlernummer.

Alle anderen Funktionen liefern bei Erfolg null zurück. Bei Fehler liefern sie -1 und setzen `errno`.

Fehler `signal()` schlägt fehl, wenn gilt:

`EINVAL` *sig* ist eine ungültige Signalnummer,
oder es wurde versucht, ein Signal abzufangen, das nicht abgefangen werden kann,
oder ein Signal zu ignorieren, das nicht ignoriert werden kann,
oder es wurde versucht, die Aktion auf `SIG_DFL` zu setzen bei einem Signal, dass weder abgefangen noch ignoriert werden kann.

BS2000
`EFAULT` Unzulässige Adresse. ❑

`sigset()`, `sighold()`, `sigrelse()`, `sigignore()` und `sigpause()` schlagen fehl, wenn gilt:

`EINVAL` *sig* ist eine ungültige Signalnummer oder bei `sigset()` und `sigignore()` wurde der Versuch gemacht, ein Signal abzufangen, das nicht abgefangen werden kann oder ein Signal zu ignorieren, das nicht ignoriert werden kann.

Hinweis `sigaction()` bietet einen verständlicheren und verlässlicheren Mechanismus für die Signalsteuerung als `signal()`. Neue Anwendungen sollten daher `sigaction()` benutzen.

`sighold()` in Verbindung mit `sigrelse()` oder `sigpause()` kann dazu verwendet werden, kritische Programmbereiche zu erstellen, in denen der Empfang eines Signals zeitweise abgeschaltet wird.

Die Funktion `sigsuspend()` kann anstatt `sigpause()` verwendet werden, um die Portabilität zu erhöhen.

Siehe auch `exec`, `pause()`, `sigaction()`,, `waitid()`, `signal.h`.

signgam - Variable für Vorzeichen von lgamma

Syntax #include <math.h>

extern int *signgam*;

Beschreibung

Siehe `lgamma()`.

sigpause - Signal aus Signalmaske entfernen und Prozess deaktivieren

Syntax #include <signal.h>

int sigpause(int *sig*);

Beschreibung

Siehe `signal()`.

Hinweis Werden Threads verwendet, so wirkt sich die Funktion auf den Prozess oder auf einen Thread wie folgt aus: `sigpause()` entfernt ein Signal aus der Signalmaske und suspendiert den Thread.

sigpending - blockierte Signale ermitteln

Syntax #include <signal.h>

int sigpending(sigset_t **set*);

Beschreibung

`sigpending()` speichert die Menge der Signale, deren Zustellung blockiert ist und die für den aufrufenden Prozess anstehen, in dem Objekt ab, auf das *set* zeigt.

Returnwert

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

Fehler `sigpending()` schlägt fehl, wenn gilt:

Erweiterung	
`EFAULT`	*set* ist ein ungültiger Zeiger. ❑

Siehe auch `sigaddset()`, `sigdelset()`, `sigemptyset()`, `sigfillset()`, `sigismember()`, `sigprocmask()`, `signal.h`.

sigprocmask - blockierte Signale ermitteln oder ändern

Syntax #include <signal.h>

int sigprocmask(int *how*, const sigset_t **set*, sigset_t **oset*);

Beschreibung

`sigprocmask()` erlaubt dem aufrufenden Prozess, seine Signalmaske, d.h. die Menge der blockierten Signale, zu überprüfen oder zu ändern.

Wenn *set* ungleich dem Nullzeiger ist, zeigt es auf eine Signalmenge, die verwendet wird, um die augenblicklich blockierte Signalmenge zu ändern.

how gibt an, auf welche Weise die Signalmenge geändert werden soll. Es kann einen der folgenden Werte annehmen (siehe auch `signal.h`):

`SIG_BLOCK` Die Ergebnismenge besteht aus der Vereinigung der aktuellen und der durch *set* angegebenen Signalmenge.

`SIG_UNBLOCK` Die Ergebnismenge besteht aus der Schnittmenge der aktuellen und des Komplements der durch *set* angegebenen Signalmenge.

`SIG_SETMASK` Die Ergebnismenge entspricht der durch *set* angegebenen Signalmenge.

Wenn *oset* kein Nullzeiger ist, wird die alte Maske in dem Bereich abgespeichert, auf den *oset* zeigt.

Wenn *set* ein Nullzeiger ist, spielt der Wert des Arguments *how* keine Rolle und die Signalmaske des Prozesses bleibt unverändert; daher kann der Aufruf verwendet werden, um die derzeit blockierten Signale abzufragen.

Wenn es anstehende, nichtblockierte Signale nach einem Aufruf von `sigprocmask()` gibt, wird wenigstens eines dieser Signale zugestellt, bevor der Aufruf von `sigprocmask()` zurückkehrt.

Signale, die nicht ignoriert werden können, können auch nicht blockiert werden (siehe `signal.h`). Dies wird durch das System sichergestellt, ohne dass ein Fehler angezeigt wird.

Wenn eines der Signale `SIGFPE`, `SIGILL` oder `SIGSEGV` erzeugt wird, während es blockiert ist, ist das Ergebnis undefiniert, es sei denn, das Signal wurde durch `kill()` oder `raise()` erzeugt.

Wenn `sigprocmask()` fehlschlägt, wird die Signalmaske des Prozesses nicht geändert.

`sigprocmask()` ist nicht threadsicher. Verwenden Sie bei Bedarf die Funktion `pthread_sigmask()`.

Returnwert

0	bei Erfolg.
-1	bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen. Die Signalmaske des Prozesses wird nicht geändert.

Fehler `sigprocmask()` schlägt fehl, wenn gilt:

`EINVAL`	Der Wert von *how* entspricht keinem zulässigen Wert.
Erweiterung `EFAULT`	*set* oder *oset* weisen über den zugewiesenen Adressraum des Prozesses hinaus. ❑

Siehe auch `kill()`, `raise()`, `sigaction()`, `sigaddset()`, `sigdelset()`, `sigemptyset()`, `sigfillset()`, `sigismember()`, `sigpending()`, `sigsuspend()`, `signal.h`.

sigrelse - Signal aus Signalmaske entfernen

Syntax #include <signal.h>

int sigrelse(int *sig*);
void (*sigset(int *sig*, void (**disp*) (int))) (int);

Beschreibung

Siehe signal().

sigset - Signalbehandlung ändern

Syntax #include <signal.h>

void (*sigset(int *sig*, void (**func*)(int)))(int);

Beschreibung

sigset() wird verwendet, um die Signalbehandlung zu ändern.

siehe signal().

Hinweis sigset() ist nicht threadsicher.

sigsetjmp - Marke für nichtlokalen Sprung durch Signal setzen

Syntax #include <setjmp.h>

int sigsetjmp(sigjmp_buf *env*, int *savemask*);

Beschreibung

`sigsetjmp()` ist als Makro implementiert und sichert seine Aufrufumgebung in sein Argument *env* für eine spätere Benutzung durch die Funktion `siglongjmp()`.

Wenn der Wert von *savemask* ungleich 0 ist, sichert `sigsetjmp()` auch die aktuelle Signalmaske des Prozesses als einen Teil der Aufrufumgebung. Bei der Verwendung von `setjmp()` ginge diese verloren.

Alle zugreifbaren Objekte besitzen die Werte, die sie zum Zeitpunkt des Aufrufs von `longjmp()` besaßen, mit Ausnahme der Werte von automatischen Objekten. Diese sind unter folgenden Bedingungen undefiniert:

– Sie sind lokal zu der Funktion, die den entsprechenden `setjmp`-Aufruf enthält.

– Sie sind nicht vom Typ `volatile`.

– Sie wurden zwischen dem `setjmp`- und dem `longjmp`-Aufruf geändert.

`sigsetjmp()` darf nur in einem der folgenden Zusammenhänge aufgerufen werden:

– als vollständiger Bedingungsausdruck einer Auswahl- oder Schleifenanweisung, z.B.:

```
if (sigsetjmp(env, mask)) ...
```

– als Operand eines Vergleichsoperators, wobei der andere Operand ein konstanter ganzzahliger Ausdruck und der Gesamtausdruck der vollständige Bedingungsausdruck einer Auswahl- oder Schleifenanweisung ist, z.B:

```
if (sigsetjmp(env, mask)==0) ...
```

– als Operand des einstelligen Operators !, wobei der Gesamtausdruck der vollständige Bedingungsausdruck einer Auswahl- oder Schleifenanweisung ist, z.B:

```
if (!sigsetjmp(env, mask) ...
```

– als vollständiger Ausdruck einer Ausdrucksanweisung (ggf. umgewandelt in den Typ `(void)`, z.B:

```
(void) sigsetjmp(env, mask);
```

Werden Threads verwendet, so wirkt sich die Funktion auf den Prozess oder auf einen Thread wie folgt aus: Wenn der Wert von *savemask* ungleich 0 ist, sichert `sigsetjmp()` auch die aktuelle Signalmaske des aufrufenden Threads als einen Teil der Aufrufumgebung.

Returnwert

0	wenn die Rückkehr von einer unmittelbaren Ausführung von `sigsetjmp()` erfolgt.
≠ 0	wenn die Rückkehr von einem `siglongjmp()`-Aufruf erfolgt.

Hinweis

Der Unterschied zwischen `setjmp()/longjmp()` und `sigsetjmp() /siglongjmp()` ist nur für solche Programme von Bedeutung, die `sigaction()`, `sigprocmask()` oder `sigsuspend()` verwenden.

Siehe auch

`siglongjmp()`, `signal()`, `sigprocmask()`, `sigsuspend()`, `setjmp.h`, Abschnitt „Signale“ auf Seite 115.

sigstack - alternativen Stack für Signal setzen oder abfragen

```
#include <signal.h>

int sigstack (struct sigstack *ss, struct sigstack *oss);
```

Beschreibung

Mit `sigstack()` können Benutzer einen alternativen Stack definieren, der als Signal-Stack bezeichnet wird und in dem die Signale verarbeitet werden. Wenn durch die Aktion eines Signals angezeigt wird, dass die Bearbeitungsroutine in einem Signal-Stack ausgeführt werden soll (angegeben mit einem Aufruf von `sigaction()`), prüft das System, ob der Prozess derzeit in diesem Stack ausgeführt wird. Wird der Prozess nicht im Signal-Stack ausgeführt, schaltet das System so lange in den Signal-Stack um, bis die Routine zur Signalbearbeitung beendet ist.

Ein Signal-Stack wird durch eine `sigstack`-Struktur angegeben, die folgende Elemente enthält:

```
char   *ss_sp;          /* pointer of signal stack */
int    ss_onstack;      /* current status */
```

ss_sp ist die Anfangsadresse des Stacks. Ist das Feld *ss_onstack* ungleich null, so soll der Signal-Stack aktiviert werden.

Ist *ss* kein Nullzeiger, setzt `sigstack()` den Status des Signal-Stack auf den Wert in der `sigstack`-Struktur, auf den *ss* zeigt. Die Länge des Stack muss mindestens `SIGSTKSZ` Bytes betragen. Ist *ss_onstack* nicht null, geht das System davon aus, dass der Prozess im Signal-Stack ausgeführt wird. Ist *ss* ein Nullzeiger, bleibt der Status des Signal-Stack unverändert. Ist *oss* kein Nullzeiger, wird der aktuelle Status des Signal-Stack in der `sigstack`-Struktur, auf die *oss* zeigt, gespeichert.

Returnwert

0	bei erfolgreicher Ausführung.
-1	bei Fehler. Es wird `errno` gesetzt, um den Fehler anzuzeigen.

Fehler `sigstack()` schlägt fehl, wenn gilt:

`EPERM`	Es wurde der Versuch gemacht, einen aktiven Stack zu verändern

Hinweis

Signal-Stacks werden nicht automatisch vergrößert, wie dies bei normalen Stacks der Fall ist. Wenn der Signal-Stack überläuft, können daher unerwartete Ergebnisse auftreten.

Eine portierbare Anwendung sollte `sigaltstack()` statt `sigstack()` verwenden.

Programme sollten eine Signalbehandlungs-Routine nicht mit `longjmp()` beenden, wenn diese in einem Stack abläuft, der mit `sigstack()` eingerichtet wurde. Unter Umständen wird dieser Stack für die weitere Verwendung unbrauchbar. Es wird daher empfohlen für diesen Fall die Funktionen `siglongjmp()`, `setcontext()` oder `swapcontext()` zu verwenden.

sigsuspend - auf Signal warten

Syntax

```
#include <signal.h>

int sigsuspend(const sigset_t *sigmask);
```

Beschreibung

`sigsuspend()` ersetzt die aktuelle Signalmaske des Prozesses durch die Signalmenge, auf die *sigmask* zeigt, und blockiert den Prozess solange, bis ein Signal zugestellt wird, dessen Signalaktion entweder die Ausführung einer Signalbehandlungsfunktion oder der Prozessabbruch ist.

Wenn die Signalaktion der Prozessabbruch ist, kehrt `sigsuspend()` nicht zurück.

Wenn die Signalaktion die Ausführung einer Signalbehandlungsfunktion ist, kehrt die Funktion nach Ende dieser Signalbehandlungsfunktion zurück, wobei die Signalmaske so wiederhergestellt wird, wie sie vor dem Aufruf von `sigsuspend()` eingestellt war.

Signale, die nicht ignoriert werden können, können auch nicht blockiert werden (siehe `signal.h`). Dies wird durch das System sichergestellt, ohne dass ein Fehler angezeigt wird.

Werden Threads verwendet, so wirkt sich die Funktion auf den Prozess oder auf einen Thread wie folgt aus: `sigsuspend()` ersetzt die aktuelle Signalmaske des aufrufenden Threads mit der angegebenen Signalmenge und suspendiert dann den Thread.

Returnwert

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

Da `sigsuspend()` die Prozessausführung solange unterbricht, bis sie von einem Signal unterbrochen wird, kann `sigsuspend()` keinen Returnwert für erfolgreiche Beendigung haben.

Fehler

`sigsuspend()` schlägt fehl, wenn gilt:

`EINTR` Ein Signal wurde vom aufrufenden Prozess abgefangen und die Steuerung wird von der Signalbehandlungs-Funktion zurückgegeben.

Erweiterung

`EFAULT` *sigmask* weist über den zugewiesenen Adressraum des Prozesses hinaus. ❑

Siehe auch

`pause()`, `sigaction()`, `sigaddset()`, `sigdelset()`, `sigemptyset()`, `sigfillset()`, `signal.h`.

sin - Sinus berechnen

Syntax #include <math.h>

double sin(double *x*);

Beschreibung

`sin()` berechnet für die Gleitkommazahl *x*, die den Winkel im Bogenmaß angibt, die trigonometrische Funktion Sinus.

Returnwert sin(*x*) bei Erfolg (Gleitkommazahl im Intervall [-1.0, +1.0]).

Siehe auch `acos()`, `asin()`, `atan()`, `atan2()`, `cos()`, `sinh()`, `tan()`, `math.h`.

sinh - Sinus hyperbolicus berechnen

Syntax #include <math.h>

double sinh(double *x*);

Beschreibung

`sinh()` berechnet den Sinus hyperbolicus für die Gleitkommazahl *x*.

Returnwert sinh(*x*) bei Erfolg.

`+HUGE_VAL` bei Überlauf. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `sinh()` schlägt fehl, wenn gilt:

`ERANGE` Der Wert von *x* verursacht einen Überlauf.

Siehe auch `acos()`, `asin()`, `atan()`, `cos()`, `cosh()`, `sin()`, `tanh()`, `math.h`.

sleep - Prozess für festgesetzte Zeitspanne anhalten

Syntax #include <unistd.h>

unsigned int sleep(unsigned int *seconds*);

Beschreibung

`sleep()` sorgt dafür, dass der aktuelle Prozess solange angehalten wird, bis entweder die durch *seconds* angegebene Zeit von Echtzeit-Sekunden vergangen ist oder bis ein Signal an den aufrufenden Prozess zugestellt wird, dessen Signalaktion entweder eine Signalbehandlungsfunktion oder die Prozessbeendigung ist. Die Anhaltezeit kann aus Prioritätsgründen des Systems länger als *seconds* sein.

Wenn während der Ausführung von `sleep()` das Signal `SIGALRM` für den aufrufenden Prozess erzeugt und das `SIGALRM`-Signal ignoriert oder blockiert wird, ist nicht definiert, ob `sleep()` zurückkehrt, wenn das Signal bearbeitet wird.

Wenn das Signal blockiert ist, ist undefiniert, ob es auch noch nach der Rückkehr von `sleep()` ansteht oder ob es verworfen wird.

Wenn während der Ausführung von `sleep()` das Signal `SIGALRM` für den aufrufenden Prozess erzeugt wird, wenn dieses Signal nicht das Ergebnis eines vorhergehenden Aufrufs der Funktion `alarm()` ist und wenn das Signal `SIGALRM` nicht blockiert oder ignoriert wird, ist es undefiniert, ob das Signal noch eine andere Wirkung hat, als `sleep()` zur Rückkehr zu zwingen.

Wenn eine Signalbehandlungsfunktion `sleep()` unterbricht, ist das Ergebnis unter folgenden Bedingungen undefiniert:

– wenn der Zeitpunkt, zu dem ein Signal `SIGALRM` erzeugt werden soll, ermittelt oder verändert wird

– wenn die dem Signal `SIGALRM` zugeordnete Signalaktion verändert wird

– wenn verändert wird, ob das Signal `SIGALRM` von der Zustellung blockiert werden soll

Wenn eine Signalbehandlungsfunktion `sleep()` unterbricht und `siglongjmp()` oder `longjmp()` aufruft, um eine Umgebung wiederherzustellen, die vor dem Aufruf von `sleep()` gesichert wurde, sind sowohl die dem Signal `SIGALRM` zugeordnete Signalaktion als auch die Zeit, zu der dieses Signal ausgelöst werden soll, undefiniert. Es ist auch nicht definiert, ob das Signal `SIGALRM` blockiert wird, wenn die Signalmaske des Prozesses als Teil der Umgebung nicht wiederhergestellt wird (siehe auch `sigsetjmp()`).

Werden Threads verwendet, so wirkt sich die Funktion auf den Prozess oder auf einen Thread wie folgt aus: `sleep()` bewirkt, dass der aktuelle Thread suspendiert wird, bis eine angegebene Zeit abgelaufen ist oder ein Signal an den Thread zugestellt wurde.

Returnwert 0 wenn `sleep()` zurückkehrt, weil die eingestellte Zeit abgelaufen ist.

seconds minus schlafend verbrachte Zeit in Sekunden
wenn die Funktion `sleep()` zurückkehrt, weil sie durch Zustellung eines Signals vorzeitig beendet wurde.

`sleep()` ist immer erfolgreich.

Hinweis Obwohl das Programm mit `sleep()` angehalten wird, läuft die Zeit für eine zuvor gestellte Alarmuhr (mit `alarm()`) weiter. Dies hat folgende Auswirkungen:

1. Die vorher eingestellte Alarmzeit sei kleiner als die `sleep`-Zeit, z.B.:

   ```
   alarm(2);
   sleep(30);
   ```

   Nach Ablauf von zwei "Schlaf"-Sekunden wird der Alarm ausgelöst und der `sleep`-Aufruf beendet.

2. Die vorher eingestellte Alarmzeit sei größer als die `sleep`–Zeit, z.B.:

   ```
   alarm(30);
   sleep(5);
   ```

   Die Zeit der Alarmuhr läuft um 5 "schlafende" Sekunden weiter. Die Alarmuhr steht nach dem `sleep`-Aufruf auf 25.

Die Zeit, die das Programm tatsächlich angehalten wird, kann auch noch aus folgenden Gründen von *sec* abweichen:

– sie kann bis zu einer Sekunde kürzer sein, weil das „Aufwecken“ in festen 1-Sekunden-Intervallen stattfindet,

– sie kann aus Prioritätsgründen beliebig länger sein, weil das System „Wichtigeres“ zu tun hat.

Siehe auch `alarm()`, `pause()`, `sigaction()`, `unistd.h`.

sprintf - formatiert in Zeichenkette schreiben

Syntax #include <stdio.h>

int sprintf(char *s, const char *format[, arglist]);

Beschreibung

Siehe `fprintf()`.

sqrt - Quadratwurzel berechnen

Syntax #include <math.h>

double sqrt(double *x*);

Beschreibung

`sqrt()` berechnet die Quadratwurzel zu einer nichtnegativen Gleitkommazahl *x*.

Returnwert

`sqrt`(*x*)	falls *x* > = 0 ist.
0	falls *x* negativ ist. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `sqrt()` schlägt fehl, wenn gilt:

`EDOM`	Der Wert von *x* ist negativ.

Siehe auch `exp()`, `hypot()`, `log()`, `log10()`, `pow()`, `sinh()`, `math.h`.

srand - Pseudo-Zufallszahlen (int) mit Startwert generieren

Syntax #include <stdlib.h>

void srand(unsigned int *seed*);

Beschreibung

Mit `srand()` wird der Zufallsgenerator initialisiert, der von `rand()` aufgerufen wird.

seed ist eine beliebige ganze Zahl, die den Zufallsgenerator auf eine Zufallszahl setzt. Die Zahl 1 setzt den Zufallsgenerator auf seine voreingestellte Startzahl.

Siehe auch `rand()`.

srandom - Pseudo-Zufallszahlen

Syntax #include <stdlib.h>

void srandom(unsigned int *seed*);

Beschreibung

siehe `initstate()`.

srand48 - Startwert (double) für Pseudo-Zufallszahlen setzen

Syntax #include <stdlib.h>

void srand48(long int *seedval*);

Beschreibung

Siehe `drand48()`.

sscanf - formatiert aus Zeichenkette lesen

Syntax #include <stdio.h>

int sscanf(const char **s*, const char **format*[, *arglist*]);

Beschreibung

Siehe `fscanf()`.

stat - Dateistatus abfragen

Name **stat, stat64**

Syntax
```
#include <sys/stat.h>
#include <sys/types.h>

int stat (const char *path, struct stat *buf);
int stat64 (const char *path, struct stat64 *buf);
```

Beschreibung

`stat()` erhält Informationen über die angegebene Datei und schreibt diese Informationen in die Struktur, auf die *buf* zeigt.

path zeigt auf einen Pfadnamen, der die Datei benennt. Das Lese-, Schreib- oder Ausführungrecht dieser Datei wird nicht benötigt. Es müssen jedoch alle Dateiverzeichnisse, die im Pfadnamen aufgeführt werden, durchsuchbar sein.

buf ist ein Zeiger auf eine Struktur vom Typ *stat*, der in der Include-Datei `sys/stat.h` definiert ist. In diese Struktur werden die Informationen zur Datei eingetragen.

`stat()` aktualisiert alle zeitbezogenen Strukturkomponenten so, wie es im Fachwortverzeichnis unter „Dateizeiten-Änderung“ beschrieben wird, bevor diese Komponenten in die Struktur `stat` geschrieben werden.

Die Strukturkomponenten `st_mode`, `st_ino`, `st_dev`, `st_uid`, `st_gid`, `st_atime`, `st_ctime` und `st_mtime` haben danach sinnvolle Werte für alle Dateitypen. Der Wert der Strukturkomponente `st_nlink` wird auf die Anzahl der Verweise auf die Datei gesetzt.

Es besteht kein funktionaler Unterschied zwischen `stat()` und `stat64()`, außer dass `stat64()` eine `stat64`- Struktur verwendet.

Zum Inhalt der `stat`-Struktur, auf die von *buf* gewiesen wird, gehören folgende Elemente:

```
mode_t    st_mode;    /* Dateimodus (siehe mknod()) */
ino_t     st_ino;     /* Dateikennziffer (i-Node) */
dev_t     st_dev;     /* Gerätekennung, die einen
                         Verzeichniseintrag für diese Datei enthält */
dev_t     st_rdev;    /* Gerätekennung, nur für zeichen- oder
                         blockorientierte Gerätedateien definiert */
nlink_t   st_nlink;   /* Anzahl der Verweise */
uid_t     st_uid;     /* Benutzerkennung des Dateibesitzers */
gid_t     st_gid;     /* Gruppenkennung des Dateibesitzers */
off_t     st_size;    /* Dateigröße in Bytes */
time_t    st_atime;   /* Zeit des letzten Zugriffs */
time_t    st_mtime;   /* Zeit der letzten Datenänderung */
time_t    st_ctime;   /* Zeit der letzten Änderung des Dateistatus
                         Die Zeit wird in Sekunden gemessen ab dem
                         1. Januar 1970, 00:00:00 Uhr */
```

Erweiterung

```
long      st_blksize; /* Bevorzugte Ein-/Ausgabe-Blockgröße */
blkcnt_t  st_blocks;  /* Anzahl zugewiesener st_blksize-Blöcke */ q
```

Die Struktur `stat64` ist wie die von `stat()` definiert, mit Ausnahme folgender Komponenten:

```
ino64_t    st_ino
off64_t    st_size und
blkcnt64_t st_blocks
```

Die Elemente der Struktur haben folgende Bedeutung:

`st_mode` Der Modus der Datei ist im Systemaufruf `mknod()` beschrieben.

`st_ino` kennzeichnet die Datei im gegebenen Dateisystem eindeutig. Das Paar `st_ino` und `st_dev` kennzeichnet normale Dateien eindeutig.

`st_dev` kennzeichnet das Dateisystem, in dem die Datei liegt, eindeutig.

`st_rdev` darf nur von Verwaltungskommandos benutzt werden. Dieses Kennzeichen ist nur für block- oder zeichenorientierte Dateien gültig und hat nur in dem System eine Bedeutung, in dem die Datei konfiguriert wurde.

`st_nlink` darf nur von Verwaltungskommandos benutzt werden.

`st_uid` Benutzernummer des Eigentümers der Datei.

`st_gid` Gruppennummer der Gruppe, der die Datei zugeordnet ist.

`st_size` Für normale Dateien ist dies die Größe der Datei in Byte. Für block- oder zeichenorientierte Dateien ist dieses nicht definiert.

`st_atime` Uhrzeit, zu der zuletzt auf die Dateidaten zugegriffen wurde. Wird von folgenden Systemaufrufen geändert: `creat()`, `mknod()`, `utime()` und `read()`.

`st_mtime` Uhrzeit, zu der Daten zuletzt geändert wurden. Wird von folgenden Systemaufrufen geändert: `creat()`, `mknod()`, `utime()` und `write()`.

`st_ctime` Uhrzeit, zu der der Dateistatus zuletzt geändert wurde. Wird von folgenden Systemaufrufen geändert: `chmod()`, `chown()`, `creat()`, `link()`, `mknod()`, `unlink()`, `utime()` und `write()`.

Erweiterung

`st_blksize` Ein Hinweis auf die 'beste' Größe einer Einheit für Ein/Ausgabe-Operationen. Dieses Feld ist für block- oder zeichenorientierte Gerätedateien nicht definiert.

`st_blocks` Die Gesamtanzahl von physikalischen Blöcken der Größe 512 Byte, die zurzeit auf der Platte belegt ist. Dieses Feld ist für block- oder zeichenorientierte Gerätedateien nicht definiert.

BS2000

Bei BS2000-Dateien werden folgende Elemente der `stat`-Struktur gesetzt:

mode_t *st_mode* Dateimodus, der Zugriffsrechte und Dateityp beinhaltet.

Zugriffsrechte: Hier wird das Basic ACL auf die Dateischutzbits abgebildet. Die Schutzbits sind alle 0, wenn die Datei keinen Basic ACL Schutz hat.
Dateityp: Einführung eines neuen Dateityps S_IFDVSBS2=X'40000000'. Dieser Typ ist allerdings nicht disjunkt zu S_IFPOSIXBS2. Abgefragt kann mit dem Makro S_ISDVSBS2(*mode*) werden.

time_t *st_atime* Zeitpunkt des letzten Zugriffs wie im BS2000 üblich (last access time), aber in Sekunden seit dem 1.1.1970 UTC.

time_t *st_mtime* Zeitpunkt der letzten Änderung (last modification time).

time_t *st_ctime* Zeitpunkt der Erzeugung (creation time).

long *st_blksize* Blockgröße, 2k (d.h. 1 PAM Page).

long *st_blocks* Anzahl der von der Datei belegten Blöcke auf der Platte.

dev_t *st_dev* enthält die 4 Byte lange `catid`.

Die beiden hintereinander liegenden Felder

uid_t *st_uid* und

gid_t *st_uid* enthalten die 8 Byte lange BS2000-Userid.

Alle anderen Felder werden auf 0 gesetzt.

Returnwert

0	bei Erfolg.
-1	bei Fehler. Für POSIX-Dateien wird `errno` gesetzt, um den Fehler anzuzeigen.

Hinweis `stat()` wird jetzt auch für BS2000-Dateien ausgeführt.

Fehler `stat()` und `stat64()` schlagen fehl, wenn gilt:

`EACCES` Für eine Komponente des Pfades besteht kein Durchsuchrecht.

Erweiterung

EFAULT	*buf* oder *path* weisen auf eine ungültige Adresse.
EINTR	Ein Signal wurde während des Systemaufrufs stat() oder lstat() abgefangen.
EINVAL	Die genannte Datei existiert nicht oder das Argument *path* zeigt auf eine leere Zeichenkette.
EIO	Beim Lesen des Dateisystems trat ein Ein-/Ausgabefehler auf.
ELOOP	Bei der Übersetzung von *path* wurden zuviele symbolische Verweise angetroffen.
EMULTIHOP	Komponenten von *path* erfordern den Sprung auf mehrere ferne Rechner, aber der Dateisystemtyp erlaubt dies nicht.
ENAMETOOLONG	Die Länge von *path* überschreitet {PATH_MAX} oder eine Pfadnamen-Komponente ist länger als {NAME_MAX}, während {_POSIX_NO_TRUNC} aktiv ist.
ENOLINK	*path* weist auf einen fernen Rechner, zu dem keine aktive Verbindung existiert.
ENOENT	Die angegebene Datei ist nicht vorhanden oder ist der Null-Pfadname.
ENOTDIR	Eine Komponente des Pfades ist kein Dateiverzeichnis.
EOVERFLOW	Eine Komponente ist zu groß, um in der Struktur, auf die *buf* zeigt, gespeichert zu werden.

Siehe auch chmod(), chown(), creat(), fstat(), lstat(), link(), mknod(), sys/stat.h, sys/types.h.

statvfs - Dateisystem-Informationen lesen

Name **statvfs, statvfs64**

Syntax
```
#include <sys/statvfs.h>
#include <sys/types.h>

int statvfs (const char *path, struct statvfs *buf);
int statvfs64 (const char *path, struct statvfs64 *buf);
```

Beschreibung

Siehe `fstatvfs()`.

_ _STDC_ _ - Makro für ANSI-Konformität

Syntax _ _STDC_ _

Beschreibung

Dieses Makro generiert den Wert 1 bei einer Übersetzung mit `SOURCE-PROPERTIES=PARAMETERS(LANGUAGE-STANDARD=ANSI)`, sonst ist das Makro nicht definiert.

Hinweis Dieses Makro muss in keiner Include-Datei definiert werden. Sein Name wird vom Compiler erkannt und ersetzt.

_ _STDC_VERSION_ _ - Amendment 1 konform?

Syntax _ _STDC _VERSION_ _

Beschreibung

Gibt an, welche Version des ANSI-Standards unterstützt wird
Dieses Makro wird zu der Dezimalkonstanten 199409L expandiert und zeigt damit an, dass die Implementierung Amendment 1-konform ist.

Hinweis Das Makro muss in keiner Include-Datei definiert werden. Sein Name wird vom Compiler erkannt und ersetzt.

stderr, stdin, stdout - Variablen für Standard-Ein-/Ausgabe-Ströme

Syntax

```
#include <stdio.h>

extern FILE *stderr, *stdin, *stdout;
```

Beschreibung

Eine Datei mit zugeordneter Pufferung heißt **Datenstrom** und ist als Zeiger auf einen definierten Datentyp `FILE` deklariert. `fopen()` erzeugt bestimmte beschreibende Daten für einen Datenstrom und liefert einen Zeiger zurück, mit dem dieser Datenstrom in allen weiteren Transaktionen gekennzeichnet wird.

Zum Zeitpunkt des Programmstarts gibt es drei vordefinierte Datenströme, die nicht explizit geöffnet werden müssen (siehe `stdio.h`):

`stdin`	Standard-Eingabe für das Lesen konventioneller Eingaben
`stdout`	Standard-Ausgabe für das Schreiben konventioneller Ausgaben
`stderr`	Standard-Fehlerausgabe für die Ausgabe von Diagnose- und Fehlermeldungen

Ein offener Standard-Fehlerausgabestrom wird nicht vollständig gepuffert (siehe `setvbuf()`); Standard-Eingabe- und Standard-Ausgabestrom werden nur dann vollständig gepuffert, wenn der Datenstrom nicht mit einem interaktiven Gerät verbunden ist.

Folgende symbolische Werte in `unistd.h` definieren die Dateideskriptoren, die den Datenströmen `stdin`, `stdout` und `stderr` zugeordnet werden, wenn die Anwendung gestartet wird:

`STDIN_FILENO`
: Dateideskriptor für Standard-Eingabe, `stdin`. Er hat den Wert 0.

`STDOUT_FILENO`
: Dateideskriptor für Standard-Ausgabe, `stdout`. Er hat den Wert 1.

`STDERR_FILENO`
: Dateideskriptor für Standard-Fehlerausgabe, `stderr`. Er hat den Wert 2.

Siehe auch `fclose()`, `feof()`, `ferror()`, `fileno()`, `fopen()`, `fread()`, `fseek()`, `getc()`, `gets()`, `popen()`, `printf()`, `putc()`, `puts()`, `read()`, `scanf()`, `setbuf()`, `setvbuf()`, `tmpfile()`, `ungetc()`, `vprintf()`, `stdio.h`, `unistd.h`.

step - reguläre Ausdrücke vergleichen

Syntax #include <regexp.h>

int step(const char **string*, const char **exbuf*);

Beschreibung

Siehe `regexp()`.

Hinweis Diese Funktion wird zukünftig vom X/Open-Standard nicht mehr unterstützt.

strcasecmp, strncasecmp - Zeichenkettenvergleich ohne Berücksichtigung der Groß-/Kleinschreibung

Syntax #include <strings.h>

int strcasecmp(const char **s1*, const char **s2*);

int strncasecmp(const char **s1*, const char **s2*, size_t *n*);

Beschreibung

Die Funktion `strcasecmp()` vergleicht die Zeichenkette auf die *s1* verweist, mit der, auf die *s2* verweist. Die zu vergleichenden Zeichenketten müssen mit dem Nullbyte abgeschlossen sein. Die Unterschiede in der Groß-/Kleinschreibung werden dabei nicht berücksichtigt. `strncasecmp()` verfährt analog, nur werden dabei höchstens *n* Bytes zum Vergleich herangezogen.

In der POSIX-Lokalität konvertieren `strcasecmp()` und `strncasecmp()` zuerst Groß- in Kleinbuchstaben und nehmen dann den Zeichenvergleich vor. Die Ergebnisse sind in anderen Lokalitäten nicht spezifiziert.

Returnwert Ganzzahl Nach erfolgreicher Ausführung liefert `strcasecmp()` eine ganze Zahl zurück, die größer, gleich oder kleiner null ist, abhängig davon, ob die durch *s1* bezeichnete Zeichenkette größer, gleich oder kleiner ist als die Zeichenkette, auf die *s2* verweist. Groß-/Kleinschreibung wird dabei nicht berücksichtigt.
`strncasecmp()` verhält sich analog, nur dass hier höchstens die ersten *n* Zeichen beider Zeichenketten berücksichtigt werden.

Siehe auch `strings.h`.

strcat - zwei Zeichenketten zusammenfügen

Syntax #include <string.h>

char *strcat(char **s1*, const char **s2*);

Beschreibung

`strcat()` hängt eine Kopie der Zeichenkette *s2* ans Ende der Zeichenkette *s1* und liefert einen Zeiger auf *s1* zurück.

Das Nullbyte (`\0`) am Ende der Zeichenkette *s1* wird vom ersten Zeichen der Zeichenkette *s2* überschrieben.
`strcat()` schließt die Zeichenkette mit dem Nullbyte (`\0`) ab.

Returnwert Zeiger auf die Ergebniszeichenkette *s1*.

Hinweis Als Argumente werden Zeichenketten erwartet, die mit dem Nullbyte (`\0`) abgeschlossen sind.

`strcat()` überprüft nicht, ob der Speicherbereich von *s1* groß genug für das Ergebnis ist!

Bei sich überlappenden Speicherbereichen ist das Verhalten undefiniert.

Siehe auch `strncat()`, `string.h`.

strchr - Zeichenkette nach Zeichen durchsuchen

Syntax #include <string.h>

char *strchr(const char **s*, int *c*);

Beschreibung

`strchr()` sucht das erste Vorkommen des Zeichens *c* in der Zeichenkette *s* und liefert bei Erfolg einen Zeiger auf die gesuchte Position in *s*.

Das abschließende Nullbyte (`\0`) wird als Zeichen mitberücksichtigt.

Returnwert Zeiger auf die Position von *c* in der Zeichenkette *s*, bei Erfolg.

Nullzeiger, wenn *c* in der Zeichenkette *s* nicht enthalten ist.

Hinweis `strchr()` und `index()` sind äquivalent.

Siehe auch `index()`, `rindex()`, `strrchr()`, `string.h`.

strcmp - zwei Zeichenketten vergleichen

Syntax #include <string.h>

int strcmp(const char **s1*, const char **s2*);

Beschreibung

`strcmp()` vergleicht zwei Zeichenketten *s1* und *s2* lexikalisch, z.B.:

"Zirkel" ist lexikalisch kleiner als "Zirkus".
"Busse" ist lexikalisch größer als "Bus".

Returnwert Ganzzahliger Wert, und zwar:

< 0	*s1* ist lexikalisch kleiner als *s2*.
= 0	*s1* und *s2* sind lexikalisch gleich groß.
> 0	*s1* ist lexikalisch größer als *s2*.

Hinweis Als Argumente werden Zeichenketten erwartet, die mit dem Nullbyte (`\0`) abgeschlossen sind. Ist das nicht der Fall, ist das Ergebnis zufällig.

Es gilt die Sortierreihenfolge des EBCDIC-Zeichensatzes.

Siehe auch `strncmp()`, `string.h`.

strcoll - Zeichenketten nach Sortierreihenfolge vergleichen

Syntax

```
#include <string.h>

int strcoll(const char *s1, const char *s2);
```

Beschreibung

`strcoll()` liefert eine ganze Zahl, die größer, kleiner oder gleich null ist, abhängig davon, ob die Zeichenkette *s1* größer, gleich oder kleiner als die Zeichenkette *s2* ist. Der Vergleich der Zeichenketten hängt davon ab, wie die Kategorie `LC_COLLATE` der jeweiligen lokalen Umgebung eingestellt ist (siehe `setlocale()`).

Mit `strcoll()` und `strxfrm()` können Zeichenketten umgebungsabhängig sortiert werden. `strcoll()` ist für Anwendungen gedacht, bei denen die Anzahl der Vergleiche pro Zeichenkette gering ist. Wenn Zeichenketten oft verglichen werden, sollte `strxfrm()` zusammen mit `strcmp()` so verwendet werden, dass der Transformationsprozess nur einmal stattfindet.

Returnwert Ganzzahliger Wert, und zwar:

< 0	*s1* ist lexikalisch kleiner als *s2*.
= 0	*s1* und *s2* sind lexikalisch gleich groß.
> 0	*s1* ist lexikalisch größer als *s2*.

Fehler `strcoll()` schlägt fehl, wenn gilt:

`EINVAL`	Die Argumente *s1* oder *s2* enthalten Zeichen, die außerhalb des Definitionsbereichs der Sortierreihenfolge liegen.

Hinweis Als Argumente werden Zeichenketten erwartet, die mit dem Nullbyte (`\0`) abgeschlossen sind.

Da `strcoll()` keinen Returnwert für die Anzeige eines Fehlers zurückgibt, können Fehler nur wie folgt festgestellt werden: `errno` wird auf 0 gesetzt; anschließend wird die Funktion aufgerufen und `errno` geprüft. Wenn `errno` ungleich 0 ist, muss ein Fehler aufgetreten sein.

Siehe auch `setlocale()`, `strcmp()`, `strxfrm()`, `string.h`.

strcpy - Zeichenkette kopieren

Syntax #include <string.h>

char *strcpy(char **s1*, const char **s2*);

Beschreibung

`strcpy()` kopiert die Zeichenkette *s2* einschließlich des Nullbytes (`\0`) in den Speicherbereich, auf den *s1* zeigt. *s1* muss groß genug sein, um die Zeichenkette *s2* einschließlich des Nullbytes (`\0`) aufnehmen zu können.

Returnwert Zeiger auf die Ergebniszeichenkette *s1*.

Hinweis Als zweites Argument wird eine Zeichenkette erwartet, die mit dem Nullbyte (`\0`) abgeschlossen ist.
`strcpy()` überprüft nicht, ob *s1* groß genug für das Ergebnis ist.
Bei sich überlappenden Speicherbereichen ist das Verhalten undefiniert.

Siehe auch `strncpy()`, `string.h`.

strcspn - Länge einer komplementären Teilzeichenkette ermitteln

Syntax #include <string.h>

size_t strcspn(const char **s1*, const char **s2*);

Beschreibung

`strcspn()` berechnet ab Beginn der Zeichenkette *s1* die Länge des Segmentes, das kein einziges Zeichen aus der Zeichenkette *s2* enthält. Das abschließende Nullbyte (`\0`) gilt nicht als Teil der Zeichenkette *s2*.
Sobald ein Zeichen in *s1* mit einem Zeichen in *s2* übereinstimmt, wird die Funktion beendet und die Segmentlänge zurückgeliefert.
Wenn bereits das erste Zeichen in *s1* mit einem Zeichen in *s2* übereinstimmt, ist die Segmentlänge gleich 0.

Returnwert Ganzzahliger Wert, der die Segmentlänge (Anzahl ungleicher Zeichen) ab Beginn der Zeichenkette *s1* angibt.

Hinweis Als Argumente werden Zeichenketten erwartet, die mit dem Nullbyte (`\0`) abgeschlossen sind.

Siehe auch `strspn()`, `string.h`.

strdup - Zeichenkette kopieren

Syntax #include <string.h>

char *strdup(const char **s1*);

Beschreibung

`strdup()` liefert einen Zeiger auf eine neue Zeichenkette zurück, welche eine Kopie der Zeichenkette ist, auf die *s1* zeigt. Der Speicherplatz für die neue Zeichenkette wird mit `malloc()` zugewiesen. Der zurückgelieferte Zeiger kann der Funktion `free()` übergeben werden. Wenn die neue Zeichenkette nicht angelegt werden kann, wird ein Nullzeiger zurückgegeben.

Returnwert Die Funktion liefert bei Erfolg einen Zeiger auf eine neue Zeichenkette zurück. Andernfalls wird ein Nullzeiger zurückgegeben und `errno` gesetzt, um den Fehler anzuzeigen.

Fehler `strdup()` schlägt fehl, wenn gilt:

`ENOMEM` Speicherplatz reicht nicht aus.

Siehe auch `malloc()`, `free()`, `string.h`.

strerror - Meldungstext ermitteln

Syntax #include <string.h>

char *strerror(int *errnum*);

Beschreibung

`strerror()` bildet die Fehlernummer in *errnum* auf einen lokalitätsabhängigen Meldungstext ab und liefert einen Zeiger auf diese Zeichenkette (siehe auch Abschnitt „Fehlerbehandlung“ auf Seite 130). Die zurückgegebene Zeichenkette darf vom Programm nicht verändert werden, kann aber durch einen nachfolgenden Aufruf der Funktionen `strerror()` oder `popen()` überschrieben werden.

Der Inhalt der von `strerror()` zurückgegebenen Meldungszeichenkette sollte durch Setzen der `LC_MESSAGES`-Kategorie der aktuellen Lokalität festgelegt werden. Fehlernummern und Fehlermeldungen sind in `errno.h` vollständig aufgeführt und erläutert.

Returnwert Zeiger auf eine Meldungszeichenkette bei Erfolg.

Nullzeiger bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `strerror()` schlägt fehl , wenn gilt:

`EINVAL` Der Wert von *errnum* ist keine gültige Fehlernummer.

Hinweis Da kein Returnwert für die Anzeige eines Fehlers reserviert ist, sollte eine Anweisung für die Fehlerbehandlung `errno` gleich 0 setzen, dann `strerror()` aufrufen, dann `errno` prüfen: Wenn `errno` ungleich 0 ist, muss ein Fehler aufgetreten sein.

Siehe auch `perror()`, `popen()`, `errno.h`, `string.h`, Abschnitt „Fehlerbehandlung“ auf Seite 130.

strfill - Teilzeichenkette kopieren *(BS2000)*

Syntax #include <string.h>

char *strfill(char **s1*, const char **s2*, size_t *n*);

Beschreibung

`strfill()` kopiert maximal *n* Zeichen aus der Zeichenkette *s2* in den Speicherbereich, auf den *s1* zeigt.

Je nach Länge bzw. Inhalt der Zeichenketten *s1*, *s2* und der Anzahl *n* wird folgendermaßen kopiert:

1. Unabhängig von der Länge der Zeichenkette *s1* werden (mit Ausnahme von Fall 5.) immer *n* Zeichen nach *s1* kopiert; d.h.:
   - Wenn *s1* mehr als *n* Zeichen enthält, bleiben die restlichen rechten Zeichen in *s1* erhalten.
   - Wenn *s1* weniger als *n* Zeichen enthält, wird *s1* bis zur Länge *n* verlängert. In diesem Fall ist *s1* nach dem Kopiervorgang nicht automatisch mit einem Nullbyte abgeschlossen (siehe auch Hinweis).
2. *s2* enthält weniger als *n* Zeichen:

   Zusätzlich zu den kopierten Zeichen aus *s2* werden noch so viele Leerzeichen hinzugefügt, bis *n* Zeichen geschrieben wurden.
3. *s2* enthält mehr als *n* Zeichen:

   Es werden nur die führenden *n* Zeichen aus *s2* kopiert.
4. *s2* ist leer:

   s1 wird mit *n* Leerzeichen aufgefüllt.
5. *s2* wird als Nullzeiger übergeben:

   Es werden (*n* - `strlen`(*s1*)) Leerzeichen an die Zeichenkette *s1* angehängt. Wenn diese Subtraktion ein negatives Ergebnis oder 0 ergibt, d.h. die Anzahl der Zeichen in *s1* größer oder gleich *n* ist, bleibt der Inhalt von *s1* unverändert.

Returnwert Zeiger auf die Ergebniszeichenkette *s1*.

Hinweis Als Argumente werden Zeichenketten erwartet, die mit dem Nullbyte (`\0`) abgeschlossen sind.

`strfill()` überprüft nicht, ob *s1* groß genug für das Ergebnis ist und schließt die Ergebniszeichenkette nicht automatisch mit dem Nullbyte (`\0`) ab! Um kein unvorhergesehenes Ergebnis zu erhalten, sollten Sie nach jedem Aufruf von `strfill()` die Zeichenkette *s1* explizit mit dem Nullbyte abschließen (siehe auch Beispiel).

Bei sich überlappenden Speicherbereichen ist das Verhalten undefiniert.

Siehe auch `strncpy()`.

strfmon - Monetären Wert in Zeichenkette umwandeln

Syntax #include <monetary.h>

ssize_t strfmon(char *s, size_t *maxsize*, const char **format*, ...);

Beschreibung

`strfmon()` schreibt gemäß der Angabe *format* Zeichen vom Typ Character in das Feld, auf das *s* zeigt. Es werden nicht mehr als *maxsize* Byte in das Feld geschrieben.

format ist eine Zeichenkette, die zwei Arten von Objekten enthält: einfache Zeichen, die in den Ausgabestrom kopiert werden, und Konvertierungs-Spezifikationen. Konvertierungs-Spezifikationen bewirken, dass Argumente (null, eins oder mehrere) konvertiert und formatiert werden. Falls nicht genügend Argumente für das angegebene Format vorhanden sind, ist das Ergebnis undefiniert. Sind mehr Argumente vorhanden, als im Format vorgesehen, werden die überzähligen Argumente ignoriert.

Eine Konvertierungs-Spezifikation besteht aus folgenden Elementen:

1. einem %-Zeichen
2. optionalen Flags
3. einer optionalen Feldgröße
4. einer optionalen linken Genauigkeit
5. einer optionalen rechten Genauigkeit
6. einem Konvertierungs-Zeichen, das bestimmt, wie konvertiert wird (Pflichtangabe)

Flags

Um die Konvertierung zu steuern, können Sie ein Flag oder mehrere der hier aufgeführten Flags angeben:

=*f* Ein Gleichheitszeichen, dem ein einzelnes Zeichen *f* folgt. Dieses Zeichen wird als Füllzeichen bei numerischen Werten verwendet. Das Füllzeichen muss sich in einem einzigen Byte darstellen lassen, damit es nicht mit Angaben zur Feldgröße und zur Ausrichtung kollidiert. Voreingestelltes Füllzeichen ist das Leerzeichen.
Dieses Flag hat keinen Einfluss auf das Auffüllen wegen einer Feldgrößen-Angabe: hierfür wird immer das Leerzeichen als Füllzeichen verwendet.
Das Flag wird ignoriert, wenn keine linke Genauigkeit angegeben ist.

^ Monetäre Werte werden ohne Gruppierungszeichen formatiert. Voreingestellt ist, dass monetäre Werte mit den für die aktuelle Lokalität gültigen Gruppierungszeichen formatiert werden.

+ oder (
: Steuert, wie positive und negative monetäre Werte dargestellt werden. Nur eines der beiden Zeichen + oder (darf angegeben werden.
Ist + angegeben, werden die in der aktuellen Lokalität definierten Werte für + und - verwendet (in den USA z.B. die leere Zeichenkette für positive Werte und das Zeichen - für negative Werte).
Ist (angegeben, werden negative Werte in Klammern eingeschlossen.
Voreingestellt ist die Angabe +.

!
: Unterdrückt das Währungszeichen in der Ausgabe.

-
: Steuert die Ausrichtung. Wenn dieses Flag gesetzt ist, werden Werte in den Feldern linksbündig statt rechtsbündig ausgerichtet (d.h. rechts aufgefüllt).

Feldgröße

w
: Eine Folge von Dezimalziffern, die die minimale Feldgröße in Bytes festlegt. Das Ergebnis der Konvertierung wird rechtsbündig in dem Feld abgelegt und evtl. aufgefüllt (das Ergebnis wird linksbündig abgelegt, wenn das Flag - gesetzt ist).
Voreingestellt ist die Feldgröße 0.

Linke Genauigkeit

#n
: Eine Folge von Dezimalziffern, der das Zeichen # vorangestellt ist. Dieser Wert gibt an, wie viele Ziffern maximal links vom Radix-Zeichen (z.B. das Komma in DM **15,20) erwartet werden.
Diese Option kann dazu verwendet werden, das Ergebnis von mehreren `strfmon`-Aufrufen in Spalten auszurichten. Außerdem kann sie verwendet werden, um freie Positionen mit einem speziellen Zeichen aufzufüllen, wie z.B. $***123.45.
Diese Option bewirkt, dass ein monetärer Wert so formatiert wird, als hätte er *n* Ziffern. Werden mehr als *n* Ziffernpositionen benötigt, wird diese Konvertierungs-Spezifikation ignoriert. Freie Ziffernpositionen werden mit dem numerischen Füllzeichen aufgefüllt (siehe Flag =*f*).

  Falls in der aktuellen Lokalität eine Gruppierung definiert ist und die Gruppierung nicht unterdrückt wird (Flag ^), werden die Gruppierungszeichen eingefügt, bevor freie Positionen mit Füllzeichen aufgefüllt werden. Füllzeichen werden nicht gruppiert, auch dann nicht, wenn das Füllzeichen numerisch ist.

  Um die Ausrichtung sicherzustellen, werden alle Zeichen wie Währungszeichen oder Minuszeichen vor oder nach der Zahl in der formatierten Ausgabe durch Leerzeichen so positioniert, dass ihre positiven und negativen Formate die gleichen Längen haben.

Rechte Genauigkeit

.p Eine Folge von Dezimalziffern, der das Zeichen . vorangestellt ist. Dieser Wert gibt an, wie viele Ziffern rechts vom Radix-Zeichen (z.B. dem Komma in DM **15,20) erscheinen sollen. Hat *p* den Wert 0, entfällt auch das Radix-Zeichen. Wenn keine rechte Genauigkeit angegeben ist, wird die in der aktuellen Lokalität definierte rechte Genauigkeit verwendet.
Der zu formatierende Betrag wird vor der Formatierung auf die angegebene Anzahl von Ziffern gerundet.

Konvertierungs-Zeichen

Es gibt folgende Konvertierungs-Zeichen:

i Das Argument vom Typ `double` wird gemäß dem internationalen Währungsformat formatiert, das in der Lokalität definiert ist (z.B. in den USA: `USD 1,234.56`).

n Das Argument vom Typ `double` wird gemäß dem nationalen Währungsformat formatiert, das in der Lokalität definiert ist (z.B. in den USA: `$1,234.56`).

% Konvertiert zu einem %., es wird kein Argument konvertiert. Die gesamte Konvertierungs-Spezifikation muss %% sein.

Lokalitäts-Informationen

Das Verhalten der Funktion wird durch die Kategorie `LC_MONETARY` der Lokalität des Programms beeinflusst. Das gilt insbesondere für das monetäre Radix-Zeichen (das ein anderes Zeichen sein kann als das numerische Radix-Zeichen, das für die `LC_NUMERIC`-Kategorie gilt), die Gruppierungszeichen, die Währungssymbole und Währungsformate. Das internationale Währungssymbol sollte dem ISO 4217:1987-Standard entsprechen.

Returnwert Anzahl der Bytes, die in das Feld geschrieben wurden, auf das *s* zeigt (ohne das abschließende Nullbyte), wenn die Gesamtzahl der geschriebenen Bytes inklusive des Nullbyte nicht größer als *maxsize* ist.

-1 sonst. Im Fehlerfall ist der Inhalt des Feldes undefiniert. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `strfmon()` schlägt fehl, wenn gilt:

`E2BIG` Wegen Platzmangels im Puffer wurde die Konversion abgebrochen.

Beispiel Die folgenden Beispiele beziehen sich auf eine Lokalität in den USA und die Werte 123.45, -123.45 und 3456.781:

Konvert.-Spezifikation	**Ergebnis**	**Kommentar**
%n	$123.45 -$123.45 $3,456.78	Default-Formatierung
%11n	$123.45 -$123.45 $3,456.78	Ausrichtung rechts innerhalb eines 11 Zeichen großen Feldes
%#5n	$ 123.45 -$ 123.45 $ 3,456.78	Werte bis 99,999 werden in einer Spalte ausgerichtet
%=*#5n	$***123.45 -$***123.45 $*3,456.78	Angabe eines Füllzeichens für freie Positionen
%=0#5n	$000123.45 -$000123.45 $03,456.78	Füllzeichen werden nicht gruppiert, selbst wenn das Füllzeichen eine Ziffer ist
%^#5n	$ 123.45 -$ 123.45 $ 3456.78	Gruppierungszeichen unterdrücken
%^#5.0n	$ 123 -$ 123 $ 3456	Auf ganze Zahl runden
%^#5.4n	$ 123.4500 -$ 123.4500 $ 3456.7800	Rechte Genauigkeit erhöhen
%(#5n	$ 123.45 ($ 123.45) $ 3456.78	Alternative Darstellung für positive/negative Werte
%!(#5n	123.45 (123.45) 3456.78	Währungssymbol unterdrücken

Siehe auch `localeconv()`, `monetary.h`.

strftime - Datum und Uhrzeit in Zeichenkette umwandeln

Syntax #include <time.h>

size_t strftime(char *s, size_t *maxsize*, const char **format*, const struct tm **timeptr*);

Beschreibung

`strftime()` formatiert Datum und Zeit nach den Angaben aus der Zeichenkette *format* in das Feld, auf das `s` zeigt. Dabei besteht die Zeichenkette *format* aus einer oder mehreren Umwandlungsanweisungen und gewöhnlichen Zeichen. Alle gewöhnlichen Zeichen, einschließlich des abschließenden Nullbytes, werden unverändert in die Zeichenkette kopiert. Bei Verwendung von `strftime()` werden nicht mehr als *maxsize* Zeichen in die Zeichenkette geschrieben.
Wenn *format* gleich `(char *)0` ist, dann wird für `strftime()` das voreingestellte Format `"%c"` verwendet.

Jede Umwandlungsanweisung wird durch die entsprechenden Zeichen ersetzt, die in der folgenden Liste beschrieben werden. Die entsprechenden Zeichen werden durch die Kategorie `LC_TIME` der lokalen Umgebung, bei `strftime()` durch den Inhalt von *timeptr* bestimmt:

```
%%    ist das Zeichen %
%a    abgekürzter Wochentagsname der Lokalität
%A    ausgeschriebener Wochentagsname der Lokalität
%b    abgekürzter Monatsname der Lokalität
%B    ausgeschriebener Monatsname der Lokalität
%c    entsprechende Datums- und Zeitdarstellung der Lokalität
%C    Jahrhundert (Jahr geteilt durch 100 als ganze Zahl) (00-99)
%d    Monatstag (01-31)
%D    Datum als %m/%d/%y
%e    Monatstag (1-31; vor einzelnen Ziffern steht ein Leerzeichen)
%f    Datums- und Zeitdarstellung nach date()
%h    abgekürzter Monatsname der Lokalität
%H    Stunde (00-23), 24-Stunden Darstellung
%I    Stunde (01-12), 12-Stunden Darstellung
%j    Tag des Jahres (001-366)
%m    Nummer des Monats (01-12)
%M    Minute (00-59)
```

%n entspricht \n

%p Bezeichnung der Lokalität entweder für AM oder PM

%r Zeit im Format %I:%M:%S [AM•PM]

%R Zeit im Format %H:%M

%S Sekunden (00–61), erlaubt Schaltsekunden

%t fügt ein Tabulatorzeichen ein

%T Zeit im Format %H:%M:%S

%u Wochentag als Zahl (1–7), Montag = 1

%U Nummer der Woche im Jahr (00–53). Die erste Woche beginnt mit dem ersten Sonntag des Jahres. Alle Tage vor dem ersten Sonntag des Jahres gehören zur Woche 0.

%V Nummer der Woche im Jahr (01–53), Montag ist der erste Tag der Woche. Wenn die Woche mit dem 1. Januar im neuen Jahr vier oder mehr Tage hat, hat diese Woche die Nummer 1, sonst hat sie die Nummer 53 des vorhergehenden Jahres und die Darauf folgende Woche hat die Nummer 1.

%w Wochentag als Zahl (0–6), Sonntag = 0

%W Nummer der Woche im Jahr (00–53), Montag ist der erste Tag der Woche 1. Alle Tage vor dem ersten Montag des Jahres gehören zur Woche 0.

%x die entsprechende Datumsdarstellung der Lokalität

%X die entsprechende Zeitdarstellung der Lokalität

%y Jahr innerhalb des Jahrhunderts (00–99)

%Y Jahr in der Form *ccyy* (z.B. 1986)

%Z Zeitzonenname oder Abkürzung dieses Namens; enthält keine Zeichen, wenn keine Zeitzone existiert

Der Unterschied zwischen %U und %W besteht darin, welcher Tag der erste in einer Woche ist. Die Woche 01 ist die erste Woche im Januar, die mit einem Sonntag (bei %U) oder einem Montag (bei %W) anfängt. Die Wochennummer 00 enthält diejenigen Tage vor dem ersten Sonntag (%U) oder Montag (%W) im Januar.

Modifizierte Umwandlungsanweisungen

Einige Umwandlungsanweisungen können mit den Zeichen E und O modifiziert werden. Dadurch wird angegeben, dass ein alternatives Format bzw. eine andere Anweisung verwendet werden soll, als bei der unmodifizierten Anweisung (siehe Handbuch „SINIX 5.41 Leitfaden für Programmierer: Internationalisierung - Lokalisierung“). Wenn dieses Format oder diese Anweisung nicht in der aktuellen lokalen Umgebung vorliegt, ist das Verhalten wie bei der unmodifizierten Umwandlungsanweisung.

`%Ec` alternative Darstellung der Lokalität für Datum und Zeit.

`%EC` Name des Basisjahrs (Zeitraum) bei der alternativen Darstellung der Lokalität.

`%Ex` alternative Darstellung der Lokalität für das Datum.

`%EX` alternative Darstellung der Lokalität für die Zeit.

`%Ey` Offset zu `%EC` (nur Jahr) in der alternativen Darstellung der Lokalität.

`%EY` alternative Darstellung für das Jahr.

`%Od` Monatstag; Verwendung der alternativen numerischen Symbole der Lokalität; es wird mit führenden Nullen nach Bedarf aufgefüllt, wenn es ein alternatives Symbol für Null gibt, andernfalls wird mit führenden Leerzeichen aufgefüllt.

`%Oe` Monatstag; Verwendung der alternativen numerischen Symbole der Lokalität; bei Bedarf wird mit führenden Leerzeichen aufgefüllt.

`%OH` Stunde (Angabe in 24-Stunden-Darstellung); Verwendung der alternativen numerischen Symbole der Lokalität.

`%OI` Stunde (Angabe in 12-Stunden-Darstellung); Verwendung der alternativen numerischen Symbole der Lokalität.

`%Om` Monat; Verwendung der alternativen numerischen Symbole der Lokalität.

`%OM` Minuten; Verwendung der alternativen numerischen Symbole der Lokalität.

`%OS` Sekunden; Verwendung der alternativen numerischen Symbole der Lokalität.

`%Ou` Nummer des Wochentags in der alternativen Darstellung der Lokalität (Montag = 1).

`%OU` Nummer der Woche (Sonntag ist der erste Tag der Woche; Regeln entsprechend `%U`); Verwendung der alternativen numerischen Symbole der Lokalität.

`%OV` Nummer der Woche (Sonntag ist der erste Tag der Woche; Regeln entsprechend `%V`); Verwendung der alternativen numerischen Symbole der Lokalität.

`%Ow` Nummer des Wochentags (Sonntag = 0); Verwendung der alternativen numerischen Symbole der Lokalität.

`%OW` Nummer der Woche (Montag ist der erste Tag der Woche); Verwendung der alternativen numerischen Symbole der Lokalität.

`%Oy` Jahr (Offset zu `%C`) in der alternativen Darstellung der Lokalität und mit den alternativen Symbolen der Lokalität.

Die voreingestellte Sprache für die Ausgaben von `strftime()` ist amerikanisches Englisch. Der Benutzer kann die Ausgabesprache von `strftime()` durch Einstellung der Kategorie `LC_TIME` mit `setlocale()` für die Lokalität auswählen.

Die Zeitzone wird aus der Umgebungsvariablen `TZ` übernommen (siehe `ctime()`).

Returnwert Anzahl der Bytes, die nach *s* kopiert wurden (ohne das abschließende Nullbyte)
wenn die Anzahl der resultierenden Bytes einschließlich Nullbyte nicht größer als *maxsize* ist.

0 bei Fehler. Der Inhalt von *s* ist unbestimmt.

Siehe auch `clock()`, `ctime()`, `getenv()`, `setlocale()`, `time.h`.

strlen - Länge einer Zeichenkette ermitteln

Syntax #include <string.h>

size_t strlen(const char *s);

Beschreibung

`strlen()` bestimmt die Länge der Zeichenkette *s*, ohne das abschließende Nullbyte (`\0`).

BS2000
Während der `sizeof`-Operator immer die definierte Länge liefert, berechnet `strlen()` die aktuelle Anzahl von Zeichen in einer Zeichenkette. Ein Zeilenendezeichen (`\n`) wird mitgezählt. ❑

Returnwert Länge der Zeichenkette *s*
bei Erfolg. Das abschließende Nullbyte wird nicht mitgezählt.

Hinweis Als Argument wird eine Zeichenkette erwartet, die mit dem Nullbyte (`\0`) abgeschlossen ist.

strlower - Zeichenkette in Kleinbuchstaben umwandeln *(BS2000)*

Syntax #include <string.h>

char *strlower(char *s1, const char *s2);

Beschreibung

`strlower()` kopiert die Zeichenkette *s2* einschließlich des Nullbytes (`\0`) in den Speicherbereich, auf den *s1* zeigt und wandelt die Großbuchstaben in Kleinbuchstaben um.
Wenn die Zeichenkette *s2* als Nullzeiger übergeben wird, entfällt der Kopiervorgang und in *s1* werden die Großbuchstaben in Kleinbuchstaben umgewandelt.
s1 ist die Ergebniszeichenkette, in die kopiert werden soll bzw. in der die Groß- in Kleinbuchstaben umgewandelt werden sollen.
Wenn *s2* nicht als Nullzeiger übergeben wird, muss *s1* groß genug sein, um *s2* einschließlich des Nullbytes (`\0`) aufnehmen zu können.

Returnwert Zeiger auf die Ergebniszeichenkette *s1*.

Hinweis Als Argumente werden Zeichenketten erwartet, die mit dem Nullbyte (`\0`) abgeschlossen sind.
`strlower()` überprüft nicht, ob *s1* groß genug für das Ergebnis ist.
Bei sich überlappenden Speicherbereichen ist das Verhalten undefiniert.

Siehe auch `strupper()`, `tolower()`, `toupper()`.

strncasecmp - Zeichenkettenvergleich ohne Berücksichtigung der Groß-/Kleinschreibung

Syntax #include <strings.h>

int strncasecmp(const char **s1*, const char **s2*, size_t *n*);

Beschreibung

siehe `strcasecmp()`.

strncat - zwei Teilzeichenketten zusammenfügen

Syntax #include <string.h>

char *strncat(char **s1*, const char **s2*, size_t *n*);

Beschreibung

`strncat()` hängt maximal *n* Zeichen der Zeichenkette *s2* an das Ende der Zeichenkette *s1* an und liefert einen Zeiger auf *s1* zurück.

Das Nullbyte (`\0`) am Ende der Zeichenkette *s1* wird vom ersten Zeichen der Zeichenkette *s2* überschrieben.

Wenn die Zeichenkette *s2* weniger als *n* Zeichen enthält, werden nur die Zeichen aus *s2* an *s1* angehängt. Wenn die Zeichenkette *s2* mehr als *n* Zeichen enthält, werden nur die führenden *n* Zeichen von *s2* an *s1* angehängt.

`strncat()` schließt die Zeichenkette mit dem Nullbyte (`\0`) ab.

Returnwert Zeiger auf die Ergebniszeichenkette *s1*.

Hinweis Als Argumente werden Zeichenketten erwartet, die mit dem Nullbyte abgeschlossen sind.

`strncat()` überprüft nicht, ob der Speicherbereich *s1* groß genug für das Ergebnis ist. Bei sich überlappenden Speicherbereichen ist das Verhalten undefiniert.

Siehe auch `strcat()`, `string.h`.

strncmp - zwei Teilzeichenketten vergleichen

Syntax #include <string.h>

int strncmp(const char **s1*, const char **s2*, size_t *n*);

Beschreibung

`strncmp()` vergleicht die Zeichenketten *s1* und *s2* bis zur maximalen Länge *n* lexikalisch; z.B liefert

```
strncmp("Sie","Siemens",3)
```

das Ergebnis 0 (gleich), weil die beiden Argumente in den ersten drei Zeichen übereinstimmen.

Returnwert Ganzzahliger Wert, und zwar:

< 0	*s1* ist in den ersten *n* Zeichen lexikalisch kleiner als *s2*.
0	*s1* und *s2* sind in den ersten *n* Zeichen lexikalisch gleich groß.
> 0	*s1* ist in den ersten *n* Zeichen lexikalisch größer als *s2*.

Hinweis Als Argumente werden Zeichenketten erwartet, die mit dem Nullbyte (`\0`) abgeschlossen sind.

Es gilt die Sortierreihenfolge des EBCDIC-Zeichensatzes.

Siehe auch `strcmp()`, `string.h`.

strncpy - Teilzeichenkette kopieren

Syntax #include <string.h>

char *strncpy(char **s1*, const char **s2*, size_t *n*);

Beschreibung

`strncpy()` kopiert maximal *n* Zeichen der Zeichenkette *s2* in den Speicherbereich, auf den *s1* zeigt.

Wenn die Zeichenkette *s2* weniger als *n* Zeichen enthält, wird nur in der Länge von *s2* (`strlen` + 1) kopiert, *s1* wird dann bis zur Länge *n* mit Nullbytes aufgefüllt.

Wenn die Zeichenkette *s2* *n* Zeichen (ohne das Nullbyte) oder mehr enthält, wird die Zeichenkette *s1* nicht automatisch mit dem Nullbyte abgeschlossen.

Wenn die Zeichenkette *s1* mehr als *n* Zeichen enthält und das letzte kopierte Zeichen aus *s2* nicht das Nullbyte ist, bleiben ggf. restliche Daten in *s1* erhalten.

`strncpy()` schließt *s1* nicht automatisch mit dem Nullbyte ab.

Returnwert Zeiger auf die Ergebniszeichenkette *s1*.

Hinweis `strncpy()` überprüft nicht, ob der Speicherbereich *s1* groß genug für das Ergebnis ist!

Da `strncpy()` die Ergebniszeichenkette nicht automatisch mit dem Nullbyte abschließt, kann es häufig notwendig sein, *s1* explizit mit einem Nullbyte abzuschließen. Das ist z.B. der Fall, wenn nur ein Teilstück aus *s2* kopiert wird und auch *s2* kein Nullbyte enthält.

Bei sich überlappenden Speicherbereichen ist das Verhalten undefiniert.

Siehe auch `strcpy()`, `strlen()`, `string.h`.

strpbrk - erstes Vorkommen eines Zeichens in Zeichenkette ermitteln

Syntax #include <string.h>

char *strpbrk(const char **s1*, const char **s2*);

Beschreibung

`strpbrk()` sucht das erste Zeichen in der Zeichenkette *s1*, das mit irgendeinem Zeichen aus der Zeichenkette *s2* übereinstimmt und liefert bei Erfolg einen Zeiger auf die gesuchte Position in *s1*. Das abschließende Nullbyte (`\0`) gilt nicht als Teil der Zeichenkette *s2*.

Returnwert Zeiger auf das erste gefundene Zeichen in *s1*
bei Erfolg.

Nullzeiger falls keinerlei Übereinstimmung vorliegt.

Hinweis Als Argumente werden Zeichenketten erwartet, die mit dem Nullbyte (`\0`) abgeschlossen sind.

Siehe auch `strchr()`, `strrchr()`, `string.h`.

strptime - Zeichenkette in Datum und Uhrzeit umwandeln

Syntax #include <time.h>

char *strptime(const char **buf*, const char **format*, struct tm **tm*);

Beschreibung

`strptime()` konvertiert unter Berücksichtigung von *format* die Zeichenkette, auf die **buf* zeigt, in Einzelwerte, die in der Struktur abgelegt werden, auf die **tm* zeigt.

Die Zeichenkette *format* besteht aus null, einer oder mehreren Umwandlungsanweisungen. Jede Umwandlungsanweisung besteht aus einem der folgenden Elemente:
einem oder mehreren Zwischenraumzeichen (wie in `isspace()` definiert)
einem gewöhnlichen Zeichen (weder `%` noch Zwischenraumzeichen)
oder einer Konvertierungs-Spezifikation.

Jede Konvertierungs-Spezifikation besteht aus einem `%`-Zeichen, gefolgt von einem Konvertierungszeichen, das die gewünschte Umwandlung angibt. Zwischen zwei Konvertierungs-Spezifikationen muss ein Zwischenraum-Zeichen oder ein nicht-alphanumerisches Zeichen stehen.

Folgende Konvertierungs-Zeichen werden unterstützt:

`%%` wird ersetzt durch %

`%a` Wochentag, wobei die Namen aus der Lokalität verwendet werden. Es kann entweder der abgekürzte oder der ausgeschriebene Name angegeben werden.

`%A` gleiche Bedeutung wie %a

`%b` Monat, wobei die Namen aus der Lokalität verwendet werden. Es kann entweder der abgekürzte oder der ausgeschriebene Name angegeben werden.

`%B` leiche Bedeutung wie %b

`%c` Datums- und Zeitdarstellung entsprechend der Definition in der Lokalität

`%C` Jahrhundert (Jahr geteilt durch 100 als ganze Zahl) (00-99)

`%d` Monatstag (01-31)

`%D` Datum als %m/%d/%y

`%e` gleiche Bedeutung wie %d

`%h` gleiche Bedeutung wie %b

`%H` Stunde (00-23), 24-Stunden Darstellung

`%I` Stunde (01-12), 12-Stunden Darstellung

`%j` Tag des Jahres (001–366)

`%m` Nummer des Monats (01–12)

`%M` Minute (00–59)

`%n` wird ersetzt durch ein Zwischenraum-Zeichen

`%p` äquivalente Bezeichnung der Lokalität für AM oder PM

`%r` Zeit im Format %I:%M:%S%p

`%R` Zeit im Format %H:%M

`%S` Sekunden (00–61), erlaubt Schaltsekunden

`%t` wird ersetzt durch ein Zwischenraum-Zeichen

`%T` Zeit im Format %H:%M:%S

`%U` Nummer der Woche im Jahr (00–53). Die erste Woche beginnt mit dem ersten Sonntag des Jahres. Alle Tage vor dem ersten Sonntag des Jahres gehören zur Woche 0.

`%w` Wochentag als Zahl (0–6), Sonntag = 0

`%W` Nummer der Woche im Jahr (00–53), Montag ist der erste Tag der Woche 1. Alle Tage vor dem ersten Montag des Jahres gehören zur Woche 0.

`%x` Datum in der Darstellung der Lokalität

`%X` Zeit in der Darstellung der Lokalität

`%y` Jahr innerhalb des Jahrhunderts (00–99)

`%Y` Jahr in der Form *ccyy* (z.B. 1986)

Modifizierte Umwandlungsanweisungen

Einige Umwandlungsanweisungen können mit den Zeichen E und O modifiziert werden. Dadurch wird angegeben, dass ein alternatives Format bzw. eine andere Anweisung verwendet werden soll, als bei der unmodifizierten Anweisung (siehe Handbuch „SINIX 5.41 Leitfaden für Programmierer: Internationalisierung - Lokalisierung“). Wenn dieses alternative Format oder diese alternative Anweisung in der aktuellen lokalen Umgebung nicht existiert, ist das Verhalten wie bei der unmodifizierten Umwandlungsanweisung.

`%Ec` alternative Darstellung der Lokalität für Datum und Zeit.

`%EC` Name des Basisjahrs (Zeitraum) in der alternativen Darstellung der Lokalität.

`%Ex` alternative Darstellung der Lokalität für das Datum.

`%EX` alternative Darstellung der Lokalität für die Zeit.

`%Ey` Offset zu `%EC` (nur Jahr) in der alternativen Darstellung der Lokalität.

`%EY` alternative Darstellung für das Jahr.

`%Od` Monatstag; Verwendung der alternativen numerischen Symbole der Lokalität; es wird mit führenden Nullen nach Bedarf aufgefüllt, wenn es ein alternatives Symbol für Null gibt, andernfalls wird mit führenden Leerzeichen aufgefüllt.

`%Oe` gleiche Bedeutung wie `%Od`

`%OH` Stunde (Angabe in 24-Stunden-Darstellung); Verwendung der alternativen numerischen Symbole der Lokalität.

`%OI` Stunde (Angabe in12-Stunden-Darstellung); Verwendung der alternativen numerischen Symbole der Lokalität.

`%Om` Monat; Verwendung der alternativen numerischen Symbole der Lokalität.

`%OM` Minuten; Verwendung der alternativen numerischen Symbole der Lokalität.

`%OS` Sekunden; Verwendung der alternativen numerischen Symbole der Lokalität.

`%OU` Nummer der Woche (Sonntag ist der erste Tag der Woche; Regeln entsprechend `%U`); Verwendung der alternativen numerischen Symbole der Lokalität.

`%OV` Nummer der Woche (Sonntag ist der erste Tag der Woche; Regeln entsprechend `%V`); Verwendung der alternativen numerischen Symbole der Lokalität.

`%Ow` Nummer des Wochentags (Sonntag = 0); Verwendung der alternativen numerischen Symbole der Lokalität.

`%OW` Nummer der Woche (Montag ist der erste Tag der Woche); Verwendung der alternativen numerischen Symbole der Lokalität.

`%Oy` Jahr (Offset zu `%C`) in der alternativen Darstellung der Lokalität und mit den alternativen Symbolen der Lokalität.

Eine Umwandlungsanweisung, die aus Zwischenraum-Zeichen besteht, wird ausgeführt, indem der Input bis zum ersten Zeichen gelesen wird, das kein Zwischenraum-Zeichen ist (dieses Zeichen bleibt ungelesen), oder bis keine Zeichen mehr vorhanden sind.

Eine Umwandlungsanweisung, die aus einem gewöhnlichen Zeichen besteht, wird ausgeführt, indem das nächste Zeichen aus dem Puffer gelesen wird. Wenn das aus dem Puffer gelesene Zeichen nicht mit dem Zeichen der Umwandlungsanweisung übereinstimmt, schlägt diese fehl und das abweichende Zeichen sowie alle weiteren Zeichen bleiben ungelesen.

Eine Folge von Umwandlungsanweisungen, die aus `%n`, `%t`, Zwischenraum-Zeichen und Kombinationen davon besteht, wird ausgeführt, indem bis zum ersten Zeichen gelesen wird, das kein Zwischenraum-Zeichen ist (dieses Zeichen bleibt ungelesen), oder bis keine Zeichen mehr vorhanden sind.

Alle anderen Konvertierungs-Spezifikationen werden ausgeführt, indem solange Zeichen eingelesen werden, bis ein zur nächsten Umwandlungsanweisung passendes Zeichen gelesen wird (dieses bleibt im Puffer) oder bis keine Zeichen mehr vorhanden sind. Die gelesenen Zeichen werden dann mit den Werten in der Lokalität verglichen, die der Konvertierungs-Spezifikation entsprechen. Wenn der passende Wert in der Lokalität gefunden wird, werden die entsprechenden Strukturelemente der `tm`-Struktur auf die dieser Information entsprechenden Werte gesetzt.
Groß-/Kleinschreibung wird bei der Suche ignoriert, wenn es sich um den Vergleich von Elementen wie Wochentags- und Monatsnamen handelt.
Wenn kein passender Wert in der Lokalität gefunden wird, schlägt `strptime()` fehl und es werden keine weiteren Zeichen gelesen.

Returnwert Zeiger auf das Zeichen hinter dem letzten gelesenen Zeichen
bei Erfolg.

Nullzeiger sonst.

Hinweis Die spezielle Behandlung von Zwischenraum-Zeichen und viele „gleiche Formate“ sollen den Einsatz von identischen Format-Strings bei `strftime()` und `strptime()` erleichtern.

Siehe auch `scanf()`, `strftime()`, `time()`, `time.h`.

strrchr - letztes Vorkommen eines Zeichens in Zeichenkette ermitteln

Syntax #include <string.h>

char *strrchr(const char **s*, int *c*);

Beschreibung
`strrchr()` sucht das letzte Vorkommen des Zeichens *c* in der Zeichenkette *s* und liefert bei Erfolg einen Zeiger auf die gesuchte Position in *s*.

Das abschließende Nullbyte (`\0`) wird als Zeichen mitberücksichtigt.

Returnwert Zeiger auf die Position von *c* in der Zeichenkette *s*
bei Erfolg.

Nullzeiger wenn *c* in der Zeichenkette *s* nicht enthalten ist.

Hinweis Die Funktionen `strrchr()` und `rindex()` sind äquivalent.

Siehe auch `index()`, `rindex()`, `strchr()`, `string.h`.

strspn - Länge einer Teilzeichenkette berechnen

Syntax #include <string.h>

size_t strspn(const char *s1, const char *s2);

Beschreibung

`strspn()` berechnet ab Beginn der Zeichenkette *s1* die Länge des Segmentes, das ausschließlich Zeichen aus der Zeichenkette *s2* enthält.
Sobald ein Zeichen in *s1* mit keinem Zeichen in *s2* übereinstimmt, wird die Funktion beendet und die Segmentlänge zurückgeliefert.
Wenn bereits das erste Zeichen in *s1* mit keinem Zeichen in *s2* übereinstimmt, ist die Segmentlänge gleich 0.

Returnwert Ganzzahliger Wert
der die Segmentlänge (Anzahl passender Zeichen) ab Beginn der Zeichenkette *s1* angibt.

Hinweis Als Argumente werden Zeichenketten erwartet, die mit dem Nullbyte (`\0`) abgeschlossen sind.

Siehe auch `strcspn()`, `string.h`.

strstr - Teilzeichenkette in Zeichenkette suchen

Syntax #include <string.h>

char *strstr(const char *s1, const char *s2);

Beschreibung

`strstr()` sucht das erste Vorkommen der Zeichenkette *s2* (ohne das abschließende Nullbyte) in der Zeichenkette *s1*.

Returnwert
- Zeiger — auf den Beginn der gefundenen Zeichenkette in *s1*.
- Nullzeiger — wenn *s2* in *s1* nicht enthalten ist.
- Zeiger — auf den Beginn von *s1*, wenn *s2* die Länge 0 hat.

Hinweis Als Argumente werden Zeichenketten erwartet, die mit dem Nullbyte (`\0`) abgeschlossen sind.

Siehe auch `strchr()`, `string.h`.

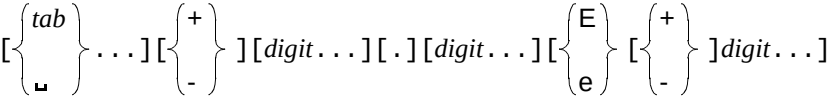
strtod - Zeichenkette in Gleitkommazahl (double) umwandeln

Syntax #include <stdlib.h>

double strtod(const char **s*, char ***endptr*);

Beschreibung

`strtod()` wandelt die Zeichenkette, auf die *s* zeigt, in eine Gleitkommazahl vom Typ `double` um. Die umzuwandelnde Zeichenkette kann wie folgt aufgebaut sein:



Für *tab* sind alle Zwischenraumzeichen zulässig (siehe Definition bei `isspace()`).

`strtod()` erkennt auch Zeichenketten, die mit Ziffern beginnen, dann aber mit beliebigen Zeichen enden. In diesem Fall schneidet `strtod()` zunächst den Ziffernteil ab und wandelt ihn in einen Gleitkommawert um.

Zusätzlich erhält man von `strtod()` über das zweite Argument *endptr* vom Typ `char **` einen Zeiger (**endptr*) auf das erste nicht umwandelbare Zeichen in der Zeichenkette *s*; jedoch nur, wenn *endptr* nicht als Nullzeiger übergeben wird.

Ist *endptr* ein Nullzeiger, wird `strtod()` wie die Funktion `atof()` ausgeführt:

`atof(s)` entspricht `strtod(s, (char **)NULL)` oder auch `strtod(s, NULL)`.

Wenn *endptr* kein Nullzeiger ist, wird ein Zeiger (**endptr*) auf das erste Zeichen in *s* zurückgeliefert, das die Umwandlung beendet.
Wenn überhaupt keine Umwandlung möglich ist, wird **endptr* auf die Anfangsadresse der Zeichenkette *s* gesetzt.

Returnwert

Gleitkommazahl vom Typ `double`	für Zeichenketten, die eine wie oben beschriebene Struktur haben und einen Zahlenwert darstellen, der im zulässigen Gleitkommabereich liegt.
0	für Zeichenketten, die nicht der oben beschriebenen Syntax entsprechen bzw. nicht mit umwandelbaren Zeichen beginnen.
`HUGE_VAL`	für Zeichenketten, deren Zahlenwert außerhalb des zulässigen Gleitkommabereichs liegt. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `strtod()` schlägt fehl, wenn gilt:

`ERANGE`	Der Returnwert verursacht einen Über- oder Unterlauf.
`EINVAL`	Umwandlung konnte nicht ausgeführt werden.

Hinweis Das Dezimalzeichen in der umzuwandelnden Zeichenkette wird durch die Lokalität (Kategorie `LC_NUMERIC`) beeinflusst. Voreingestellt ist der Punkt.

Siehe auch `atof()`, `atoi()`, `atol()`, `isspace()`, `strtol()`, `strtoul()`, `stdlib.h`.

strtok - Zeichenkette in Tokens zerlegen

Syntax #include <string.h>

char * strtok(char **s1*, const char **s2*);

Beschreibung

Mit `strtok()` lässt sich eine Gesamtzeichenkette *s1* in Teilzeichenketten – sog. "Tokens" – zerlegen, z.B. ein Satz in die einzelnen Wörter oder eine Quellprogrammanweisung in die kleinsten syntaktischen Einheiten. Der Zeiger auf *s1* darf nur beim ersten `strtok`-Aufruf übergeben werden. Ab dem zweiten Aufruf ist ein Nullzeiger anzugeben.

Beginn- und Endekriterium für jedes Token sind Trennzeichen (Separatoren), die in einer zweiten Zeichenkette *s2* anzugeben sind. Tokens können durch einen oder mehrere dieser Separatoren bzw. durch Beginn und Ende der Gesamtzeichenkette *s1* begrenzt sein. Typische Separatoren zwischen den Wörtern eines Satzes sind z.B. Leerzeichen, Doppelpunkt, Komma etc.

Pro Aufruf bearbeitet `strtok()` genau eine Teilzeichenkette. Der erste Aufruf liefert einen Zeiger auf den Beginn der ersten gefundenen Teilzeichenkette, die weiteren Aufrufe jeweils einen Zeiger auf den Beginn der nächsten Teilzeichenketten. Jede Teilzeichenkette schließt `strtok()` mit dem Nullbyte (`\0`) ab.

Bei jedem Aufruf kann eine andere Trennzeichenfolge *s2* angegeben werden.

`strtok()` ist nicht threadsicher. Verwenden Sie bei Bedarf die reentrante Funktion `strtok_r()`.

Returnwert Zeiger auf den Beginn einer Teilzeichenkette

Beim ersten Aufruf ein Zeiger auf die erste Teilzeichenkette, beim nächsten Aufruf ein Zeiger auf die nachfolgende Teilzeichenkette etc. `strtok()` schließt jede Teilzeichenkette in *s1* mit einem Nullbyte (`\0`) ab, wobei das jeweils erste gefundene Trennzeichen mit dem Nullbyte (`\0`) überschrieben wird.

Nullzeiger falls keine bzw. keine weitere Teilzeichenkette gefunden wurde.

Siehe auch `string.h.`, `strtok_r()`.

strtok_r - Zeichenkette threadsicher in Tokens zerlegen

Syntax #include <string.h>

char *strtok_r(char **s*, const char **sep*, char ***lasts*);

Beschreibung

Die Funktion `strtok_r()` ist die threadsichere Version von `strtok()`.

Die Funktion `strtok_r()` betrachtet die mit einem Nullbyte abgeschlossene Zeichenkette *s* als eine Folge von 0 oder mehr Teilzeichenketten - sog. "Tokens" . Die Tokens sind durch einen oder mehrere Separatoren getrennt, die in der Zeichenkette *sep* angegeben sind. Das Argument *lasts* zeigt auf einen vom Anwender bereitgestellten Zeiger, über den `strtok_r()` die für die weitere Bearbeitung derselben Zeichenkette notwendigen Informationen ermitteln kann.

Beim ersten Aufruf von `strtok_r()` zeigen *s* auf eine mit einem Nullbyte abgeschlossene Zeichenkette und *sep* auf eine mit einem Nullbyte abgeschlossene Zeichenkette mit Trennzeichen. Der Wert, auf den *lasts* zeigt, wird ignoriert. Die Funktion `strtok_r()` liefert einen Zeiger auf das erste Zeichen des Tokens zurück, überschreibt das erste gefundene Trennzeichen mit dem NULL-Zeichen (\0) und aktualisiert den Zeiger, auf den *lasts* zeigt.

Um weitere Teilzeichenketten zu ermitteln, ist in nachfolgenden Aufrufen für *s* ein Nullzeiger und für *lasts* der unveränderte Wert vom vorherigen Aufruf anzugeben. Das kann solange fortgesetzt werden, bis keine Token mehr übrigbleiben. In diesem Fall wird ein Nullzeiger zurückgegeben.

Bei jedem Aufruf kann eine andere Trennzeichenfolge *sep* angegeben werden.

Die Funktion `strtok_r()` gibt einen Zeiger auf das Token zurück. Konnte kein Token gefunden werden, wird ein Nullzeiger zurückgegeben.

Returnwert Zeiger auf das gefundene Token
bei Erfolg.

Nullzeiger wenn kein Token gefunden wird.

Siehe auch `strtok(), string()`.

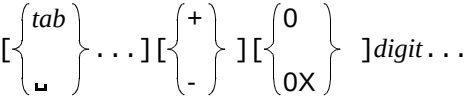
strtol - Zeichenkette in ganze Zahl (long) umwandeln

Syntax #include <stdlib.h>

long int strtol(const char *s, char **endptr, int base);

Beschreibung

`strtol()` wandelt die Zeichenkette, auf die *s* zeigt, in eine ganze Zahl vom Typ `long int` um. Die umzuwandelnde Zeichenkette kann wie folgt aufgebaut sein:



Für *tab* sind alle Zwischenraumzeichen zulässig (siehe Definition bei `isspace()`).

Für *digit* sind je nach der Basis (siehe *base*) die Ziffern 0 bis 9 und die Buchstaben a (oder A) bis z (oder Z) zulässig.

`strtol()` erkennt auch Zeichenketten, die mit umwandelbaren Ziffern (auch Oktal- bzw. Sedezimal-Ziffern) beginnen, dann aber mit beliebigen Zeichen enden. In diesem Fall schneidet `strtol()` zunächst den Ziffernteil ab und wandelt ihn um.

Zusätzlich erhält man von `strtol()` über das zweite Argument *endptr* vom Typ `char **` einen Zeiger auf das erste nicht umwandelbare Zeichen in der Zeichenkette *s*; jedoch nur, wenn *endptr* nicht als Nullzeiger übergeben wird.

Wenn überhaupt keine Umwandlung möglich ist, wird **endptr* auf die Anfangsadresse der Zeichenkette *s* gesetzt.

Ein drittes Argument *base* bestimmt die Basis (z.B. Dezimal-, Oktal- oder Sedezimal-Basis) für die Umwandlung.

base ist eine ganze Zahl von 0 bis 36. Von Basis 11 bis 36 werden die Buchstaben a (oder A) bis z (oder Z) in der umzuwandelnden Zeichenkette als Ziffern angenommen, und zwar mit den entsprechenden Werten 10 (a/A) bis 35 (z/Z).

Falls *base* gleich 0 ist, wird die Basis folgendermaßen aus dem Aufbau der Zeichenkette *s* bestimmt:

führende 0	Basis 8
führendes 0X bzw. 0x	Basis 16
sonst	Basis 10

Falls mit Parameter *base* = 16 gerechnet wird, werden die Zeichen 0X bzw. 0x nach einem evtl. Vorzeichen in der Zeichenkette *s* ignoriert.

Returnwert Ganzzahliger Wert vom Typ `long int`
für Zeichenketten, die eine wie oben beschriebene Struktur haben und einen Zahlenwert darstellen.

0 für Zeichenketten, die nicht der oben beschriebenen Syntax entsprechen.

`LONG_MAX` bzw. `LONG_MIN`
bei Überlauf, abhängig vom Vorzeichen.

Fehler `strtol()` schlägt fehl, wenn gilt:

`ERANGE` Der Returnwert verursacht einen Überlauf.

`EINVAL` Der Wert von *base* wird nicht unterstützt.

Hinweis Ist *endptr* ein Nullzeiger und *base* gleich 10, wird `strtol()` wie die Funktion `atol()` ausgeführt:

`atol(s)` entspricht `strtol(s, NULL, 10)`.

Siehe auch `atol()`, `atoi()`, `isalpha()`, `strtod()`, `strtoul()`, `stdlib.h`.

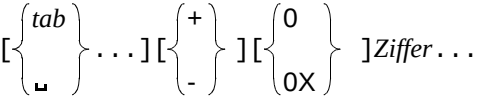
strtoll - Zeichenkette in ganze Zahl umwandeln (long long int)

Syntax #include <stdlib.h>

long long int strtoll(const char restrict *s, char ** restrict *zg*, int *base*);

Beschreibung

`strtoll()` wandelt eine EBCDIC-Zeichenkette, auf die *s* zeigt, in eine ganze Zahl vom Typ `long long int` um. Die umzuwandelnde Zeichenkette kann wie folgt aufgebaut sein:



Für *tab* sind alle Steuerzeichen für „Zwischenraum“ zulässig (siehe Definition bei `isspace()`).

Für *Ziffer* sind je nach der Basis (siehe *base*) die Ziffern 0 bis 9 und die Buchstaben a (oder A) bis z (oder Z) zulässig.

`strtoll()` erkennt auch Zeichenketten, die mit umwandelbaren Ziffern (auch Oktal- bzw. Sedezimal-Ziffern) beginnen, dann aber mit beliebigen Zeichen enden. In diesem Fall schneidet `strtoll()` zunächst den Ziffernteil ab und wandelt ihn um.

Zusätzlich erhält man von `strtoll()` über das zweite Argument *zg* vom Typ `char` ** einen Zeiger auf das erste nicht umwandelbare Zeichen in der Zeichenkette *s*; jedoch nur, wenn *zg* nicht als Nullzeiger übergeben wird.

Ein drittes Argument *base* bestimmt die Basis (z.B. Dezimal-, Oktal- oder Sedezimal-Basis) für die Umwandlung.

Die Funktion verfügt über folgende Parameter:

const char **s*
: Zeiger auf die umzuwandelnde EBCDIC-Zeichenkette.

char ***zg*
: Wenn *zg* kein Nullzeiger ist, wird ein Zeiger (**zg*) auf das erste Zeichen in *s* zurückgeliefert, das die Umwandlung beendet.
Wenn überhaupt keine Umwandlung möglich ist, wird **zg* auf die Anfangsadresse der Zeichenkette *s* gesetzt.

int *base*
: Ganze Zahl von 0 bis 36, die als Basis für die Berechnung verwendet werden soll.

  Von Basis 11 bis 36 werden die Buchstaben a (oder A) bis z (oder Z) in der umzuwandelnden Zeichenkette als Ziffern angenommen, und zwar mit den entsprechenden Werten 10 (a/A) bis 35 (z/Z).

Falls *base* gleich 0 ist, wird die Basis folgendermaßen aus dem Aufbau der Zeichenkette *s* bestimmt:

führende 0	Basis 8
führendes 0X bzw. 0x	Basis 16
sonst	Basis 10

Falls mit Parameter *base* = 16 gerechnet wird, werden die Zeichen 0X bzw. 0x nach einem evtl. Vorzeichen in der Zeichenkette *s* ignoriert.

Returnwert Ganzzahliger Wert vom Typ `long long int`
für Zeichenketten, die eine wie oben beschriebene Struktur haben und einen Zahlenwert darstellen.

0 für Zeichenketten, die nicht der oben beschriebenen Syntax entsprechen. Es wird keine Konvertierung durchgeführt. Wenn der Wert von *base* nicht unterstützt wird, wird `errno` auf `EINVAL` gesetzt.

`LLONG_MAX` bzw. `LLONG_MIN`
abhängig vom Vorzeichen.

`ULLONG_MAX`
bei Überlauf. `errno` wird auf `ERANGE` gesetzt.

Hinweise Ist *zg* ein Nullzeiger und *base* gleich 10, unterscheidet sich `strtoll()` von der Funktion `atoll()` nur durch die Fehlerbehandlung.

`atoll(s)` entspricht `strtoll(s, (char **)NULL, 10)`.

Siehe auch `atol()`, `atoll()`, `atoi()`, `strtol()`, `stroul()`, `stroull()`, `wcstol()`, `wcstoll()`, `wcstoul()`, `wcstoull()`

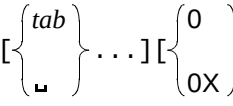
strtoul - Zeichenkette in ganze Zahl (unsigned long) umwandeln

Syntax #include <stdlib.h>

unsigned long int strtoul(const char **s*, char ***endptr*, int *base*);

Beschreibung

`strtoul()` wandelt die Zeichenkette, auf die *s* zeigt, in eine ganze Zahl vom Typ `unsigned long int` um. Die umzuwandelnde Zeichenkette kann wie folgt aufgebaut sein:



Für *tab* sind alle Zwischenraumzeichen zulässig (siehe Definition bei `isspace()`).

Für *digit* sind je nach der Basis (siehe *base*) die Ziffern 0 bis 9 und die Buchstaben a (oder A) bis z (oder Z) zulässig.

`strtoul()` erkennt auch Zeichenketten, die mit umwandelbaren Ziffern (auch Oktal- bzw. Sedezimal-Ziffern) beginnen, dann aber mit beliebigen Zeichen enden. In diesem Fall schneidet `strtoul()` zunächst den Ziffernteil ab und wandelt ihn um.

Zusätzlich erhält man von `strtoul()` über das zweite Argument *endptr* vom Typ `char **` einen Zeiger auf das erste nicht umwandelbare Zeichen in der Zeichenkette *s*; jedoch nur, wenn *endptr* nicht als Nullzeiger übergeben wird.

Wenn überhaupt keine Umwandlung möglich ist, wird **endptr* auf die Anfangsadresse der Zeichenkette *s* gesetzt.

Ein drittes Argument *base* bestimmt die Basis (z.B. Dezimal-, Oktal- oder Sedezimal-Basis) für die Umwandlung. *base* ist eine ganze Zahl von 0 bis 36.

Von Basis 11 bis 36 werden die Buchstaben a (oder A) bis z (oder Z) in der umzuwandelnden Zeichenkette als Ziffern angenommen, und zwar mit den entsprechenden Werten 10 (a/A) bis 35 (z/Z).

Falls *base* gleich 0 ist, wird die Basis folgendermaßen aus dem Aufbau der Zeichenkette *s* bestimmt:

führende 0	Basis 8
führendes 0X bzw. 0x	Basis 16
sonst	Basis 10

Falls mit Parameter *base* = 16 gerechnet wird, werden die Zeichen 0X bzw. 0x nach einem evtl. Vorzeichen in der Zeichenkette *s* ignoriert.

Returnwert Ganzzahliger Wert vom Typ `unsigned long`
für Zeichenketten, die eine wie oben beschriebene Struktur haben und einen Zahlenwert darstellen.

0 für Zeichenketten, die nicht der oben beschriebenen Syntax entsprechen.

größtmöglicher Wert (`unsigned long`)
bei Überlauf. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `strtoul()` schlägt fehl, wenn gilt:

`EINVAL` Der Wert von *base* wird nicht unterstützt.

`ERANGE` Der Returnwert verursacht einen Überlauf.

`EINVAL` Umwandlung konnte nicht ausgeführt werden.

Siehe auch `atol()`, `atoi()`, `isalpha()`, `strtol()`, `stdlib.h`.

strtoull - Zeichenkette in ganze Zahl umwandeln (unsigned long long)

Syntax #include <stdlib.h>

unsigned long long int strtoull(const char restrict *s, char **restrict *zg*, int *base*);

Beschreibung

`strtoull()` wandelt eine EBCDIC-Zeichenkette, auf die *s* zeigt, in eine ganze Zahl vom Typ `unsigned long long int` um. Die umzuwandelnde Zeichenkette kann wie folgt aufgebaut sein:

```
 ⌈tab⌉     ⌈0 ⌉
[{   }...][{  }]Ziffer...
 ⌊ ␣ ⌋     ⌊0X⌋
```

Für *tab* sind alle Steuerzeichen für „Zwischenraum“ zulässig (siehe Definition bei `isspace`).

Für *Ziffer* sind je nach der Basis (siehe *base*) die Ziffern 0 bis 9 und die Buchstaben a (oder A) bis z (oder Z) zulässig.

`strtoull()` erkennt auch Zeichenketten, die mit umwandelbaren Ziffern (auch Oktal- bzw. Sedezimal-Ziffern) beginnen, dann aber mit beliebigen Zeichen enden. In diesem Fall schneidet `strtoull()` zunächst den Ziffernteil ab und wandelt ihn um.

Zusätzlich erhält man von `strtoull()` über das zweite Argument *zg* vom Typ `char` ** einen Zeiger auf das erste nicht umwandelbare Zeichen in der Zeichenkette *s*; jedoch nur, wenn *zg* nicht als Nullzeiger übergeben wird.

Ein drittes Argument *base* bestimmt die Basis (z.B. Dezimal-, Oktal- oder Sedezimal-Basis) für die Umwandlung.

Die Funktion verfügt über folgende Parameter:

const char **s*
: Zeiger auf die umzuwandelnde EBCDIC-Zeichenkette.

char ***zg*
: Wenn *zg* kein Nullzeiger ist, wird ein Zeiger (**zg*) auf das erste Zeichen in *s* zurückgeliefert, das die Umwandlung beendet.
Wenn überhaupt keine Umwandlung möglich ist, wird **zg* auf die Anfangsadresse der Zeichenkette *s* gesetzt.

int *base*
: Ganze Zahl von 0 bis 36, die als Basis für die Berechnung verwendet werden soll.

  Von Basis 11 bis 36 werden die Buchstaben a (oder A) bis z (oder Z) in der umzuwandelnden Zeichenkette als Ziffern angenommen, und zwar mit den entsprechenden Werten 10 (a/A) bis 35 (z/Z).

Falls *base* gleich 0 ist, wird die Basis folgendermaßen aus dem Aufbau der Zeichenkette *s* bestimmt:

führende 0	Basis 8
führendes 0X bzw. 0x	Basis 16
sonst	Basis 10

Falls mit Parameter *base* = 16 gerechnet wird, werden die Zeichen 0X bzw. 0x in der Zeichenkette *s* ignoriert.

Returnwert Ganzzahliger Wert vom Typ `unsigned long long int`
für Zeichenketten, die eine wie oben beschriebene Struktur haben und einen Zahlenwert darstellen.

`0` für Zeichenketten, die nicht der oben beschriebenen Syntax entsprechen. Es wird keine Konvertierung durchgeführt. Wenn der Wert von *base* nicht unterstützt wird, wird `errno` auf `EINVAL` gesetzt.

`LLONG_MAX` bzw. `LLONG_MIN`
abhängig vom Vorzeichen.

`ULLONG_MAX`
bei Überlauf. `errno` wird auf `ERANGE` gesetzt.

Siehe auch `atol(), atoll(), atoi(), strtol(), strtoll(), stroul(), wcstol(), wcstoll(), wcstoul(), wcstoull()`

strupper - Zeichenkette in Großbuchstaben umwandeln *(BS2000)*

Syntax #include <string.h>

char *strupper(char **s1*, const char **s2*);

Beschreibung

`strupper()` kopiert die Zeichenkette *s2* einschließlich des Nullbytes (`\0`) in den Speicherbereich, auf den *s1* zeigt, und wandelt dabei die Kleinbuchstaben in Großbuchstaben um.

Wenn die Zeichenkette *s2* als Nullzeiger übergeben wird, entfällt der Kopiervorgang und in *s1* werden die Kleinbuchstaben in Großbuchstaben umgewandelt.

Wenn *s2* nicht als Nullzeiger übergeben wird, muss *s1* groß genug sein, um die Zeichenkette *s2* einschließlich des Nullbytes (`\0`) aufnehmen zu können.

Returnwert Zeiger auf die Ergebniszeichenkette *s1*.

Hinweis Als Argumente werden Zeichenketten erwartet, die mit dem Nullbyte (`\0`) abgeschlossen sind.

`strupper()` überprüft nicht, ob *s1* groß genug für das Ergebnis ist.

Bei sich überlappenden Speicherbereichen ist das Verhalten undefiniert.

Siehe auch `strlower()`, `tolower()`, `toupper()`.

strxfrm - Zeichenkette abhängig von LC_COLLATE umwandeln

Syntax #include <string.h>

size_t strxfrm(char *s1, const char *s2, size_t n);

Beschreibung

`strxfrm()` transformiert die Zeichenkette *s2* und schreibt die resultierende Zeichenkette in das Feld *s1*. Die Transformation läuft so ab, dass bei Anwendung von `strcmp()` auf zwei transformierte Zeichenketten dasselbe Resultat zurückgegeben wird, als ob `strcoll()` auf die beiden originalen Zeichenketten angewendet würde. Die Transformation hängt von der lokalen Einstellung der Kategorie `LC_COLLATE` des Programms ab (siehe `setlocale()`).

Maximal *n* Bytes (einschließlich des abschließenden Nullbytes) werden in das Feld kopiert, auf das *s1* zeigt. Wenn *n* gleich 0 ist, darf *s1* ein Nullzeiger sein. Wenn zwei sich überlappende Objekte kopiert werden, ist das Verhalten undefiniert.

Returnwert

Länge	der transformierten Zeichenkette (ohne das abschließende Nullbyte) bei Erfolg.
Wert ≥ *n*	der Inhalt des Felds *s1* ist undefiniert.

Da der Returnwert im Fehlerfall nicht festgelegt ist, können Fehler nur wie folgt festgestellt werden: `errno` wird auf 0 gesetzt, anschließend wird `strxfrm()` aufgerufen und `errno` geprüft. Wenn `errno` ungleich 0 ist, muss ein Fehler aufgetreten sein.

Fehler `strxfrm()` schlägt fehl, wenn gilt:

`EINVAL`	Das Argument *s2* enthält Zeichen, die außerhalb des Bereichs der Sortierreihenfolge liegen.

Hinweis

Als Argument *s2* wird eine Zeichenkette erwartet, die mit dem Nullbyte abgeschlossen ist

Die Zeichenkette *s2* wird durch `strxfrm` nicht verändert. Die Transformation wird in einem Arbeitsbereich durchgeführt.

Wenn der Returnwert größer oder gleich *n* ist, ist der Inhalt der Zeichenkette *s1* unbestimmbar, da kein Nullbyte geschrieben wurde.

Wenn in der aktuellen Lokalität einem Zeichen in der Zeichenkette *s2* der sedezimale Wert 0 zugeordnet ist, schließt dieses Zeichen als Nullbyte die transformierte Zeichenkette ab.

Siehe auch `setlocale()`, `strcoll()`, `strcmp()`, `string.h`.

swab - Bytes austauschen

Syntax #include <unistd.h>

void swab(const void *_src_, void *_dest_, ssize_t _nbytes_);

Beschreibung

`swab()` kopiert *nbytes* Bytes, auf die *src* zeigt, in das Objekt, auf das *dest* zeigt, wobei benachbarte Bytes vertauscht werden. Das Argument *nbytes* sollte gerade und nicht negativ sein. Wenn *nbytes* ungerade und positiv ist, dann kopiert und vertauscht `swab()` *nbytes-1* Bytes und die Anordnung des letzten Bytes ist undefiniert. Wenn *nbytes* negativ ist, dann macht `swab()` nichts. Bei überlappenden Argumenten ist das Verhalten von `swab()` undefiniert.

Siehe auch `unistd.h`.

swapcontext - Benutzerkontext wechseln

Syntax #include <ucontext.h>

int swapcontext (ucontext_t *_oucp_, const ucontext_t *_ucp_);

Beschreibung

siehe `makecontext()`.

swprintf - Langzeichen formatiert ausgeben

Syntax #include <wchar.h>

int swprintf(wchar_t *s, size_t *n*, const wchar_t **format* **[,** *arglist***])**;

Beschreibung

siehe `fwprintf()`.

swscanf - formatiert lesen

Syntax #include <wchar.h>

int swscanf(const wchar_t **s*, const wchar_t **format* [, *arglist*]);

Beschreibung

siehe `fwscanf()`.

symlink - symbolischen Verweis auf eine Datei erzeugen

Syntax

```
#include <unistd.h>

int symlink(const char *path1,, const char *path2);
```

Beschreibung

`symlink()` erzeugt einen symbolischen Verweis. Sein Name ist der Pfadname, auf den durch *path2* verwiesen wird. Dieser Pfadname darf nicht identisch sein mit dem Namen einer schon existierenden Datei bzw. eines symbolischen Verweises. Der Inhalt des symbolischen Verweises ist die Zeichenkette, auf die durch *path1* verwiesen wird. Ein symbolischer Verweis kann sich auch auf ein anderes Dateisystem beziehen. Die Datei, die durch *path1* bezeichnet wird, muss nicht vorhanden sein.

Die Datei, auf die der symbolische Verweis zeigt, wird verwendet, wenn mit dem Verweis eine `open()`-Operation durchgeführt wird. Ein `stat()` auf einen symbolischen Verweis liefert die Datei, auf die verwiesen wird, während ein `lstat()` Informationen über den Verweis selbst liefert. Dies kann zu überraschenden Ergebnissen führen, wenn ein symbolischer Verweis auf ein Verzeichnis erzeugt wird. Um in Programmen diese ungewünschten Effekte zu reduzieren, kann der Aufruf `readlink()` verwendet werden, um den Inhalt eines symbolischen Verweises zu lesen.

Returnwert

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

Fehler `symlink()` schlägt fehl, wenn gilt:

`EACCES`	Fehlende Schreiberlaubnis im Dateiverzeichnis, in dem der symbolische Verweis erstellt wurde. Die Sucherlaubnis für eine Komponente des Pfadpräfixes von *path2* existiert nicht.
`EEXIST`	Die Datei bzw. der symbolische Verweis, durch *path2* angegeben, existiert bereits.
`ENOTDIR`	Eine Komponente des Pfadpräfixes von *path2* ist kein Verzeichnis.
`EIO`	Beim Lesen oder Schreiben des Dateisystems trat ein Ein-/Ausgabe-Fehler auf.
`ELOOP`	Zu viele symbolische Verweise traten beim Übersetzen von *path2* auf.
`ENAMETOOLONG`	Die Länge von *path1* oder *path2* überschreitet `{PATH_MAX}`, oder die Länge einer Komponente von *path1* oder *path2* überschreitet `{NAME_MAX}`. Evtl. schlägt `symlink()` auch fehl, wenn die Auflösung des symbolischen Verweises ein Ergebnis erzeugt, dessen Länge `{PATH_MAX}` überschreitet.

ENOENT	Eine Komponente des Pfadnamenpräfixes von *path2* existiert nicht oder *path2* ist eine leere Zeichenkette.
ENOSPC	Das Verzeichnis, in dem der Eintrag für den neuen symbolischen Verweis erzeugt werden soll, kann nicht erweitert werden, da auf dem Dateisystem des Verzeichnisses kein Speicherplatz mehr frei ist.
	Der neue symbolische Verweis kann nicht erzeugt werden, da kein Speicherplatz mehr auf dem Dateisystem verfügbar ist, das den Verweis enthalten soll.
	Es gibt keine freien Indexeinträge auf dem Dateisystem, auf dem die Datei erzeugt werden soll.
EROFS	Der neue symbolische Verweis würde sich auf einem Dateisystem befinden, das nur lesbar ist.
Erweiterung	
EDQUOT	Das Verzeichnis, in dem sich der Eintrag für den neuen symbolischen Verweis befinden soll, kann nicht erweitert werden, da die maximale Anzahl der Plattenblöcke des Benutzers für das Dateisystem überschritten wurde.
	Der neue symbolische Verweis kann nicht erzeugt werden, da die maximale Anzahl der Plattenblöcke des Benutzers für das Dateisystem, welches den Verweis enthalten soll, überschritten wurde.
	Die maximale Anzahl von Indexeinträgen des Benutzers auf dem Dateisystem, auf dem die Datei erzeugt werden soll, wurde überschritten.
EFAULT	*path1* oder *path2* weisen über den zugewiesenen Adressraum des Prozesses hinaus.
ENOSYS	Das Dateisystem unterstützt keine symbolischen Verweise. ❑

Hinweis `symlink()` wird nur für POSIX-Dateien ausgeführt.

Siehe auch `lchown()`, `link()`, `lstat()`, `open()`, `readlink()`, `unistd.h`, Kommando `cp` in „POSIX-Kommandos“

sync - Superblock aktualisieren

Syntax

```
#include <unistd.h>

void sync(void);
```

Beschreibung

`sync()` bewirkt das Herausschreiben aller Daten im Speicher, die auf Platte/Diskette geschrieben werden sollen aber noch im Hauptspeicher gehalten werden. Hierzu gehören auch geänderte Superblöcke, geänderte Indexeinträge und verzögerte blockorientierte E/A-Dateien.

`sync()` soll von Programmen benutzt werden, die ein Dateisystem prüfen, beispielsweise `fsck()` oder `df()`. `sync()` ist vor einem Neuladen des Systems obligatorisch.

Das Schreiben ist bei Rückkehr von `sync()` nicht unbedingt schon beendet. Der Systemaufruf `fsync()` beendet das Schreiben vor der Rückkehr.

Returnwert Die Funktion gibt keine Werte zurück.

Siehe auch `fsync()`, `unistd.h`.

sysconf - numerischen Wert einer Systemvariablen ermitteln

Syntax #include <unistd.h>

long int sysconf(int *name*);

Beschreibung

Mit `sysconf()` kann der aktuelle numerische Wert einer konfigurierbaren Systemvariablen *name* ermittelt werden. Dies sind konfigurierbare Grenzwerte des Betriebssystems.

Die folgende Tabelle stellt die Systemvariablen aus den Dateien `limits.h`, `unistd.h` oder `time.h` zusammen, deren Werte mit `sysconf()` ermittelt werden können; die symbolischen Konstanten sind in `unistd.h` definiert. Sie enthalten die abfragbaren Werte für *name*:

Systemvariable	**Wert von *name* (Konstante)**
ARG_MAX	_SC_ARG_MAX
AIO_LISTIO_MAX	_SC_AIO_LISTIO_MAX
AIO_MAX	_SC_AIO_MAX
AIO_PRIO_DELTA_MAX	_SC_AIO_PRIO_DELTA_MAX
BC_BASE_MAX	_SC_BC_BASE_MAX
BC_DIM_MAX	_SC_BC_DIM_MAX
BC_SCALE_MAX	_SC_BC_SCALE_MAX
BC_STRING_MAX	_SC_BC_STRING_MAX
CHILD_MAX	_SC_CHILD_MAX
CLK_TCK	_SC_CLK_TCK
COLL_WEIGHTS_MAX	_SC_COLL_WEIGHTS_MAX
DELAYTIMER_MAX	_SC_DELAYTIMER_MAX
EXPR_NEST_MAX	_SC_EXPR_NEST_MAX
LINE_MAX	_SC_LINE_MAX
LOGIN_NAME_MAX	_SC_LOGIN_NAME_MAX
NGROUPS_MAX	_SC_NGROUPS_MAX
MQ_OPEN_MAX	_SC_MQ_OPEN_MAX
MQ_PRIO_MAX	_SC_MQ_PRIO_MAX
OPEN_MAX	_SC_OPEN_MAX
POSIX_ASYNCHRONOUS_IO	_SC_ASYNCHRONOUS_IO
_POSIX_FSYNC	_SC_FSYNC
_POSIX_MAPPED_FILES	_SC_MAPPED_FILES

Systemvariable	**Wert von *name* (Konstante)**
_POSIX_MEMLOCK	_SC_MEMLOCK
_POSIX_MEMLOCK_RANGE	_SC_MEMLOCK_RANGE
_POSIX_MEMORY_PROTECTION	_SC_MEMORY_PROTECTION
_POSIX_MESSAGE_PASSING	_SC_MESSAGE_PASSING
_POSIX_PRIORITIZED_IO	_SC_PRIORITIZED_IO
_POSIX_PRIORITY_SCHEDULING	_SC_PRIORITY_SCHEDULING
_POSIX_REALTIME_SIGNALS	_SC_REALTIME_SIGNALS
_POSIX_SEMAPHORES	_SC_SEMAPHORES
_POSIX_SHARED_MEMORY_OBJECTS	_SC_SHARED_MEMORY_OBJECTS
_POSIX_SYNCHRONIZED_IO	_SC_SYNCHRONIZED_IO
_POSIX_THREADS	_SC_THREADS
_POSIX_THREAD_ATTR_STACKADDR	_SC_THREAD_ATTR_STACKADDR
_POSIX_THREAD_ATTR_STACKSIZE	_SC_THREAD_ATTR_STACKSIZE
_POSIX_THREAD_PRIORITY_SCHEDULING	_SC_THREAD_PRIORITY_SCHEDULING
_POSIX_THREAD_PRIO_INHERIT	_SC_THREAD_PRIO_INHERIT
_POSIX_THREAD_PRIO_PROTECT	_SC_THREAD_PRIO_PROTECT
_POSIX_THREAD_PROCESS_SHARED	_SC_THREAD_PROCESS_SHARED
_POSIX_THREAD_SAFE_FUNCTIONS	_SC_THREAD_SAFE_FUNCTIONS
_POSIX_TIMERS	_SC_TIMERS
_POSIX2_C_BIND	_SC_2_C_BIND
_POSIX2_C_DEV	_SC_2_C_DEV
_POSIX2_C_VERSION	_SC_2_C_VERSION
_POSIX2_CHAR_TERM	_SC_2_CHAR_TERM
_POSIX2_FORT_DEV	_SC_2_FORT_DEV
_POSIX2_FORT_RUN	_SC_2_FORT_RUN
_POSIX2_LOCALEDEF	_SC_2_LOCALEDEF
_POSIX2_SW_DEV	_SC_2_SW_DEV
_POSIX2_UPE	_SC_2_UPE
_POSIX2_VERSION	_SC_2_VERSION
_POSIX_JOB_CONTROL	_SC_JOB_CONTROL
_POSIX_SAVED_IDS	_SC_SAVED_IDS
_POSIX_VERSION	_SC_VERSION
PTHREAD_DESTRUCTOR_ITERATIONS	_SC_THREAD_DESTRUCTOR_ITERATIONS

Systemvariable	**Wert von *name* (Konstante)**
PTHREAD_KEYS_MAX	_SC_THREAD_KEYS_MAX
PTHREAD_STACK_MIN	_SC_THREAD_STACK_MIN
PTHREAD_THREADS_MAX	_SC_THREAD_THREADS_MAX
PTHREAD_DESTRUCTOR_ITERATIONS	_SC_THREAD_DESTRUCTOR_ITERATIONS
PTHREAD_KEYS_MAX	_SC_THREAD_KEYS_MAX
PTHREAD_STACK_MIN	_SC_THREAD_STACK_MIN
PTHREAD_THREADS_MAX	_SC_THREAD_THREADS_MAX
RE_DUP_MAX	_SC_RE_DUP_MAX
RTSIG_MAX	_SC_RTSIG_MAX
SEM_NSEMS_MAX	_SC_SEM_NSEMS_MAX
SEM_VALUE_MAX	_SC_SEM_VALUE_MAX
STREAM_MAX	_SC_STREAM_MAX
SIGQUEUE_MAX	_SC_SIGQUEUE_MAX
TIMER_MAX	_SC_TIMER_MAX
TTY_NAME_MAX	_SC_TTY_NAME_MAX
TZNAME_MAX	_SC_TZNAME_MAX
Maximale Größe des Datenpuffers für die Funktionen getgrgid_r() und getgrnam_r()	_SC_GETGR_R_SIZE_MAX
Maximale Größe des Datenpuffers für die Funktionen getpwnam_r() und getpwuid_r()	_SC_GETPW_R_SIZE_MAX

Returnwert aktueller numerischer Wert von *name*
bei Erfolg.
Der Returnwert ist nicht niedriger als der entsprechende Wert in der Anwendung, wenn diese mit limits.h oder unistd.h der jeweiligen Implementierung übersetzt worden wäre. Der Wert ändert sich während der Lebensdauer des aufrufenden Prozesses nicht.

-1 wenn *name* ein ungültiger Wert ist.
In diesem Fall wird errno gesetzt, um den Fehler anzuzeigen.

wenn *name* keinen definierten Wert hat.
In diesem Fall bleibt errno unverändert.

Fehler sysconf() schlägt fehl, wenn gilt:

EINVAL Der Wert des Arguments *name* ist ungültig.

Hinweis Da alle Ergebniswerte im Erfolgsfall erlaubt sind, sollte eine Anwendung, die Fehlersituationen überprüfen will, die Variable `errno` auf 0 setzen und anschließend `sysconf()` aufrufen. Wenn die Funktion -1 zurückgibt, sollte die Anwendung prüfen, ob `errno` ungleich 0 ist.

Wenn der Wert von `sysconf(_SC_2_VERSION)` ungleich dem Wert der symbolischen Konstanten `_POSIX2_VERSION` ist, verhalten sich die Kommandos, die über `system()` oder `popen()` zur Verfügung stehen, möglicherweise nicht XPG4-konform. Unabhängig davon verhalten sich die in diesem Buch beschriebenen Schnittstellen auch dann XPG4-konform, wenn `sysconf(_SC_2_VERSION)` meldet, dass die Kommandos sich nicht mehr so verhalten, wie es im Standard definiert ist.

Siehe auch `pathconf()`, `limits.h`, `time.h`, `unistd.h`.

sysfs - Information über Dateisystemtyp abfragen *(Erweiterung)*

Syntax

```
#include <sys/fstyp.h>
#include <sys/fsid.h>

int sysfs(int opcode[, const char *fsname]| [, int fs_index, char *buf]);
```

Beschreibung

`sysfs()` gibt Informationen über die im System konfigurierten Dateisystemtypen zurück. Die Anzahl der von `sysfs()` akzeptierten Argumente hängt vom Wert *opcode* ab.

Die im C-Laufzeitsystem akzeptierten Werte für *opcode* sind:

`GETFSIND` übersetzt *fsname*, einen mit dem Nullbyte abgeschlossenen Dateisystemnamen, in einen Index der Dateisystemtypen.

`GETFSTYP` übersetzt *fs_index*, einen Index der Dateisystemtypen, in einen mit dem Nullbyte abgeschlossenen Dateisystemnamen und schreibt diesen in den Puffer, auf den *buf* zeigt. *buf* muss eine Mindestgröße von `FSTYPSZ` aufweisen (siehe `sys/fstyp.h`).

`GETNFSTYP` gibt die Gesamtzahl der im System konfigurierten Dateisystemtypen zurück.

Returnwert

Index des Dateisystemtyps
wenn *opcode* gleich `GETFSIND` ist und erfolgreicher Beendigung.

0 wenn *opcode* gleich `GETFSTYP` ist und erfolgreicher Beendigung.

Anzahl der konfigurierten Dateisystemtypen
wenn *opcode* gleich `GETNFSTYP` ist und erfolgreicher Beendigung.

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

Fehler `sysfs()` schlägt fehl, wenn gilt:

`EINVAL` *fsname* weist auf einen ungültigen Dateisystemnamen; *fs_index* ist null oder ungültig,
oder *opcode* ist ungültig,
oder es wurde versucht, auf eine BS2000-Datei zuzugreifen.

`EFAULT` *buf* oder *fsname* weisen über den zugewiesenen Adressraum des Prozesses hinaus.

Hinweis `sysfs()` greift nur auf POSIX-Dateien zu.

Siehe auch `sys/fstyp.h`, `sys/fsid.h`.

syslog - Meldung loggen

Syntax #include <syslog.h>

void syslog(int *priority*, const char **message*, .../* *argument* */);

Beschreibung

siehe `closelog()`.

system - Systemkommando ausführen

Syntax #include <stdlib.h>

int system(const char *_command_);

Beschreibung

`system()` führt das Systemkommando aus, das in der Zeichenkette *command* steht. *command* wird an einen Kommando-Interpreter übergeben. Je nach Wahl der Funktionalität wird *command* als POSIX- oder BS2000-Kommando interpretiert (siehe Abschnitt „Umfang der unterstützten C-Bibliothek“ auf Seite 18).

Wenn *command* ein POSIX-Kommando ist, verhält sich dessen Umgebung, als ob durch einen `fork`-Aufruf ein Sohnprozess erzeugt worden wäre und der Sohnprozess das `sh`-Kommando mit `execl()` wie folgt aufgerufen hätte:

```
execl(shell_path, "sh", "-c", command, (char *)0);
```

Für *shell_path* muss der Pfadname des `sh`-Kommandos eingesetzt werden.

`system()` kehrt erst zurück, wenn sich der Sohnprozess beendet hat und beeinflusst dessen Endestatus nicht.

BS2000
Wenn *command* ein BS2000-Kommando ist, wird das BS2000-Kommando in derselben Task ausgeführt, in der das `system()` aufrufende Programm läuft. Das aufrufende Programm wird entladen, wenn im `system`-Aufruf Programme oder Prozeduren gestartet werden (siehe auch „Hinweis“). ❑

Returnwert Endestatus des Kommando-Interpreters
wenn *command* kein Nullzeiger ist und das Kommando erfolgreich ausgeführt wurde. Der Endestatus des Kommando-Interpreters wird in dem Format geliefert, das durch `waitpid()` spezifiziert ist. Der Endestatus des Kommando-Interpreters entspricht dem des `sh`-Kommandos, außer wenn ein Fehler den Kommando-Interpreter an der Ausführung hindert, nachdem der Sohnprozess erzeugt wurde. Der Endestatus von `system()` ist dann, als ob der Kommando-Interpreter durch `exit(127)` oder `_exit(127)` beendet worden wäre.

≠ 0 wenn *command* ein Nullzeiger ist und ein Kommando-Interpreter vorhanden ist.

-1 wenn kein Sohnprozess erzeugt werden kann oder
wenn der Kommando-Interpreter keinen Endestatus hat.
`errno` wird gesetzt, um den Fehler anzuzeigen.

BS2000

0	wenn *command* erfolgreich ausgeführt wurde (Returnwert des BS2000-Kommandos: 0)
-1	wenn das BS2000-Kommando nicht erfolgreich ausgeführt wurde (Returnwert des Kommandos: Fehlernummer ≠ 0)
undefiniert	wenn nach dem BS2000-Kommando nicht in das Programm zurückverzweigt wird (siehe auch „Hinweis“). ❑

Fehler `system()` schlägt fehl, wenn gilt:

`EAGAIN`	Das System hat die notwendigen Ressourcen, um einen weiteren Prozess zu erzeugen, nicht zur Verfügung oder die systemspezifische Grenze für die Maximalzahl gleichzeitig ausgeführter Prozesse für das System oder eine einzelne Benutzernummer `{CHILD_MAX}` würde überschritten werden.

Erweiterung

`EINTR`	`system()` wurde durch ein Signal unterbrochen. ❑
`ENOMEM`	Es ist nicht genügend Speicherplatz verfügbar.

Hinweis Wenn der Returnwert von `system()` nicht -1 ist, kann dessen Wert durch die Makros entschlüsselt werden, die sowohl in `sys/wait.h` als auch in `stdlib.h` definiert sind.

Mit der folgenden Funktion kann ermittelt werden, ob eine XPG4-konforme Umgebung vorhanden ist: `sysconf(_SC_2_VERSION)`.

Solange `system()` auf die Beendigung des Sohnprozesses wartet, muss sie die Signale `SIGINT` und `SIGQUIT` ignorieren und `SIGCHLD` blockieren. Signale werden dann im ausgeführten Kommando so behandelt, wie es für `fork()` und `exec` beschrieben ist. Wenn zum Beispiel `SIGINT` abgefangen oder auf `SIG_DFL` gesetzt wird, wenn `system()` aufgerufen wird, dann wird der Sohnprozess mit der Einstellung `SIG_DFL` für `SIGINT` gestartet.

Wenn `SIGINT` und `SIGQUIT` im Vaterprozess ignoriert werden, treten keine Koordinationsprobleme auf (zum Beispiel wenn zwei Prozesse vom selben Terminal lesen), wenn das ausgeführte Kommando eines der Signale ignoriert oder abfängt. Dies ist normalerweise auch dann richtig, wenn der Benutzer ein Kommando an die Anwendung abgesetzt hat, das synchron ausgeführt werden soll (wie es beim Kommando `"!"` bei vielen interaktiven Anwendungen der Fall ist). In beiden Fällen sollte das Signal nur an den Sohnprozess und nicht an die Anwendung geliefert werden. In einer Situation kann das Ignorieren der Signale nicht den gewünschten Effekt haben. Dies ist dann der Fall, wenn die Anwendung `system()` dazu verwendet, einen für den Benutzer transparenten Prozess auszuführen. Wenn der Benutzer ein Unterbrechungszeichen eingibt (zum Beispiel `^C`), während `system()` so verwendet wird, könnte man erwarten, dass die Anwendung abgebrochen wird. Es wird jedoch nur das ausgeführte Kommando abgebrochen. Anwendungen, die

system() so verwenden, müssen den Endestatus von system() sorgfältig prüfen und feststellen, ob das ausgeführte Kommando erfolgreich beendet wurde. Wenn das Kommando fehlschlägt, müssen entsprechende Schritte in die Wege geleitet werden.

Wenn SIGCHLD blockiert wird, während auf die Beendigung des Sohnprozesses gewartet wird, verhindert dies, dass die Anwendung das Signal abfängt und den Status des Sohnprozesses von system() abfragt, bevor system() selbst den Status abfragen kann.

Der Kontext, in dem das Kommando ausgeführt wird, kann sich vom Kontext unterscheiden, in dem system() aufgerufen wurde. Wenn z.B. Dateideskriptoren, bei denen das Flag FD_CLOEXEC gesetzt ist, geschlossen werden, unterscheiden sich Prozessnummer und Vaterprozessnummer von system() und dem Kommando. Wenn das ausgeführte Kommando seine Umgebungsvariablen oder das aktuelle Dateiverzeichnis ändert, wird auch diese Veränderung nicht im Kontext des Aufrufs berücksichtigt.

Nach einem chroot-Aufruf kann sh nicht vorhanden sein.

Es gibt keine festgelegte Möglichkeit, wie eine Anwendung einen bestimmten Pfad für die Shell herausfinden kann. confstr() kann jedoch einen Wert für PATH zur Verfügung stellen, der sh-Kommandos sicher findet.

BS2000
Das BS2000-Kommando kann maximal 2048 Zeichen lang sein und muss nicht mit dem System-Schrägstrich (/) angegeben werden.

Nach einigen Kommandos (START-PROG, LOAD-PROG, CALL-PROCEDURE, DO, HELP-SDF) wird nicht in das aufrufende Programm zurückverzweigt. Lässt ein Programm solche vorzeitigen Programmbeendigungen zu, sollte es vor dem system-Aufruf die Puffer leeren (fflush()) bzw. die Dateien schließen.

system() übergibt die Zeichenkette *cmd* unverändert dem BS2000-Kommandoprozessor MCLP als Eingabe (siehe auch Handbuch „BS2000/OSD-BC Makroaufrufe an den Ablaufteil“). Es erfolgt keine Umsetzung in Großbuchstaben.

Siehe auch bs2system(), exec, fork(), pipe(), sysconf(), wait(), limits.h, signal.h, stdio.h, Kommando sh im Handbuch „POSIX Kommandos (BS2000/OSD)“.

tan - Tangens berechnen

Syntax #include <math.h>

double tan(double *x*);

Beschreibung

`tan()` berechnet für die Gleitkommazahl *x* (im zulässigen Gleitkommaintervall) die trigonometrische Funktion Tangens.

x ist die Gleitkommazahl, die den Winkel im Bogenmaß angibt.

Returnwert

tan(*x*)	Tangens von *x* bei Erfolg.
`+/-HUGE_VAL`	bei Überlauf. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `tan()` schlägt fehl, wenn gilt:

`ERANGE`	Der Wert von *x* verursacht einen Überlauf.

Siehe auch `atan()`, `cos()`, `sin()`, `tanh()`, `math.h`.

tanh - Tangens hyperbolicus berechnen

Syntax #include <math.h>

double tanh(double *x*);

Beschreibung

`tanh()` berechnet für die Gleitkommazahl *x* (im zulässigen Gleitkommaintervall) die Funktion Tangens hyperbolicus.

Returnwert

tanh(*x*)	Tangens hyperbolicus von *x* bei Erfolg.

Siehe auch `atan()`, `cos()`, `cosh()`, `sin()`, `sinh()`, `tan()`, `math.h`.

tcdrain - auf Übertragung einer Ausgabe warten

Syntax

```
#include <termios.h>

int tcdrain (int fildes);
```

Beschreibung

`tcdrain()` wartet, bis alle Ausgaben auf das Objekt übertragen worden sind, das durch *fildes* angegeben wird. *fildes* ist ein Dateideskriptor, der mit einem Terminal verbunden ist.

Das Signal `SIGTTOU` wird an die Prozessgruppe geschickt, wenn ein Prozess, der Mitglied einer Hintergrund-Prozessgruppe ist, versucht, `tcdrain()` mit dem Dateideskriptor *fildes*, der mit seinem steuernden Terminal verbunden ist, aufzurufen. Blockiert oder ignoriert der aufrufende Prozess `SIGTTOU`-Signale, darf er die Operation ausführen, und es wird kein Signal `SIGTTOU` gesendet.

Returnwert

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

Fehler `tcdrain()` schlägt fehl, wenn gilt:

`EBADF`	*fildes* ist kein gültiger Dateideskriptor.
`EINTR`	Während des Systemaufrufs `tcdrain()` wurde ein Signal abgefangen.
Erweiterung `EINVAL`	Es wurde versucht, auf eine BS2000-Datei zuzugreifen. ❑
`EIO`	Die Prozessgruppe des schreibenden Prozesses ist verwaist, und der schreibende Prozess ignoriert oder blockiert `SIGTTOU` nicht.
`ENOTTY`	Die mit *fildes* verbundene Datei ist kein Terminal.

Hinweis Auf blockorientierte Terminals hat `tcdrain()` keine Wirkung.

Siehe auch `tcflush()`, `termios.h`, `unistd.h`, Abschnitt „Allgemeine Terminalschnittstelle“ auf Seite 96.

tcflow - Datenübertragung anhalten oder erneut starten

Syntax

```
#include <termios.h>

int tcflow(int fildes, int action);
```

Beschreibung

`tcflow()` hält die Übertragung oder den Empfang von Daten zu oder von dem Objekt an, auf das *fildes* verweist, abhängig vom Wert *action*. *fildes* ist ein Dateideskriptor, der einem Terminal zugeordnet ist.

Ist *action* gleich `TCOOFF`, wird die Ausgabe angehalten. Ist *action* gleich `TCOON`, wird die Ausgabe neu gestartet. Ist *action* gleich `TCIOFF`, wird die Eingabe durch Übertragung des `STOP`-Zeichens angehalten. Ist *action* gleich `TCION`, wird die Eingabe durch Übertragung des `START`-Zeichens neu gestartet.
Die Voreinstellung beim Öffnen einer Terminaldatei ist, dass weder Eingabe noch Ausgabe angehalten sind.

Das Signal `SIGTTOU` wird an die Prozessgruppe geschickt, wenn ein Prozess, der Mitglied einer Hintergrund-Prozessgruppe ist, versucht, `tcflow()` mit dem Dateideskriptor *fildes*, der mit seinem steuernden Terminal verbunden ist, aufzurufen. Blockiert oder ignoriert der aufrufende Prozess `SIGTTOU`-Signale, darf er die Operation ausführen, und es wird kein Signal `SIGTTOU` gesendet.

Erweiterung
Bei Verbindung mit einem fernen Rechner werden alle Werte unterstützt. ❑

Returnwert

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

Fehler `tcflow()` schlägt fehl, wenn gilt:

`EBADF`	*fildes* ist kein gültiger Dateideskriptor.
`EINVAL`	*action* besitzt keinen unterstützten Wert.

Erweiterung

`EINVAL`	Es wurde versucht, auf eine BS2000-Datei zuzugreifen. ❑
`EIO`	Die Prozessgruppe des schreibenden Prozesses ist verwaist, und der schreibende Prozess ignoriert oder blockiert `SIGTTOU` nicht.
`ENOTTY`	Die mit *fildes* verbundene Datei ist kein Terminal.

Hinweis Auf blockorientierte Terminals hat `tcflow()` keine Wirkung.

Siehe auch `tcsendbreak()`, `termios.h`, `unistd.h`, Abschnitt „Allgemeine Terminalschnittstelle“ auf Seite 96.

tcflush - nicht übertragene Daten verwerfen

Syntax #include <termios.h>

int tcflush(int *fildes*, int *queue_selector*);

Beschreibung

fildes ist ein Dateideskriptor, der mit einem Terminal verbunden ist. Nach erfolgreicher Beendigung verwirft `tcflush()` Daten, die auf das Objekt, auf das *fildes* zeigt, geschrieben, aber noch nicht übertragen wurden, oder empfangen, aber noch nicht gelesen wurden, je nachdem, welchen Wert *queue_selector* hat.

Ist *queue_selector* gleich `TCIFLUSH`, werden empfangene, aber noch nicht gelesene Daten verworfen. Ist *queue_selector* gleich `TCOFLUSH`, werden geschriebene, aber noch nicht übertragene Daten verworfen. Ist *queue_selector* gleich `TCIOFLUSH`, werden sowohl empfangene, aber noch nicht gelesene, als auch geschriebene, aber noch nicht übertragene Daten verworfen.

Das Signal `SIGTTOU` wird an die Prozessgruppe geschickt, wenn ein Prozess, der Mitglied einer Hintergrund-Prozessgruppe ist, versucht, `tcflush()` mit dem Dateideskriptor *fildes*, der mit seinem steuernden Terminal verbunden ist, aufzurufen. Blockiert oder ignoriert der aufrufende Prozess `SIGTTOU`-Signale, darf er die Operation ausführen, und es wird kein Signal `SIGTTOU` gesendet.

Erweiterung
Bei Verbindung mit einem fernen Rechner werden alle Werte unterstützt. ❑

Returnwert

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

Fehler `tcflush()` schlägt fehl, wenn gilt:

`EBADF`	*fildes* ist kein gültiger Dateideskriptor.
`EINVAL`	*queue_selector* besitzt keinen unterstützten Wert.

Erweiterung

`EINVAL`	Es wurde versucht, auf eine BS2000-Datei zuzugreifen. ❑
`EIO`	Die Prozessgruppe des schreibenden Prozesses ist verwaist, und der schreibende Prozess ignoriert oder blockiert `SIGTTOU` nicht.
`ENOTTY`	Die mit *fildes* verbundene Datei ist kein Terminal.

Hinweis Auf blockorientierte Terminals hat `tcflush()` keine Wirkung.

Siehe auch `tcdrain()`, `termios.h`, `unistd.h`, Abschnitt „Allgemeine Terminalschnittstelle“ auf Seite 96.

tcgetattr - Terminalparameter ermitteln

Syntax

#include <termios.h>

int tcgetattr(int *fildes*, struct termios **termios_p*);

Beschreibung

`tcgetattr()` liest die Parameter des *fildes* zugewiesenen Terminals und schreibt sie in die `termios`-Struktur, auf die *termios_p* zeigt.

fildes ist ein Dateideskriptor, der einem Terminal zugeordnet ist.

termios_p ist ein Zeiger auf eine `termios`-Struktur.

Jeder Prozess darf `tcgetattr()` ausführen.

`tcgetattr()` kann von einem Hintergrundprozess aufgerufen werden; die Terminaleigenschaften können danach von einem Vordergrundprozess geändert werden.

Erweiterung
Die Ausgabe-Baudrate entspricht immer der Eingabe-Baudrate, nämlich 38400 (weitere Einzelheiten siehe `tcsetattr()`). ❑

Unterstützt das Terminal keine aufgespalteten Baudraten, ist die Eingabe-Baudrate, die in die `termios`-Struktur geschrieben wird, gleich null.

Returnwert

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

Fehler

`tcgetattr()` schlägt fehl, wenn gilt:

`EBADF`	*fildes* ist kein gültiger Dateideskriptor.
Erweiterung `EINVAL`	Es wurde versucht, auf eine BS2000-Datei zuzugreifen. ❑
`ENOTTY`	Die mit *fildes* verbundene Datei ist kein Terminal.

Siehe auch

`tcsetattr()`, `termios.h`, Abschnitt „Allgemeine Terminalschnittstelle“ auf Seite 96.

tcgetpgrp - Vordergrund-Prozessgruppennummer ermitteln

Syntax

```
#include <unistd.h>

Optional
#include <sys/types.h> ❑

pid_t tcgetpgrp(int fildes);
```

Beschreibung

`tcgetpgrp()` liefert den Wert der Vordergrund-Prozessgruppennummer, die mit einem Terminal verbunden ist.

Existiert keine Vordergrund-Prozessgruppe, liefert `tcgetpgrp()` einen Wert größer als 1, der mit keiner Prozessgruppennummer einer vorhandenen Prozessgruppe übereinstimmt.

`tcgetpgrp()` kann von einem Prozess aufgerufen werden, der Mitglied einer Hintergrund-Prozessgruppe ist; die Information kann jedoch nachträglich von einem Prozess geändert werden, der Mitglied einer Vordergrund-Prozessgruppe ist.

Returnwert

Wert der Vordergrund-Prozessgruppennummer, die mit dem Terminal verbunden ist,	bei Erfolg.
-1	bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler

`tcgetpgrp()` schlägt fehl, wenn gilt:

`EBADF`	*fildes* ist kein gültiger Dateideskriptor.
Erweiterung `EINVAL`	Es wurde versucht, auf eine BS2000-Datei zuzugreifen. ❑
`ENOTTY`	Der aufrufende Prozess besitzt kein steuerndes Terminal, oder die Datei ist nicht das steuernde Terminal.

Siehe auch `setsid()`, `setpgid()`, `tcsetpgrp()`, `sys/types.h`, `unistd.h`.

tcgetsid - Sitzungsnummer des angegebenen Terminals ermitteln

Syntax #include <termios.h>

pid_t tcgetsid(int *fildes*);

Beschreibung

`tcgetsid()` liefert die Prozessgruppennummer der Sitzung, die durch das in *fildes* angegebene Terminal gesteuert wird.

Returnwert Prozessgruppennummer der Sitzung, die mit dem angegebenen Terminal verbunden ist bei Erfolg.

`(pid_t)`-1 sonst. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `tcgetsid()` schlägt fehl, wenn gilt:

`EACCES` Dem Argument *fildes* ist kein steuerndes Terminal zugeordnet.

`EBADF` Das *fildes*-Argument ist kein gültiger Dateideskriptor.

`ENOTTY` Die Datei *fildes* ist kein Terminal.

Siehe auch `termios.h`.

tcsendbreak - serielle Datenübertragung unterbrechen

Syntax

```
#include <termios.h>

int tcsendbreak(int fildes, int duration);
```

Beschreibung

Erweiterung
Abweichend vom XPG4 hat diese Funktion keine Wirkung und kehrt zurück, ohne eine Aktion ausgeführt zu haben. ❑

Returnwert

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

Fehler `tcsendbreak()` schlägt fehl, wenn gilt:

`EBADF`	*fildes* ist kein gültiger Dateideskriptor.
Erweiterung	
`EINVAL`	Es wurde versucht, auf eine BS2000-Datei zuzugreifen. ❑
`EIO`	Die Prozessgruppe des schreibenden Prozesses ist verwaist, und der schreibende Prozess ignoriert oder blockiert `SIGTTOU` nicht.
`ENOTTY`	Die mit *fildes* verbundene Datei ist kein Terminal.

Siehe auch `termios.h`, `unistd.h`, Abschnitt „Allgemeine Terminalschnittstelle“ auf Seite 96.

tcsetattr - Terminalparameter setzen

Syntax #include <termios.h>

int tcsetattr(int *fildes*, int *optional_actions*, const struct termios **termios_p*);

Beschreibung

Die Funktion `tcsetattr()` setzt die Parameter für das Terminal, das durch den Dateideskriptor *fildes* angesprochen wird. Sie legt diese wie folgt in der `termios`-Struktur ab, auf die *termios_p* zeigt:

Ist *optional_actions* gleich `TCSANOW`, wird die Änderung sofort vorgenommen.
Ist *optional_actions* gleich `TCSADRAIN`, wird die Änderung vorgenommen, nachdem alle Ausgaben, die auf *fildes* geschrieben wurden, übertragen worden sind. Diese Funktion sollte verwendet werden, wenn Parameter geändert werden, die die Ausgabe beeinflussen.
Ist *optional_actions* gleich `TCSAFLUSH`, wird die Änderung vorgenommen, nachdem alle Ausgaben, die auf *fildes* geschrieben wurden, übertragen worden sind, und alle Eingaben, die bis dahin empfangen, aber noch nicht gelesen wurden, werden verworfen, bevor die Änderungen vorgenommen werden.

Ist die Ausgabe-Baudrate, die in der `termios`-Struktur abgelegt ist, auf die *termios_p* zeigt, gleich null, bewirkt der Aufruf von `tcsetattr()` eine Beendigung der Terminalverbindung.

Ist dieser Wert ungleich null, sind alle Werte in der `termios`-Struktur wirkungslos. Sind auch die übrigen Werte in der `termios`-Struktur wirkungslos, wird -1 zurückgegeben und `errno` auf `EINVAL` gesetzt.

Ist die Eingabe-Baudrate, die in der `termios`-Struktur abgelegt ist, auf die *termios_p* zeigt, gleich null, entspricht die Eingabe-Baudrate, die in der Hardware gesetzt wird, der Ausgabe-Baudrate, die in der `termios`-Struktur abgelegt ist.

Die Funktion `tcsetattr()` kehrt erfolgreich zurück, wenn sie einen Teil der angeforderten Aktionen ausgeführt hat, auch wenn einige nicht ausgeführt werden konnten.
`tcsetattr()` setzt alle Attribute, die von der Implementierung unterstützt werden, und lässt alle nicht unterstützten unverändert. Konnte keine Aktion ausgeführt werden, wird -1 zurückgegeben und `errno` auf `EINVAL` gesetzt. Sind Eingabe- und Ausgabe-Baudrate unterschiedlich und eine Kombination, die nicht von der Hardware unterstützt wird, wird keine Baudrate geändert. Ein nachfolgender `tcgetattr`-Aufruf gibt den aktuellen Status des Terminals zurück, der die vorgenommenen Änderungen und die unveränderten Werte des vorhergehenden `tcgetattr`-Aufrufs wiedergibt. Die Funktion `tcsetattr()` verändert die Werte der `termios`-Struktur nicht, ganz gleich, ob sie sie tatsächlich übernimmt oder nicht.

Nur ein `tcsetattr`-Aufruf oder das Schließen des letzten mit dem Terminal verbundenen Dateideskriptors im System kann verursachen, dass die in diesem Handbuch angegebenen Terminal-Parameter geändert werden.

Das Signal SIGTTOU wird an die Prozessgruppe geschickt, wenn ein Prozess, der Mitglied einer Hintergrund-Prozessgruppe ist, versucht, tcsetattr() mit dem Dateideskriptor *fildes*, der mit seinem steuernden Terminal verbunden ist, aufzurufen. Blockiert oder ignoriert der aufrufende Prozess SIGTTOU-Signale, darf er die Operation ausführen, und es wird kein Signal SIGTTOU gesendet.

Returnwert

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

Fehler tcsetattr() schlägt fehl, wenn gilt:

EBADF	*fildes* ist kein gültiger Dateideskriptor.
EINVAL	*optional_actions* besitzt keinen unterstützten Wert.
Erweiterung EINVAL	Es wurde versucht, auf eine BS2000-Datei zuzugreifen. ❑
EIO	Die Prozessgruppe des schreibenden Prozesses ist verwaist, und der schreibende Prozess ignoriert oder blockiert SIGTTOU nicht.
ENOTTY	Die mit *fildes* verbundene Datei ist kein Terminal.

Hinweis Wenn eine Anwendung versucht, die Baudraten zu verändern, dann sollte sie zuerst tcsetattr() aufrufen und danach tcgetattr(), um zu bestimmen, welche Baudraten tatsächlich ausgewählt wurden.

Siehe auch cfgetispeed(), tcgetattr(), termios.h, unistd.h, Abschnitt „Allgemeine Terminalschnittstelle“ auf Seite 96.

tcsetpgrp - Vordergrund-Prozessgruppennummer setzen

Syntax

#include <unistd.h>

Optional
#include <sys/types.h> ❑

int tcsetpgrp(int *fildes*, pid_t *pgid_id*);

Beschreibung

Hat der Prozess ein steuerndes Terminal, setzt `tcsetpgrp()` die Vordergrund-Prozessgruppennummer, die zu diesem Terminal gehört, auf den Wert *pgid_id*. Die Datei des durch *fildes* angegebenen Terminals muss das steuernde Terminal des aufrufenden Prozesses sein. Das steuernde Terminal muss mit der Sitzung des aufrufenden Prozesses verbunden sein. Der Wert von *pgid_id* muss der Prozessgruppennummer eines Prozesses entsprechen, der sich in derselben Sitzung wie der aufrufende Prozess befindet.

Returnwert

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

Fehler

`tcsetpgrp()` schlägt fehl, wenn gilt:

`EBADF`	*fildes* ist kein gültiger Dateideskriptor.
`EINVAL`	*pgid_id* ist keine gültige Prozessgruppennummer.

Erweiterung

`EINVAL`	Es wurde versucht, auf eine BS2000-Datei zuzugreifen. ❑
`ENOTTY`	Der aufrufende Prozess besitzt kein steuerndes Terminal oder das steuernde Terminal ist nicht länger mit der Sitzung des aufrufenden Prozesses verbunden.
`EPERM`	Der Wert von *pgid_id* entspricht nicht der Prozessgruppennummer eines Prozesses in derselben Sitzung wie der aufrufende Prozess.

Siehe auch `tcgetpgrp()`, `sys/types.h`, `unistd.h`.

tdelete - Knoten aus Binärbaum löschen

Syntax #include <search.h>

void *tdelete(const void **key*, void ***rootp*, int (**compar*) (const void *, const void *));

Beschreibung

Siehe `tsearch()`.

tell - aktuellen Wert des Lese-/Schreibzeigers ermitteln *(BS2000)*

Syntax

```
#include <stdio.h>

long tell(int fildes);
```

Beschreibung

`tell()` liefert den aktuellen Wert des Lese-/Schreibzeigers für die Datei mit dem Dateideskriptor *fildes*. `tell()` lässt sich auf Binärdateien (PAM, INCORE) und Textdateien (SAM, ISAM) anwenden. SAM-Dateien werden mit elementaren Funktionen stets als Textdateien verarbeitet.
fildes ist der Dateideskriptor der Datei, für die der aktuelle Wert des Lese-/Schreibzeigers bestimmt werden soll.

Returnwert

aktueller Wert des Lese-/Schreibzeigers
: Anzahl Bytes, die der Lese-/Schreibzeiger vom Dateianfang entfernt ist, in Binärdateien, bei Erfolg.

absolute Position
: des Lese-/Schreibzeigers in Textdateien, bei Erfolg.

-1
: bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen (z.B. `tell()` nicht erlaubt, Block/Satzanzahl zu groß).

Hinweis

Die Aufrufe `tell(`*fildes*`)` und `lseek(`*fildes*`, OL, SEEK_CUR)` sind äquivalent.
`tell()` ist nicht anwendbar auf Systemdateien (SYSDTA, SYSLST, SYSOUT).

Da die Informationen über die Dateiposition in einem 4 Byte langen Feld abgelegt werden, ergeben sich für die Größe von SAM- und ISAM-Dateien folgende Einschränkungen bei der Bearbeitung mit `tell()`/`lseek()`:

SAM-Datei

Satzlänge	≤ 2048 Byte
Satzanzahl/Block	≤ 256
Blockanzahl	≤ 2048

ISAM-Datei

Satzlänge	≤ 32 KByte
Satzanzahl	≤ 32 K

Siehe auch `lseek()`, `fseek()`, `ftell()`, `stdio.h`.

telldir - Position des Lese-/Schreibzeigers im Dateiverzeichnisstrom ermitteln

Syntax #include <dirent.h>

long int telldir(DIR **dirp*);

Beschreibung

`telldir()` liefert die aktuelle Position, die dem angegebenen Dateiverzeichnisstrom zugeordnet ist.

Wenn die letzte Operation auf dem Dateiverzeichnisstrom ein `seekdir()` war, gibt `telldir()` die Position zurück, die im *loc*-Argument des `seekdir()`-Aufrufs angegeben war.

Returnwert aktuelle Position bei Erfolg

Erweiterung
-1 bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen. ❑

Fehler `telldir()` schlägt fehl, wenn gilt:

Erweiterung
`EBADF` Der dem Dateiverzeichnis zugeordnete Dateideskriptor ist nicht mehr gültig. Dieser Fehler entsteht, wenn das Dateiverzeichnis geschlossen wurde. ❑

Hinweis `telldir()` wird nur für POSIX-Dateien ausgeführt.

Siehe auch `readdir()`, `seekdir()`, `dirent.h`.

tempnam - Pfadnamen für temporäre Datei erzeugen

Syntax #include <stdio.h>

char *tempnam(const char **dir*, const char **pfx*);

Beschreibung

`tempnam()` erzeugt einen Pfadnamen, der für eine temporäre Datei genutzt werden kann.

`tempnam()` ermöglicht die Steuerung der Dateiverzeichniswahl.

dir zeigt auf den Namen des Dateiverzeichnisses, in dem die Datei erstellt werden soll. Wenn die Umgebungsvariable `TMPDIR` gesetzt ist, wird das dort angegebene Dateiverzeichnis verwendet, sonst das unter **dir* genannte. Wenn *dir* der Nullzeiger ist und das Dateiverzeichnis `{P_tmpdir}` kein zugreifbares Dateiverzeichnis bezeichnet, werden die Dateinamen mit dem Verzeichnisnamen `/tmp` erzeugt. Falls auch dieser nicht zugreifbar ist, wird 0 zurückgegeben.

In `stdio.h` ist `P_tmpdir` mit `"/var/tmp"` als das Dateiverzeichnis definiert, in dem die temporäre Datei angelegt wird.

Bei vielen Anwendungen ist es vorteilhaft, wenn die temporären Dateien bestimmte bevorzugte Anfangsbuchstaben in ihren Namen aufweisen. Hierfür verwendet man das Argument *pfx*. Dieses Argument kann ein Nullzeiger sein oder auf eine Zeichenkette von maximal fünf Bytes zeigen, die als die ersten Bytes des Namens der temporären Datei eingesetzt werden.

Der von `tempnam()` erzeugte Namensteil besteht aus zwei Teilen: Der erste Teil besteht aus drei Großbuchstaben (AAA, BAA, ..., ZAA, ZBA, ..., ZZZ). Der zweite Teil besteht aus einem Buchstaben und den fünf letzten Zeichen der Prozessnummer. Falls die Prozessnummer aus weniger als fünf Zeichen besteht, wird sie mit führenden Nullen zu fünf Zeichen ergänzt. Ingesamt ergibt sich also zum Beispiel: `/var/tmp/AAAa00123`.

`tempnam()` verwendet `malloc()`, um Speicherplatz für den erzeugten Dateinamen zu erhalten, und gibt einen Zeiger auf diesen Bereich zurück. Daher kann jeder von `tempnam()` zurückgegebene Zeigerwert als Argument für `free()` dienen (siehe `malloc()`). Wenn `tempnam()` aus irgendeinem Grunde das erwartete Ergebnis nicht liefern kann, d.h., wenn `malloc()` erfolglos war oder kein geeignetes Dateiverzeichnis gefunden wurde, wird ein Nullzeiger zurückgegeben.

`tempnam()` ist erfolglos, wenn nicht genug Speicherplatz vorhanden ist.

Returnwert Zeiger auf eine Zeichenkette, die den generierten Pfadnamen enthält, bei Erfolg.

Nullzeiger bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

0	wenn `/tmp` nicht zugreifbar ist, oder wenn die Variable PROGRAM-ENVIRONMENT nicht auf SHELL gesetzt ist.

Fehler `tempnam()` schlägt fehl, wenn gilt:

`ENOMEM`	Es ist nicht genügend Speicherplatz für den neuen Pfadnamen vorhanden.
Erweiterung `EINVAL`	Es wurde versucht, auf eine BS2000-Datei zuzugreifen. ❑

Hinweis `tempnam()` wird nur für POSIX-Dateien ausgeführt.

`tempnam()` generiert bei jedem Aufruf einen anderen Pfadnamen.

Dateien, die mit `tempnam()` und entweder von `fopen()` oder `creat()` erstellt wurden, sind nur insofern temporär, weil sie sich in einem Dateiverzeichnis befinden, das für temporären Gebrauch bestimmt ist, und weil ihre Namen eindeutig sind. Der Benutzer ist dafür verantwortlich, die Datei zu löschen, wenn diese nicht mehr gebraucht wird. Wenn diese Funktion mehr als `{TMP_MAX}`-mal (definiert in `stdio.h`) in einem einzigen Prozess aufgerufen wird, werden vorher benutzte Namen wieder verwendet.

Es ist möglich, dass während des Zeitraums von der Erstellung eines Pfadnamens bis zum Öffnen der Datei ein anderer Prozess eine Datei mit dem gleichen Namen erstellt. Dies kann jedoch nicht eintreten, wenn der andere Prozess `tempnam()` oder `mktemp()` verwendet und der Pfadname so gewählt wird, dass seine Duplizierung auf andere Weise unwahrscheinlich ist.

Siehe auch `fopen()`, `free()`, `open()`, `tmpfile()`, `tmpnam()`, `unlink()`, `stdio.h`.

tfind - Knoten in Binärbaum suchen

Syntax #include <search.h>

void *tfind(const void **key*, void *const **rootp*, int (**compar*) (const void *, const void *));

Beschreibung

Siehe `tsearch()`.

_ _TIME_ _ - Makro für Übersetzungszeitpunkt

Syntax _ _TIME_ _

Beschreibung

Dieses Makro generiert die Übersetzungsuhrzeit einer Quelldatei als Zeichenkette in der Form:

"*hh*:*mm*:*ss*\0"

Dabei bedeuten:

hh Stunden

mm Minuten

ss Sekunden

Hinweis Das Format der Zeitangabe entspricht der Funktion `asctime()`.

Dieses Makro muss in keiner Include-Datei definiert werden. Sein Name wird vom Compiler erkannt und ersetzt.

Siehe auch `asctime()`, `_ _DATE_ _`.

time - Zeit seit Epochenwert ermitteln

Syntax

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

time_t time(time_t *tloc);
```

Beschreibung

`time()` liefert die aktuelle Zeit (Ortszeit) als Anzahl der Sekunden, die seit 00:00:00 UTC (Universal Time Coordinated, 1. Januar 1970) vergangen sind.

Wenn *tloc* ungleich null ist, wird zusätzlich der Returnwert an die Stelle gespeichert, auf die *tloc* zeigt.

BS2000
`time()` liefert die aktuelle Zeit (Ortszeit) als Anzahl der Sekunden, die seit dem 1. Januar 1950 00:00:00 vergangen sind. ❑

Returnwert Zeit in Sekunden (siehe oben) bei Erfolg.

`(time_t)-1` bei Fehler.
`errno` wird gesetzt, um den Fehler anzuzeigen.

Hinweis `time()` scheitert und seine Aktionen sind undefiniert, wenn *tloc* auf eine unzulässige Adresse zeigt.

Siehe auch `ctime()`, `time.h`.

times - Prozesszeit ermitteln

Syntax #include <sys/times.h>

clock_t times(struct tms **buffer*) ;

Beschreibung

`times()` füllt die Struktur `tms`, auf die *buffer* zeigt, mit Informationen über Laufzeiten (siehe `sys/times.h`).

Alle Zeitangaben werden im Raster Zeittakteinheiten definiert.

Die Laufzeiten von beendeten Kindprozessen werden in die Komponenten `tms_cutime` und `tms_cstime` des Vaterprozesses aufgenommen, sobald die Funktion `wait()` die Prozessnummer dieses beendeten Kindprozesses liefert. Wenn ein Kindprozess nicht auf seine Kindprozesse wartet, werden deren Zeiten nicht mit aufgenommen.

- Die Komponente `tms_utime` ist die Rechenzeit, die für die Ausführung von Benutzeranweisungen des aufrufenden Prozesses verbraucht wurde.
- Die Komponente `tms_stime` ist die Rechenzeit, die für die Ausführung von Systemanweisungen des aufrufenden Prozesses verbraucht wurde.
- Die Komponente `tms_cutime` ist die Summe der Zeiten `tms_utime` und `tms_cutime` der Kindprozesse.
- Die Komponente `tms_cstime` ist die Summe der Zeiten `tms_stime` und `tms_cstime` der Kindprozesse.

Returnwert abgelaufene Echtzeit in Zeittakteinheiten seit einem bestimmten Zeitpunkt (z.B. seit dem Einschalten des Systems). Dieser Zeitpunkt ändert sich nicht von einem Aufruf der Funktion `times()` innerhalb eines Prozesses zu einem anderen. Das Ergebnis kann den möglichen Wertebereich des Typs `clock_t` überschreiten (Überlauf).

`(clock_t)`-1 bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Hinweis Portable Anwendungen sollten die Funktion `sysconf(_SC_CLK_TCLK)` verwenden, um die Anzahl der Zeittakteinheiten pro Sekunde zu bestimmen, da sich diese von System zu System unterscheiden kann.

Siehe auch `exec`, `fork()`, `sysconf()`, `time()`, `wait()`, `sys/times.h`.

timezone - Variable für Differenz zwischen Ortszeit und UTC

Syntax

```
#include <time.h>

extern long int timezone;
```

Beschreibung

Die externe Variable `timezone` enthält die Differenz, gemessen in Sekunden, zwischen UTC (Universal Time Coordinated, 1. Januar 1970) und der lokalen Standardzeit. Die Voreinstellung für `timezone` ist 0 (UTC).

Die Datei `/usr/lib/locale/` *language*`/LC_TIME` enthält umgebungsspezifische Daten- und Zeitinformationen.

Hinweis

Das Ändern der Zeit während des Zeitraums der Änderung von `timezone` nach `altzone` oder umgekehrt kann unvorhersehbare Ergebnisse hervorrufen.
Der Systemverwalter muss das Start- und Enddatum der Sommerzeit jährlich ändern, wenn das Format des Julianischen Kalenders verwendet wird.

Siehe auch `altzone`, `asctime()`, `ctime()`, `daylight`, `environ`, `gmtime()`, `localtime()`, `mktime()`, `strftime()`, `tzname`, `tzset()`.

tmpfile - temporäre Datei erzeugen

Syntax

```
#include <stdio.h>

FILE *tmpfile(void);
```

Beschreibung

`tmpfile()` erzeugt eine temporäre Datei und öffnet einen dazugehörigen Datenstrom.

BS2000
`tmpfile()` erzeugt eine binäre SAM-Datei mit Standardattributen. ❑

Die Datei wird automatisch wieder gelöscht, wenn alle Verweise auf diese Datei geschlossen worden sind. Die Datei wird wie durch `fopen()` zum Aktualisieren geöffnet (`w+`).

In `stdio.h` ist `{P_tmpdir}` mit `/var/tmp` als das Dateiverzeichnis definiert, in dem die temporäre Datei angelegt wird.

Returnwert

Zeiger auf den Datenstrom für die erzeugte Datei	bei Erfolg.
Nullzeiger	bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `tmpfile()` schlägt fehl, wenn gilt:

`EINTR`	Ein Signal wurde während des Ablaufs der Funktion `tmpfile()` abgefangen.
`EMFILE`	Im aufrufenden Prozess sind `{OPEN_MAX}`-Datenströme geöffnet. Im aufrufenden Prozess sind `{FOPEN_MAX}`-Datenströme geöffnet.
`ENFILE`	Im System ist die maximal erlaubte Anzahl von Dateien geöffnet.
`ENOSPC`	Das Dateiverzeichnis oder das Dateisystem, das die neue Datei enthalten würde, kann nicht vergrößert werden.

Hinweis

Ob `tmpfile()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

Bei abnormalem Programmabbruch mit `abort()` bzw. `_exit(-1)` werden die temporären Dateien nicht gelöscht.

Siehe auch `fopen()`, `tmpnam()`, `unlink()`, `stdio.h`.

tmpnam - Basisnamen für temporäre Datei erzeugen

Syntax #include <stdio.h>

char *tmpnam(char *s)

Beschreibung

`tmpnam()` erzeugt eine Zeichenkette, die ein gültiger, eindeutiger Dateiname ist.

`tmpnam()` erzeugt jedes Mal, wenn sie vom selben Prozess aufgerufen wird, einen anderen Dateinamen (bis zu `{TMP_MAX}`-mal). Wenn die Funktion öfter als `{TMP_MAX}`-mal aufgerufen wird, werden vorher benutzte Namen wieder verwendet.

Die Implementierung verhält sich so, als ob `tmpnam()` von keiner Bibliotheksfunktion aufgerufen würde.

In `stdio.h` ist `P_tmpdir` mit `"/var/tmp"` als das Dateiverzeichnis definiert, in dem die temporäre Datei angelegt wird.

Returnwert Zeiger auf eine Zeichenkette
Bei erfolgreicher Beendigung.

Nullzeiger wenn `tmpnam()` öfter als `{TMP_MAX}`-mal aufgerufen wurde.

Wenn das Argument *s* der Nullzeiger ist, schreibt `tmpnam()` das Ergebnis in einen internen, statischen Bereich und gibt einen Zeiger auf diesen Bereich zurück. Nachfolgende Aufrufe der Funktion `tmpnam()` können denselben Bereich wieder verändern.

Wenn das Argument *s* nicht der Nullzeiger ist, wird angenommen, dass es auf einen Vektor vom Typ *char* der Mindestlänge `{L_tmpnam}` zeigt;
`tmpnam()` schreibt das Ergebnis in diesen Vektor und liefert das Argument als Returnwert.

Hinweis Wenn die Funktion `tmpnam()` öfter als `{TMP_MAX}`-mal in einem Prozess aufgerufen wird, werden vorher benutzte Namen wieder verwendet.

Der Benutzer ist dafür verantwortlich, die Datei zu löschen, auf die **s* zeigt, wenn diese nicht mehr gebraucht wird.

In der Zeit zwischen dem Erzeugen des Dateinamens und dem Erzeugen der Datei kann ein anderer Prozess den gleichen Dateinamen erzeugt haben. Daher kann der Einsatz von `tmpfile()` nützlicher sein.

Dies kann jedoch nicht eintreten, wenn der andere Prozess `tmpnam()` oder `mktemp()` verwendet und der Pfadname so gewählt wird, dass seine Duplizierung auf andere Weise unwahrscheinlich ist.

Dateien, die mit `tmpnam()` und entweder von `fopen()` oder `creat()` erstellt wurden, sind nur insofern temporär, weil sie sich in einem Dateiverzeichnis befinden, das für temporären Gebrauch bestimmt ist, und weil ihre Namen eindeutig sind.

Ob `tmpnam()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

Siehe auch `fopen()`, `open()`, `tempnam()`, `tmpfile()`, `unlink()`, `stdio.h`.

toascii - ganze Zahl in gültigen Wert umwandeln

Syntax

#include <ctype.h>

int toascii(int *i*);

Beschreibung

`toascii()` setzt die ersten 3 Bytes einer Variablen *i* durch Bit-UND-Verknüpfung (*i* & 0XFF) auf 0 und liefert den Wert des niedrigstwertigen Bytes zurück.

`toascii()` ist ein Synonym für `toebcdic()`. Auf EBCDIC-Rechnern liefert `toascii()` einen gültigen Wert aus dem EBCDIC-Zeichensatz. Ist Portabilität zu ASCII-Rechnern erforderlich, sollte `toascii()` verwendet werden.

i ist eine ganzzahlige Variable, deren niedrigstwertiges Byte geliefert werden soll.

Returnwert Wert des niedrigstwertigen Bytes der Variablen *i*
bei Erfolg.

Hinweis `toascii()` wandelt keine Werte aus anderen Zeichensätzen um (z.B. ASCII auf EBCDIC-Rechnern).

Siehe auch `isacii()`, `toebcdic()`, `ctype.h`.

toebcdic - ganze Zahl in gültigen Wert umwandeln *(BS2000)*

Syntax #include <ctype.h>

int toebcdic(int *i*);

Beschreibung

`toebcdic()` setzt die ersten 3 Bytes einer Variablen *i* durch Bit-UND-Verknüpfung (*i* & 0XFF) auf 0 und liefert den Wert des niedrigstwertigen Bytes zurück.

i ist eine ganzzahlige Variable, deren niedrigstwertiges Byte geliefert werden soll.

Returnwert Das niedrigstwertige Byte der Variablen *i*
bei Erfolg.

Hinweis `toebcdic()` ist sowohl als Makro als auch als Funktion realisiert.

`toebcdic()` wandelt keine Werte aus anderen Zeichensätzen (z.B. ASCII) um.

`toebcdic()` ist ein Synonym für `toascii()`. Ist Portabilität zu ASCII-Rechnern erforderlich, sollte `toascii()` statt `toebcdic()` verwendet werden.

Siehe auch `isascii()`, `toascii()`, `ctype.h`.

_tolower - Großbuchstaben in Kleinbuchstaben umwandeln

Syntax #include <ctype.h>

int _tolower(int *c*);

Beschreibung

`_tolower()` wandelt den Großbuchstaben *c* in den entsprechenden Kleinbuchstaben um.

c muss ein Großbuchstabe sein.

Returnwert Kleinbuchstabe zu *c*, wenn *c* ein Großbuchstabe ist.

Hinweis `_tolower()` ist nur als Makro realisiert.

Siehe auch `tolower()`, `isupper()`, `ctype.h`.

tolower - Zeichen in Kleinbuchstaben umwandeln

Syntax #include <ctype.h>

int tolower(int *c*);

Beschreibung

`tolower()` wandelt den Großbuchstaben *c* in den entsprechenden Kleinbuchstaben um.

Returnwert Kleinbuchstabe zu *c*, wenn *c* ein Großbuchstabe ist.

Siehe auch `strlower()`, `strupper()`, `toupper()`, `setlocale()`, `ctype.h`.

_toupper - Kleinbuchstaben in Großbuchstaben umwandeln

Syntax #include <ctype.h>

int _toupper(int *c*);

Beschreibung

`_toupper()` wandelt den Kleinbuchstaben *c* in den entsprechenden Großbuchstaben um.

c muss ein Kleinbuchstabe sein.

Returnwert Großbuchstabe zu *c*, wenn *c* ein Kleinbuchstabe ist.

Hinweis `_toupper()` ist nur als Makro realisiert.

Siehe auch `toupper()`, `islower()`, `ctype.h`.

toupper - Zeichen in Großbuchstaben umwandeln

Syntax #include <ctype.h>

int toupper(int *c*);

Beschreibung

`toupper()` wandelt den Kleinbuchstaben *c* in den entsprechenden Großbuchstaben um.

Returnwert Großbuchstabe zu *c*, wenn *c* ein Kleinbuchstabe ist.

Siehe auch `strupper()`, `strlower()`, `tolower()`, `setlocale()`, `ctype.h`.

towctrans - Langzeichen abbilden

Syntax #include <wctype.h>

wint_t towctrans(wint_t *wc*, wctrans_t *desc*);

Beschreibung

`towctrans()` transformiert das Langzeichen *wc* gemäß der Angabe *desc*. Der aktuelle Wert der Kategorie `LC_CTYPE` muss derselbe sein, der für den Aufruf von `towctrans()` gültig war, der den Wert *desc* zurückgab.

Die beiden folgenden Aufrufe von `towctrans()` wirken genauso, wie die dahinter in Kommentarzeichen angegebenen Aufrufe zur Umwandlung in Klein- bzw. Großbuchstaben:

```
towctrans(wc, wctrans("tolower"))          /* towlower(wc) */
towctrans(wc, wctrans("toupper"))          /* towupper(wc) */
```

Returnwert transformiertes Langzeichen
bei Erfolg.

Hinweis In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Siehe auch `tolower(), toupper(), towlower(), towupper(), wctrans()`

towlower - Langzeichen in Kleinbuchstaben umwandeln

Syntax #include <wchar.h>

wint_t towlower(wint_t *wc*);

Beschreibung

`towlower()` wandelt das Langzeichen *wc*, falls es ein Großbuchstabe ist, in den entsprechenden Kleinbuchstaben um.

Returnwert Kleinbuchstabe zu *wc*, wenn *wc* ein Großbuchstabe ist.

Hinweis *Einschränkung*
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Siehe auch `towupper()`, `setlocale()`, `wchar.h`.

towupper - Langzeichen in Großbuchstaben umwandeln

Syntax #include <wchar.h>

wint_t towupper(wint_t *wc*);

Beschreibung

`towupper()` wandelt das Langzeichen *wc*, falls es ein Kleinbuchstabe ist, in den entsprechenden Großbuchstaben um.

Returnwert Großbuchstabe zu *wc*, wenn *wc* ein Kleinbuchstabe ist.

Hinweis *Einschränkung*
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Siehe auch `towlower()`, `setlocale()`, `wchar.h`.

truncate - Datei auf angegebene Länge setzen

Name **truncate, truncate64**

Syntax #include <unistd.h>

int truncate (const char **path*, off_t *length*);
int truncate64 (const char **path*, off64_t *length*);

Beschreibung

Siehe `ftruncate()`.

`truncate()` kürzt die in *path* angegebene Datei auf *length* Bytes.

tsearch, tfind, tdelete, twalk - binäre Suchbäume bearbeiten

Syntax #include <search.h>

void *tsearch (const void **key*, void ***rootp*, int (**compar*) (const void *, const void *));
void *tfind (const void **key*, void * const **rootp*, int (**compar*) (const void *, const void *));
void *tdelete (const void **key*, void ***rootp*, int (**compar*) (const void *, const void *));
void twalk (const void **root*, void(**action*) (const void *, VISIT, int));

Beschreibung

`tsearch()`, `tfind()`, `tdelete()` und `twalk()` manipulieren binäre Suchbäume. Vergleiche werden durch eine vom Benutzer gelieferte Funktion *compar* ausgeführt. Diese Funktion wird mit zwei Argumenten aufgerufen, d.h. mit den Zeigern auf die Elemente, die verglichen werden. Sie gibt eine ganze Zahl zurück, die kleiner, gleich oder größer als 0 ist, je nachdem, ob das erste Argument kleiner, gleich oder größer als das zweite Argument ist. Die Vergleichsfunktion braucht nicht jedes Byte zu vergleichen, und daher können außer den Werten, die verglichen werden, auch willkürliche Daten in den Elementen enthalten sein.

`tsearch()` wird zum Aufbau des Baums und für den Zugriff auf den Baum verwendet. *key* ist ein Zeiger auf einen Wert, auf den zugegriffen bzw. der gespeichert werden soll. Wenn der Baum einen Wert aufweist, der gleich **key* (der Wert, auf den der Schlüssel zeigt) ist, wird ein Zeiger auf diesen gefundenen Wert zurückgegeben. Andernfalls wird **key* eingefügt und ein auf diesen *key* weisender Zeiger zurückgegeben. Es werden nur Zeiger kopiert, und daher müssen die Daten von der aufrufende Routine gespeichert werden. *rootp* zeigt auf eine Variable, die auf die Wurzel des Baums zeigt. Ein `NULL`-Wert für die Variable, auf die *rootp* zeigt, gibt einen leeren Baum an; in diesem Fall wird die Variable so gesetzt, dass sie auf den Wert zeigt, der sich an der Wurzel des neuen Baums befindet.

Wie `tsearch()` sucht auch `tfind()` nach einem Wert im Baum und gibt einen Zeiger auf diesen Wert zurück, falls dieser gefunden wird. Wird der Wert nicht gefunden, gibt `tfind()` einen Nullzeiger zurück. Die Argumente für `tfind()` sind dieselben wie für `tsearch()`.

Mit `tdelete()` wird ein Knoten in einem binären Suchbaum gelöscht. Die Argumente sind dieselben wie für `tsearch()`. Die Variable, auf die *rootp* zeigt, ändert sich, wenn der gelöschte Knoten die Wurzel des Baums war. `tdelete()` gibt einen Zeiger auf den Vaterknoten des gelöschten Knotens zurück oder einen Nullzeiger, wenn der Knoten nicht gefunden wurde.

`twalk()` durchläuft einen binären Suchbaum. *root* ist die Wurzel des Baums, der durchlaufen werden soll. Jeder Knotenpunkt im Baum kann als Wurzel für ein Durchlaufen des Baums unterhalb dieses Knotens verwendet werden. *action* ist der Name einer Funktion, die an jedem Knoten aufgerufen werden soll. Diese Funktion wird mit drei Argumenten aufgerufen. Das erste Argument ist die Adresse des besuchten Knotens. Die Struktur, auf die

dieses Argument zeigt, ist nicht spezifiziert und darf nicht verändert werden. Der Wert vom Typ 'Zeiger-auf-Knoten' kann jedoch in den Typ 'Zeiger-auf-Zeiger-auf-Element' konvertiert werden, um auf die in dem Knoten gespeicherten Elemente zugreifen zu können.

Das zweite Argument ist ein Wert des Aufzählungstyps *typedef enum* { *preorder, postorder, endorder, leaf* } *VISIT*; (definiert in der Include-Datei `search.h`), abhängig davon, ob es sich um den ersten, zweiten oder dritten Besuch des Knotens handelt, bei einem Durchlauf des Baums in die Tiefe, von links nach rechts, oder ob der Knoten ein Blatt ist. Das dritte Argument stellt die Stufe des Knotens im Baum dar, wobei die Wurzel die Stufe Null ist.

Returnwert **key* `tsearch()` und `tfind()`: bei Erfolg.
`tsearch()`: Zeiger auf die eingefügte Position.

Nullzeiger `tsearch()`: wenn nicht ausreichend Speicher zur Erstellung eines neuen Knotens zur Verfügung steht.
`tsearch()`, `tfind()` und `tdelete()`: wenn *rootp* zu Beginn `NULL` ist.
`tfind()`: wenn **key* nicht gefunden wurde.

Zeiger auf den Vaterknoten des gelöschten Knotens `tdelete()`
bei Erfolg.

Hinweis *root* für `twalk()` ist um eine Stufe der indirekten Adressierung niedriger als *rootp* für `tsearch()` und `tdelete()`.

Es gibt zwei Nomenklaturen für die Reihenfolge, in der die Knoten eines Baums durchlaufen werden. `tsearch()` verwendet die Begriffe "preorder", "postorder" und "endorder", um auszudrücken, dass ein Knoten vor seinen Söhnen oder nach dem linken und vor dem rechten Kind oder nach seinen Söhnen besucht wird. Die andere Nomenklatur verwendet "preorder", "inorder" und "postorder", um diese Reihenfolgen zu bezeichnen, wobei "postorder" eine andere Bedeutung hat.

Hinweis Wenn die aufrufende Funktion den Zeiger auf die Wurzel ändert, werden die Ergebnisse unvorhersagbar.

Siehe auch `bsearch()`, `hsearch()`, `lsearch()`, `search.h`.

ttyname - Pfadnamen eines Terminals ermitteln

Syntax #include <unistd.h>

char *ttyname(int *fildes*);

Beschreibung

`ttyname()` liefert einen Zeiger auf eine Zeichenkette. Diese enthält den mit dem Nullbyte abgeschlossenen Pfadnamen des Terminals, das dem Dateideskriptor *fildes* zugeordnet ist. Der Returnwert zeigt auf einen statischen Bereich, dessen Inhalt bei jedem Aufruf überschrieben wird.

Das steuernde Terminal kann folgende Namen haben:

`/dev/term/0000, ..., /dev/term/4096` (für Blockterminals)

`/dev/pts/0, ..., /dev/pts/4096` (bei `rlogin`-Zugang)

Returnwert Zeiger auf eine Zeichenkette bei Erfolg.

Nullzeiger bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `ttyname()` schlägt fehl, wenn gilt:

`EBADF` *fildes* ist kein gültiger Dateideskriptor.

`ENOTTY` *fildes* verweist nicht auf ein Terminal.

Hinweis `ttyname()` wird nur für POSIX-Dateien ausgeführt.

`ttyname()` ist nicht threadsicher. Verwenden Sie bei Bedarf die reentrante Funktion `ttyname_r()`.

Siehe auch `isatty()`, `ttyname_r()`, `unistd.h`.

ttyname_r - Pfadnamen eines Terminals threadsicher ermitteln

Syntax #include <unistd.h>

int ttyname_r(int *fildes*, char * *name*, size_t *namesize*);

Beschreibung

Die Funktion `ttyname_r()` speichert den mit dem Nullbyte abgeschlossenen Pfadnamen des Terminals, das dem Dateideskriptor *fildes* zugeordnet ist, im Datenbereich, auf den *name* zeigt, ab. Der Datenbereich ist *namesize* Zeichen lang und sollte genug Speicher für den Namen und das abschließende Nullbyte bereitstellen. Die maximale Länge des Terminalnamens ist `{TTY_NAME_MAX}`.

Returnwert 0 bei Erfolg.

Fehlernummer sonst.

Fehler `ttyname_r()` schlägt fehl, wenn gilt:

`EBADF`	Das Argument *fildes* ist kein gültiger Dateideskriptor.
`ENOTTY`	Das Argument *fildes* verweist nicht auf ein Terminal..
`ERANGE`	Der Wert von *namesize* ist kleiner als die Länge der zurückzugebenden Zeichenkette einschließlich des abschließenden Nullbyte.

Siehe auch `ttyname()`, `isatty()`, `unistd.h`.

ttyslot - Eintrag des aktuellen Benutzers in der utmp-Datei finden

Syntax

```
#include <stdlib.h>

int ttyslot (void);
```

Beschreibung

`ttyslot()` gibt für den aktuellen Benutzer den Index seines Eintrags in der Datei `/var/adm/utmp` zurück.
Der Eintrag des aktuellen Benutzers ist ein Eintrag, für den das `utline`-Strukturelement mit dem Namen eines Terminals in `/dev` übereinstimmt, das mit der Standardeingabe, der Standardausgabe oder der Fehlerausgabe (0, 1 oder 2) verbunden ist.

Der zurückgegebene Index ist eine ganze Zahl, die die Satznummer des Eintrags in der Datei `/var/adm/utmp` repräsentiert. Für den ersten Satz wird der Index 0 zurückgegeben.

`ttyslot()` ist nicht threadsicher.

Returnwert

Index des Eintrags	bei Erfolg.
-1	wenn bei der Suche nach dem Terminalnamen ein Fehler auftrat, oder wenn keiner der Dateideskriptoren 0, 1 oder 2 einem Terminal zugeordnet wurde.

Hinweis `ttyslot()` wird in der nächsten Version des X/Open Standards gestrichen.

Siehe auch `endutxent()`, `ttyname()`, `stdlib.h`.

twalk - Binärbaum durchlaufen

Syntax #include <search.h>

void twalk(const void **root*, void (**action*) (const void *, VISIT, int *));

Beschreibung

Siehe `tsearch()`.

tzname - Feldvariable für Zeitzonen-Zeichenketten

Syntax #include <time.h>

extern char *tzname[2];

Beschreibung

Die externe Variable `tzname` enthält Zeitzonennamen. `tzname` ist standardmäßig wie folgt gesetzt:

```
char *tzname[2] = { "GMT", "" };
```

Siehe auch `altzone`, `asctime()`, `ctime()`, `daylight`, `gmtime()`, `localtime()`, `timezone`, `tzset()`.

tzset - Information für Zeitzonenumwandlung setzen

Syntax

```
#include <time.h>

void tzset(void);
```

Beschreibung

`tzset()` benutzt den Inhalt der Umgebungsvariablen `TZ`, um die Werte unterschiedlicher externer Variablen zu überschreiben. Die Funktion `tzset()` wird von `asctime()` oder aber auch vom Benutzer aufgerufen.

`tzset()` überprüft den Inhalt der Umgebungsvariablen und weist die verschiedenen Felder den entsprechenden Variablen zu. Zum Beispiel lautet der vollständige Eintrag für New Jersey 1986:

`EST5EDT4,116/2:00:00,298/2:00:00` oder einfach nur `EST5EDT`

Ein Beispiel für die Südhalbkugel, zum Beispiel Cook Islands, könnte sein:

`KDT9:30KST10:00,63/5:00,302/20:00`

In der langen Version des New Jersey-Beispiels von `TZ` ist `tzname[0]` EST; `timezone` wird auf 5*60*60 gesetzt; `tzname[1]` ist EDT; `altzone` wird auf 4*60*60 gesetzt; die Sommerzeit beginnt am 117. Tag um 2 Uhr nachts und endet am 299. Tag um 2 Uhr nachts (es wird der Julianische Kalender benutzt). `daylight` wird auf einen positiven Wert gesetzt. Start- und Endzeit sind relativ zur Sommerzeit. Wenn Start- und Enddatum der Sommerzeit nicht geliefert werden, werden die für die Vereinigten Staaten in diesem Jahr gültigen Tage benutzt, und die Zeit wird 2 Uhr nachts sein. Wenn nur die Zeit nicht verfügbar ist, wird diese auf 2 Uhr nachts gesetzt.
Die Auswirkungen von `tzset()` sind so, dass die Werte der externen Variablen `timezone`, `altzone`, `daylight` und `tzname` geändert werden. `ctime()`, `localtime()`, `mktime()` und `strftime()` werden ebenso diese externen Variablen aktualisieren, als hätten sie `tzset()` zu der Zeit aufgerufen, die vom `time_t`- oder dem von ihnen konvertierten `struct-tm`-Wert spezifiziert wird.

Die Datei `/usr/lib/locale/` *language*`/LC_TIME` enthält umgebungsspezifische Datums- und Zeitinformationen.

`tzset()` setzt die externe Variable `daylight` auf 0, wenn für die angegebene Zeitzone keine Sommerzeit-Konvertierungen vorgenommen werden sollen, sonst auf einen Wert ungleich 0. Die externe Variable `timezone` wird auf die Differenz in Sekunden zwischen der koordinierten Universal Time (UTC) und der lokalen Standardzeit gesetzt.

Hinweis

Falls keine `TZ`-Variable vorhanden ist, werden die für MEZ gültigen Werte eingesetzt.

Siehe auch

`altzone`, `asctime()`, `ctime()`, `daylight`, `environ`, `gmtime()`, `localtime()`, `mktime()`, `strftime()`, `timezone`, `tzname()`.

ualarm - Intervall Timer setzen

Syntax #include <unistd.h>

useconds_t ualarm(useconds_t *useconds*, useconds_t *interval*)

Beschreibung

`ualarm()` sendet das Signal SIGALRM nach *useconds* Mikrosekunden an den aufrufenden Prozess. Sofern es nicht ignoriert oder abgefangen wird, beendet das Signal den Prozess.

Wenn das Argument *interval* ungleich null ist, wird das Signal SIGALRM alle *interval* Mikrosekunden nach Ablauf des Zeitgebers an den Prozess gesendet (zum Beispiel nachdem *useconds* Mikrosekunden verstrichen sind).

Auf Grund von Verzögerungen im Scheduling kann die Wiederaufnahme der Ausführung nach dem Abfangen des Signals um einige Zeit verschoben werden. Die längste Verzögerungszeit, die angegeben werden kann, beträgt 2.147.483.647 Mikrosekunden.

Returnwert Der Return-Wert ist die Zeit, die bis zur Ausgabe des Alarmsignals noch verbleibt.

Hinweis `ualarm()` ist eine vereinfachte Schnittstelle für `setitimer()`.

Siehe auch `alarm()`, `setitimer()`, `sleep ()`, `unistd.h`.

ulimit - Prozessgrenzen ermitteln oder setzen

Syntax #include <ulimit.h>

long int ulimit (int *cmd*, ...);

Beschreibung

`ulimit()` ermöglicht die Steuerung der Prozessgrenzen. Die möglichen Werte für *cmd*, die in `ulimit.h` definiert sind, beinhalten:

`UL_GETFSIZE` Liefert die Grenze für Dateigrößen des Prozesses. Die Grenze wird in 512-Byte-Blöcken angegeben und an Kindprozesse vererbt. Dateien jeder Größe können gelesen werden.

`UL_SETFSIZE` Setzt die Grenze für die Dateigröße bei Ausgabeoperationen des Prozesses auf den Wert des zweiten Arguments, das als `long int` interpretiert wird. Jeder Prozess kann seine eigene Grenze heruntersetzen, aber nur ein Prozess mit Sonderrechten darf diese Grenze erhöhen. Das Ergebnis ist die neue Grenze für die Dateigröße.

Returnwert Wert der geforderten Grenze bei Erfolg.

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

Fehler `ulimit()` schlägt fehl und die Grenze wird nicht verändert, wenn gilt:

`EINVAL` Das Argument *cmd* ist ungültig.

`EPERM` Ein Prozess ohne Sonderrechte versucht, die Grenze für die Dateigröße heraufzusetzen.

Hinweis Da bei Erfolg alle Ergebnisse erlaubt sind, sollte eine Anwendung, die Fehlersituationen überprüfen will, `errno` vor dem Aufruf von `ulimit()` gleich 0 setzen. Wenn das Ergebnis nach der Rückkehr gleich -1 und `errno` gesetzt ist, dann ist ein Fehler aufgetreten.

Siehe auch `write()`, `ulimit.h`.

umask - Schutzbitmaske abfragen und setzen

Syntax

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

Optional
#include <sys/types.h>  ❑

mode_t umask (mode_t cmask);
```

Beschreibung

`umask()` setzt für den Dateimodus die Schutzbitmaske des Prozesses gleich *cmask* und gibt den vorherigen Wert der Maske zurück. Nur die Schutzbits von *cmask* (siehe auch `sys/stat.h`) werden verwendet; andere Bits werden ignoriert.

Die Schutzbitmaske des Prozesses wird von den Funktionen `open()`, `creat()`, `mkdir()` und `mkfifo()` verwendet, um Zugriffsrechte in *mode* zu entfernen. Bitpositionen, die in *cmask* gesetzt sind, werden bei den Zugriffsrechten der erzeugten Datei entfernt.

Durch einen erneuten Aufruf von `umask()` mit dem Returnwert des ersten Aufrufs als Argument kann der Zustand, den die Maske vor dem ersten Aufruf hatte, einschließlich aller anderen Bits, wieder hergestellt werden.

Returnwert Wenn die Benutzernummer 0 ist, ist der voreingestellte Wert 022 (oktal), sonst 066. vorheriger Wert der Schutzbitmaske
bei Erfolg. Andere Bits werden ignoriert. Ein erneuter Aufruf von `umask()` mit dem Ergebnis des vorangegangenen Aufrufs als *cmask* setzt die Schutzbitmaske auf den Zustand vor dem ersten Aufruf zurück.

Hinweis `umask()` wird nur für POSIX-Dateien ausgeführt.

Siehe auch `creat()`, `mkdir()`, `mkfifo()`, `open()`, `sys/stat.h`, `sys/types.h`.

umount - Dateisystem aushängen *(Erweiterung)*

Syntax

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

int umount(const char *path);
```

Beschreibung

Mit `umount()` wird ein zuvor mit `mount()` eingehängtes Dateisystem ausgehängt, das unter dem Dateiverzeichnis liegt, auf das *path* zeigt (Einhängepunkt). *path* kann auf eine blockorienterte Gerätedatei oder ein Dateiverzeichnis zeigen. Nach dem Aushängen des Dateisystems wird das Dateiverzeichnis, in dem das Dateisystem eingehängt war, wieder normal interpretiert.

Returnwert

0	nach erfolgreicher Beendigung.
-1	bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `umount()` ist erfolglos, wenn gilt:

`EBUSY`	Eine Datei in *path* ist in Benutzung.
`EFAULT`	*path* zeigt auf eine ungültige Adresse.
`EINVAL`	*path* ist nicht vorhanden, oder *path* ist nicht eingehängt.
`ELOOP`	Zu viele symbolische Verweise wurden aufgerufen, um den Pfad zu übersetzen, auf den durch *path* verwiesen wurde.
`ENAMETOOLONG`	*path* ist länger als `{PATH_MAX}`, oder die Länge einer *path*-Komponente überschreitet `{NAME_MAX}`.
`ENOTBLK`	*path* ist keine blockorientierte Gerätedatei.
`EPERM`	Die effektive Benutzernummer ist nicht die eines Prozesses mit Sonderrechten.
`EREMOTE`	*path* zeigt auf einen fernen Pfadnamen.

Hinweis

`umount()` darf nur unter der effektiven Benutzernummer eines Prozesses mit Sonderrechten aufgerufen werden.

`umount()` wird nur für POSIX-Dateien ausgeführt.

Siehe auch `mount()`, `sys/mount.h`.

uname - Basisdaten über das aktuelle Betriebssystem ermitteln

Syntax #include <sys/utsname.h>

int uname(struct utsname **name*);

Beschreibung

`uname()` ermittelt Basisdaten über das aktuelle Betriebssystem und speichert sie in der Struktur ab, auf die *name* zeigt.

`uname()` verwendet die Struktur `utsname`, die in `sys/utsname.h` definiert ist. Strukturkomponenten sind die `char`-Vektoren `sysname`, `nodename`, `release`, `version` und `machine`. Im Vektor `sysname` wird der Name des aktuellen Betriebssystems eingetragen. Analog dazu enthält `nodename` den Namen, unter dem das Betriebssystem in einem Kommunikationsnetz bekannt ist. Die Vektoren `release` und `version` enthalten Release-Nummer und Freigabedatum des Betriebssystems, der Vektor `machine` enthält einen Namen, der die Hardware kennzeichnet, auf der das Betriebssystem abläuft.

Returnwert

nichtnegativer Wert	bei Erfolg.
-1	bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `uname()` schlägt fehl, wenn gilt:

Erweiterung	
`EFAULT`	*name* ist eine ungültige Adresse. ❑

Hinweis Die Aufnahme der Komponente `nodename` in diese Struktur besagt nicht, dass dies genügend Information ist, um Kommunikationsnetze anzusprechen.

Siehe auch `sys/utsname.h`.

ungetc - Byte in Eingabestrom zurückstellen

Syntax

```
#include <stdio.h>

int ungetc(int c, FILE *stream);
```

Beschreibung

`ungetc()` wandelt das vorher gelesene Byte *c* in den Typ `unsigned char` um und stellt das entsprechende Byte in den Datenstrom zurück, auf den *stream* zeigt. Die zurückgestellten Bytes werden durch nachfolgende Leseoperationen aus diesem Datenstrom in umgekehrter Reihenfolge zurückgegeben. Wenn zwischendurch eine Funktion zur Positionierung (`fseek()`, `fsetpos()` oder `rewind()`) für denselben Datenstrom erfolgreich aufgerufen wird, werden die zurückgestellten Bytes aus dem Datenstrom gelöscht. Der externe Speicher, der dem Datenstrom zugeordnet ist, bleibt unverändert.

BS2000
Der Aufruf einer der folgenden Funktionen hebt die Effekte des `ungetc`-Aufrufs (z.B. Rückwärtspositionierung) auf: `fseek()`, `fsetpos()`, `lseek()`, `rewind()`, `fflush()`. ❑

Das Zurückstellen genau eines Bytes ist garantiert. Wenn `ungetc()` zu oft für denselben Datenstrom aufgerufen wird, ohne dass zwischendurch eine Leseoperation oder ein Positionieren stattfindet, kann das Zurückstellen fehlschlagen. Im C-Laufzeitsystem können maximal `{BUFSIZE}` Zeichen zurückgestellt werden (siehe `stdio.h`).

Wenn der Wert von *c* gleich der Konstanten `EOF` ist, schlägt das Zurückstellen fehl und der Eingabestrom bleibt unverändert.

Ein erfolgreicher `ungetc`-Aufruf löscht das Dateiendekennzeichen für diesen Datenstrom. Der Wert des Lese-/Schreibzeigers für den Datenstrom ist nach einem Lesen oder Verwerfen aller zurückgestellten Bytes derselbe wie vor dem Zurückstellen der Bytes. Der Lese-/Schreibzeiger wird durch jeden erfolgreichen `ungetc`-Aufruf erniedrigt. Wenn sein Wert vor einem Aufruf gleich 0 ist, ist sein Wert nach dem Aufruf unbestimmt.

Returnwert

zurückgestelltes Byte	bei Erfolg.
`EOF`	wenn *c* gleich `EOF` ist, oder bei Fehler.

Hinweis

Es muss immer wenigstens ein Byte vor dem ersten `ungetc`-Aufruf aus der Datei gelesen worden sein.

Ob `ungetc()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

BS2000

Wenn beim Zugriff auf BS2000-Dateien an Stelle des zuvor eingelesenen Bytes ein anderes Byte in den Puffer zurückgestellt wurde, ist das Verhalten je nach KR- oder ANSI-Funktionalität unterschiedlich:

– Bei KR-Funktionalität (nur bei C/C++ Versionen kleiner V3 vorhanden) werden beim Schreiben des Pufferinhalts in die externe Datei Originaldaten verändert.

– Bei ANSI-Funktionalität werden beim Schreiben des Pufferinhalts in die externe Datei Originaldaten nicht verändert, d.h., es werden stets die Originaldaten vor dem `ungetc`-Aufruf in die externe Datei geschrieben. ❑

Siehe auch `fseek()`, `getc()`, `fsetpos()`, `read()`, `rewind()`, `setbuf()`, `stdio.h`.

ungetwc - Langzeichen in Eingabestrom zurückstellen

Syntax

```
#include <wchar.h>

Optional
#include <stdio.h> ❑

wint_t ungetwc(wint_t wc, FILE *stream);
```

Beschreibung

`ungetwc()` stellt das Zeichen, das mit dem Langzeichen *wc* korrespondiert, in den Eingabestrom zurück, auf den *stream* zeigt. Die zurückgestellten Zeichen werden durch nachfolgende Leseoperationen aus diesem Datenstrom in umgekehrter Reihenfolge zurückgegeben. Wenn zwischendurch eine Funktion zur Positionierung (`fseek()`, `fsetpos()` oder `rewind()`) für denselben Datenstrom erfolgreich aufgerufen wird, werden die zurückgestellten Zeichen aus dem Datenstrom gelöscht. Der externe Speicher, der dem Datenstrom zugeordnet ist, bleibt unverändert.

Das Zurückstellen eines Zeichens ist garantiert. Wenn `ungetwc()` zu oft für denselben Datenstrom aufgerufen wird, ohne dass zwischendurch eine Lese- oder Positionierungsoperation ausgeführt wird, kann das Zurückstellen fehlschlagen.

Wenn der Wert von *wc* gleich der Konstanten `WEOF` ist, schlägt die Operation fehl und der Eingabestrom bleibt unverändert.

Ein erfolgreicher `ungetwc`-Aufruf löscht das Dateiendekennzeichen für diesen Datenstrom. Der Wert des Lese-/Schreibzeigers für den Datenstrom ist nach einem Lesen oder Verwerfen aller zurückgestellten Zeichen derselbe wie vor dem Zurückstellen der Zeichen. Der Lese-/Schreibzeiger wird durch jeden erfolgreichen `ungetwc`-Aufruf erniedrigt. Wenn sein Wert vor einem Aufruf gleich 0 ist, ist sein Wert nach dem Aufruf unbestimmt.

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Returnwert

zurückgestelltes Langzeichen	bei erfolgreicher Beendigung.
`WEOF`	wenn das Langzeichen nicht zurückgestellt werden kann. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `ungetwc()` schlägt fehl, wenn gilt:

Erweiterung

`EINVAL`	Es wurde versucht, auf eine BS2000-Datei zuzugreifen.

Siehe auch `fseek()`, `fsetpos()`, `read()`, `rewind()`, `setbuf()`, `stdio.h`, `wchar.h`.

unlink - Verweis löschen

Syntax #include <unistd.h>

int unlink(const char *_path_);

Beschreibung

`unlink()` löscht den Dateiverzeichnis-Eintrag, der durch den Pfadnamen angegeben wird, auf den *path* zeigt, und vermindert den Verweiszähler der Datei, auf die sich der Dateiverzeichnis-Eintrag bezieht. Sobald alle Verweise auf eine Datei entfernt worden sind und kein Prozess die Datei geöffnet hat, wird der von der Datei belegte Speicher freigegeben, und die Datei ist fortan nicht mehr zugreifbar. Falls einer oder mehrere Prozesse die Datei während der Entfernung der letzten Verbindung geöffnet haben, wird der von der Datei belegte Speicher nicht freigegeben, bis alle Verweise auf die Datei geschlossen wurden. Wenn *path* ein symbolischer Verweis ist, wird er entfernt.

path sollte kein Verzeichnis benennen, sofern der Prozess keine entsprechenden Privilegien besitzt. Anwendungen sollten zur Entfernung von Verzeichnissen `rmdir()` benutzen.

Nach erfolgreicher Durchführung markiert `unlink()` die Strukturkomponenten `st_ctime` und `st_mtime` des übergeordneten Verzeichnisses zum Aktualisieren. Ebenso wird die Strukturkomponente `st_ctime` der Datei zum Aktualisieren markiert, wenn der Verweiszähler der Datei ungleich null ist.

BS2000
`unlink()` wird aus Kompatibilitätsgründen weiter unterstützt und bewirkt das Gleiche wie `remove()`, nämlich das Löschen der Datei (siehe `remove()`). ❑

Returnwert

0	bei Erfolg.
-1	bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen. Die unter *path* genannte Datei wird nicht verändert.

Fehler `unlink()` schlägt fehl, wenn gilt:

`EACCES`	Für eine Komponente des Pfadnamenanfangs existiert kein Durchsuchrecht oder das Schreibrecht wird für das Dateiverzeichnis verweigert, das den zu löschenden Dateiverzeichniseintrag enthält.
`EBUSY`	Der Eintrag, der entfernt werden soll, ist der Einhängepunkt für ein eingehängtes Dateisystem.

Erweiterung

`EFAULT`	*path* weist über den zugewiesenen Adressraum des Prozesses hinaus.
`EINTR`	Ein Signal wurde während des `unlink()`-Systemaufrufs aufgefangen.

`ELOOP` Bei der Übersetzung von *path* wurden zu viele symbolische Verbindungen angetroffen. ❑

`ENAMETOOLONG`
Die Länge von *path* überschreitet `{PATH_MAX}`, oder die Länge einer Komponente von *path* ist größer als `{NAME_MAX}`.

`ENOENT` Die angegebene Datei ist nicht vorhanden oder ist eine leere Zeichenkette. Der Benutzer ist kein Systemverwalter.

`ENOTDIR` Eine Komponente von *path* ist kein Dateiverzeichnis.

`EPERM` Die durch *path* angegebene Datei ist ein Dateiverzeichnis, und der aufrufende Prozess hat keine Sonderrechte.

`EROFS` Der zu entfernende Dateiverzeichnis-Eintrag ist Teil eines schreibgeschützten Dateisystems.

Hinweis `rmdir()` wird zum Löschen eines Dateiverzeichnises verwendet.

Ob `unlink()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

Siehe auch `close()`, `link()`, `remove()`, `rmdir()`, `unistd.h`.

unlockpt - Lock von Master/Slave Pseudoterminalpaar aufheben

Syntax #include <stdlib.h>

int unlockpt (int *fildes*);

Beschreibung

Die Funktion `unlockpt()` entsperrt das Slave-Pseudoterminal, das dem in *fildes* angegebenen Master-Pseudoterminal zugeordnet ist.

Portable Anwendungen müssen `unlockpt()` aufrufen, bevor sie die Slave-Seite eines Pseudoterminals öffnen.

Returnwert

0	bei Erfolg.
-1	sonst. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `unlockpt()` schlägt fehl, wenn gilt:

`EBADF`	Das Argument *fildes* ist kein zum Schreiben geöffneter Dateideskriptor.
`EINVAL`	Dem Argument *fildes* ist kein Master-Pseudoterminal zugeordnet.

Siehe auch `grantpt()`, `open()`, `ptsname()`, `stdlib.h`.

usleep - Prozess für festgesetzte Zeitspanne anhalten

Syntax

```
#include <unistd.h>

int usleep(useconds_t useconds);
```

Beschreibung

Hält den aktuellen Prozess für *useconds* Mikrosekunden an. Die tatsächliche Zeit, die der Prozess angehalten wird, kann auf Grund anderer Aktivitäten im System oder auf Grund der Zeit, die für die Verarbeitung des Aufrufs benötigt wird, länger als *useconds* Mikrosekunden sein.

Es muss gelten *useconds* < 1 000 000. Falls gilt: *useconds* = 0, hat `usleep()` keine Wirkung.

Die Routine wird implementiert, indem der Intervallzeitgeber des Prozesses gesetzt und dann gewartet wird, bis er abgelaufen ist. Der vorherige Status dieses Zeitgebers wird gesichert und wiederhergestellt. Wenn die Wartezeit (Sleep Time) die Dauer bis zum Ablauf des vorherigen Zeitgebers überschreitet, wird der Prozess nur so lange angehalten, bis das Signal aufgetreten wäre, und das Signal wird kurz vor Ablauf dieser Wartezeit gesendet.

Werden Threads verwendet, so wirkt sich die Funktion auf den Prozess oder auf einen Thread wie folgt aus: `usleep()` bewirkt, dass der aktuelle Thread suspendiert wird bis ein angegebenes Zeitintervall abgelaufen ist oder ein Signal an den Thread zugestellt wurde.

Returnwert

0	bei Erfolg.
-1	sonst.

Hinweis

`usleep()` wird aus historischen Gründen unterstützt. Statt dieser Funktion sollte `setitimer()` verwendet werden.

Siehe auch

`alarm()`, `getitimer()`, `sigaction()`, `sleep()`, `unistd.h`.

utime - Dateizugriffs- und -änderungszeitpunkte setzen

Syntax

```
#include <utime.h>

Optional
#include <sys/types.h> ❑

int utime(const char *path, const struct utimbuf *times);
```

Beschreibung

`utime()` setzt die Zugriffs- und Änderungszeit der Datei, auf die *path* zeigt.

Wenn *times* ein Nullzeiger ist, werden Zugriffs- und Änderungszeit der Datei auf die aktuelle Uhrzeit gesetzt. Die effektive Benutzernummer des Prozesses muss mit der des Eigentümers der Datei übereinstimmen, oder der Prozess muss für die Datei Schreibrecht oder Sonderrechte haben, damit `utime()` auf diese Weise genutzt werden kann.

Wenn *times* kein Nullzeiger ist, dann wird *times* als Zeiger auf eine Struktur `utimbuf` (definiert in `utime.h`) interpretiert, und Zugriffs- und Änderungszeit wird gemäß den Werten in dieser Struktur gesetzt. Nur ein Prozess, dessen effektive Benutzernummer mit der des Eigentümers der Datei übereinstimmt, oder ein Prozess mit besonderen Rechten kann `utime()` auf diese Art nutzen.

Die Zeiten in der Struktur `utimbuf` werden in Sekunden ab 00:00:00 GMT 1. Januar 1970 gemessen (siehe `utime.h`).

Bei erfolgreicher Beendigung versieht `utime() st_ctime` mit einer Änderungsmarke (siehe `sys/stat.h`).

Returnwert

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

Fehler `utime()` schlägt fehl, wenn gilt:

`EACCES`	Eine Komponente des Pfades darf nicht durchsucht werden, oder *times* ist ein Nullzeiger und die effektive Benutzernummer ist nicht diejenige des Systemverwalters und nicht diejenige des Eigentümers der Datei, und der Schreibzugriff wird verweigert.
Erweiterung `EFAULT`	*times* ist ungleich null und weist über den zugewiesenen Adressraum des Prozesses hinaus, oder *path* weist über den zugewiesenen Adressraum des Prozesses hinaus.
`EINTR`	Ein Signal wurde während des Systemaufrufs `utime()` abgefangen.
`EINVAL`	Es wurde versucht, auf eine BS2000-Datei zuzugreifen.

ELOOP	Während der Übersetzung von *path* traten zu viele symbolische Verweise auf. ❑
ENAMETOOLONG	Die Länge von *path* überschreitet {PATH_MAX} oder die Länge einer Komponente von *path* überschreitet{NAME_MAX}.
ENOENT	Die angegebene Datei ist nicht vorhanden.
ENOTDIR	Eine Komponente des Pfades ist kein Dateiverzeichnis.
EPERM	Die effektive Benutzernummer ist nicht diejenige des Systemverwalters und nicht diejenige des Eigentümers der Datei, und *times* ist nicht gleich null.
EROFS	Das Dateisystem, das die Datei enthält, ist schreibgeschützt eingehängt.

Hinweis `utime()` wird nur für POSIX-Dateien ausgeführt.

Siehe auch `sys/types.h`, `utime.h`.

utimes - Dateizugriffs- und -änderungszeitpunkt setzen

Syntax #include <sys/time.h>

int utimes(const char **path*, const struct timeval *times*[2]);

Beschreibung

`utimes()` setzt die Zugriffs- und Änderungszeiten der Datei, auf die *path* zeigt, auf die in *times* angegebenen Werte.
Die Funktion erlaubt mikrosekundengenaue Zeitangaben.

Das Argument *times* ist ein Array, das aus zwei Strukturen des Typs `timeval` besteht. Die Zugriffszeit wird auf den Wert des ersten Elements und die Änderungszeit auf den Wert des zweiten Elements gesetzt. Die Zeiten in der `timeval`-Struktur werden in Sekunden und Mikrosekunden ab 00:00:00 GMT 1. Januar 1970 gemessen (siehe `utime.h`).

Wenn *times* der Nullzeiger ist, werden Zugriffs- und Änderungszeit auf die aktuelle Zeit gesetzt. Wenn `utimes()` auf diese Weise verwendet werden soll, muss der Prozess der Eigentümer der Datei sein, über Schreibberechtigung für die Datei verfügen oder ein Prozess mit besonderen Rechten sein.

Bei erfolgreicher Beendigung versieht `utimes()` das Feld `st_ctime` mit einer Änderungsmarke (siehe `sys/stat.h`).

Returnwert

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

Fehler `utimes()` schlägt fehl, wenn gilt:

`EACCES`	Eine Komponente des Pfades darf nicht durchsucht werden, oder *times* ist ein Nullzeiger und die effektive Benutzernummer ist nicht diejenige des Systemverwalters und nicht diejenige des Eigentümers der Datei, und der Schreibzugriff wird verweigert.
Erweiterung `EFAULT`	*times* ist ungleich null und weist über den zugewiesenen Adressraum des Prozesses hinaus, oder *path* weist über den zugewiesenen Adressraum des Prozesses hinaus.
`EINTR`	Ein Signal wurde während des Systemaufrufs `utime()` abgefangen.
`EINVAL`	Es wurde versucht, auf eine BS2000-Datei zuzugreifen.
`ELOOP`	Während der Übersetzung von *path* traten zu viele symbolische Verweise auf. ❑

ENAMETOOLONG
: Die Länge von *path* überschreitet {PATH_MAX} oder die Länge einer Komponente von *path* überschreitet{NAME_MAX}.

ENOENT
: Die angegebene Datei ist nicht vorhanden.

ENOTDIR
: Eine Komponente des Pfades ist kein Dateiverzeichnis.

EPERM
: Die effektive Benutzernummer ist nicht diejenige des Systemverwalters und nicht diejenige des Eigentümers der Datei, und *times* ist nicht gleich null.

EROFS
: Das Dateisystem, das die Datei enthält, ist schreibgeschützt eingehängt.

Siehe auch sys/time.h.

va_arg - variable Argumentliste abarbeiten

Syntax

```
#include <stdarg.h>

Optional
#include <varargs.h> ❑

type va_arg(va_list ap, type);
```

Beschreibung

Die Makros `va_arg`, `va_start` und `va_end` erlauben es, portable Prozeduren mit variablen Argumentlisten, wie sie in `stdarg.h` definiert sind, zu schreiben. Sie dienen zur Bearbeitung einer Argumentliste, deren Anzahl und Typ bei jedem Funktionsaufruf variieren kann.

`va_arg` liefert Datentyp und Wert des jeweils nächsten Arguments der variablen Argumentliste *ap*, beginnend mit dem ersten Argument. Technisch gesehen expandiert das Makro zu einem Ausdruck von Datentyp und Wert des Arguments.

Vor dem ersten Aufruf von `va_arg` muss die variable Argumentliste, auf die *ap* zeigt, mit `va_start` initialisiert worden sein. Jeder `va_arg`-Aufruf verändert *ap* so, dass der Wert des jeweils nächsten Arguments zur Verfügung steht.

ap ist ein Zeiger auf die Argumentliste, die vor dem ersten Aufruf von `va_arg` mit `va_start` initialisiert wurde.

type ist ein Typname, der zum Typ des aktuellen Argumentes passt. Es sind alle C-Datentypen zulässig, für die gilt: Ein Zeiger auf ein Objekt vom Typ *type* ist durch ein einfaches Anfügen von * an *type* definiert. Unzulässig sind z.B. Vektor- und Funktionstypen.

Falls es kein nächstes Argument gibt oder *type* nicht zum aktuellen Argument passt, ist das Verhalten undefiniert.

Returnwert Wert des ersten Arguments

wenn `va_arg()` das erste Mal nach `va_start` aufgerufen wurde. Dieses Argument liegt hinter dem letzten „benannten“ Argument *parmN* in der Formalparameterliste (siehe `va_start()`). Darauf folgende Aufrufe liefern sukzessive die restlichen Argumentwerte.

Hinweis

Die Kompatibilität von Argumenttypen wird vom C-Laufzeitsystem dahingehend unterstützt, dass ähnliche Typen in derselben Weise in der Parameterliste abgelegt werden und zwar: Alle `unsigned`-Typen (inkl. `char`) werden wie `unsigned int` dargestellt (rechtsbündig in einem Wort). Alle anderen ganzzahligen Typen werden wie `int` dargestellt (rechtsbündig in einem Wort). `float` wird wie `double` dargestellt (rechtsbündig in einem Doppelwort).

Vor der Rückkehr einer Funktion, deren Argumentliste mit `va_arg` abgearbeitet wurde, muss `va_end` aufgerufen werden.

Siehe auch `va_start()`, `va_end()`, `stdarg.h`, `varargs.h`.

va_end - variable Argumentliste abschließen

Syntax

#include <stdarg.h>

Optional
#include <varargs.h> ❑

void va_end(va_list *ap*);

Beschreibung

Die Makros `va_end`, `va_start` und `va_arg` erlauben es, portable Prozeduren mit variablen Argumentlisten, wie sie in `stdarg.h` definiert sind, zu schreiben. Sie dienen zur Bearbeitung einer Argumentliste, deren Anzahl und Typ bei jedem Funktionsaufruf variieren kann.

`va_end` führt Abschlussarbeiten an der variablen Argumentliste *ap* durch. Das Makro muss vor der Rückkehr aus einer Funktion aufgerufen werden, deren Argumentliste mit `va_start` und `va_arg` abgearbeitet wurde.

ap ist die Argumentliste, die abgearbeitet wurde. Für eine weitere Verwendung ist die Argumentliste mit `va_start` neu zu initialisieren, da `va_end` die Argumentliste *ap* verändert.

Siehe auch `va_arg()`, `va_start()`, `stdarg.h`, `varargs.h`.

va_start - variable Argumentliste initialisieren

Syntax

```
#include <stdarg.h>

Optional
#include <varargs.h> ❑

void va_start(va_list ap, parmN);
```

Beschreibung

Die Makros `va_start`, `va_arg` und `va_end` erlauben es, portable Prozeduren mit variablen Argumentlisten, wie sie in `stdarg.h` definiert sind, zu schreiben. Sie dienen zur Bearbeitung einer Argumentliste, deren Anzahl und Typ bei jedem Funktionsaufruf variieren kann.

`va_start` initialisiert die variable Argumentliste *ap* für nachfolgende `va_arg`- und `va_end`-Aufrufe.

ap ist ein Zeiger auf die Argumentliste.

parmN ist der Name des letzten Arguments der variablen Argumentliste. Funktionen, die variable Argumentlisten verarbeiten, müssen mindestens ein Argument definieren.

Returnwert

Anzahl der ausgegebenen Zeichen	bei Erfolg.
0	bei Fehler.

Hinweis

Das Verhalten ist undefiniert, wenn *parmN* einen unzulässigen Datentyp hat oder wenn der Datentyp nicht zum aktuellen Argument passt.

Die Kompatibilität von Argumenttypen wird vom C-Laufzeitsystem dahingehend unterstützt, dass ähnliche Typen in derselben Weise in der Parameterliste abgelegt werden, und zwar: Alle `unsigned`-Typen (inkl. `char`) werden wie `unsigned int` dargestellt (rechtsbündig in einem Wort). Alle anderen ganzzahligen Typen werden wie `int` dargestellt (rechtsbündig in einem Wort). `float` wird wie `double` dargestellt (rechtsbündig in einem Doppelwort).

Siehe auch `va_arg()`, `va_end()`, `stdarg.h`, `varargs.h`.

valloc - auf Seitengrenze ausgerichteten Speicher anfordern

Syntax

```
#include <stdlib.h>

void *valloc (size_t size);
```

Beschreibung

`valloc()` hat die gleiche Wirkung wie `malloc()`, nur dass der zugewiesene Speicherbereich auf Seitengrenze ausgerichtet ist, d.h. ein ganzzahliges Vielfaches des Rückgabewertes von `sysconf(_SC_PAGESIZE)`.

Wenn gilt *size* = 0, gibt `valloc()` einen Nullzeiger zurück, `errno` wird in diesem Falle nicht gesetzt.

Returnwert

Zeiger auf den zugewiesenen Speicherbereich	bei Erfolg.
Nullzeiger	sonst. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `valloc()` schlägt fehlt, wenn gilt

`ENOMEM`	es steht nicht genügend Speicherplatz zur Verfügung.

Hinweis Statt `valloc()` sollten Anwendungen besser `malloc()` oder `mmap()` verwenden. In Systemen mit großer Seitengröße ist es möglich, dass `valloc()` nicht erfolgreich aufgerufen werden kann.

`valloc()` wird in der nächsten Version des X/Open-Standards nicht mehr unterstützt.

Siehe auch `malloc()`, `sysconf()`, `stdlib.h`.

vfork - neuen Prozess im virtuellen Speicher erzeugen

Syntax

```
#include <unistd.h>

pid_t vfork (void);
```

Beschreibung

`vfork()` wird auf `fork()` abgebildet. Beschreibung siehe dort.

Returnwert

0 bzw. PID	bei Erfolg. 0 wird an den Kindprozess und die Prozessnummer des Kindprozesses an den Vaterprozess zurückgeliefert.
-1	an den Vaterprozess bei Fehler. Es wird kein Kindprozess erzeugt. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler

`vfork()` schlägt fehl, wenn gilt:

`EAGAIN`	Die systembedingte Grenze der systemweit oder je Benutzer maximal möglichen Prozesse würde überschritten werden. Diese Grenzwerte werden festgelegt, wenn das System erzeugt wird.
`ENOMEM`	Der Swap-Bereich ist für den neuen Prozess nicht groß genug.

Siehe auch `exec()`, `exit()`, `fork()`, `wait()`, `unistd.h`.

vfprintf, vprintf, vsprintf - variable Argumentliste formatiert schreiben

Syntax

```
#include <stdarg.h>
#include <stdio.h>

int vprintf(const char *format, va_list ap);
int vfprintf(FILE *stream, const char *format, va_list ap);
int vsprintf(char *s, const char *format, va_list ap);
```

Beschreibung

`vfprintf()`, `vprintf()` und `vsprintf()` entsprechen jeweils den Funktionen `fprintf()`, `printf()` und `sprintf()` mit folgendem Unterschied: Sie werden statt mit einer variablen Argumentanzahl mit einer Argumentliste, wie sie in `stdarg.h` definiert ist, aufgerufen. Die Argumente der Liste sind in Anzahl und Datentyp zum Übersetzungszeitpunkt nicht bekannt.

Da die `vprint`-Funktionen das `va_arg`-Makro, aber nicht das `va_end`-Makro aufrufen, ist der Wert von *ap* nach der Rückkehr der Funktionen unbestimmt.

Returnwert Siehe `fprintf()`.

Fehler Siehe `fprintf()`.

Hinweis Nach Verwendung dieser Funktionen sollten Sie das Makro `va_end(ap)` aufrufen, um den Zeiger *ap* wieder auf einen definierten Wert zu setzen, damit eventuell nachfolgende Aufrufe dieser Funktionen korrekte Startwerte haben.

`vfprint()` beginnt in der variablen Argumentliste immer mit dem ersten Argument. Die Ausgabe ab einem beliebigen Argument lässt sich mit entsprechend vielen `va_arg`-Aufrufen vor Aufruf von `vfprintf()` erreichen. Jeder `va_arg`-Aufruf positioniert die Argumentliste um ein Argument weiter.

Ob `vfprintf()` für eine BS2000- oder eine POSIX-Datei ausgeführt wird, hängt von der Programmumgebung ab.

BS2000
Die ANSI-Syntax der Formatzeichenkette gilt sowohl im KR-Modus (nur bei C/C++ Versionen kleiner V3 vorhanden) als auch im ANSI-Modus (festgelegt mit dem `LANGUAGE-STANDARD`-Operanden der `SOURCE-PROPERTIES`-Option). ❑

Siehe auch `fprintf()`, `stdarg.h`, `stdio.h`, `varargs.h`.

vfwprintf - Langzeichen formatiert ausgeben

Syntax

```
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>

int vfwprintf(FILE *dz, const wchar_t *format, va_list arg);
```

Beschreibung

Ausführliche Beschreibung siehe `fwprintf()`.

vprintf - Formatierte Ausgabe auf Standardausgabe

Syntax

```
#include <stdio.h>

int vprintf(const char *format, va_list arg);
```

Beschreibung

`vprintf()` gleicht der Funktion `printf()`. Im Unterschied zu `printf()` erlaubt `vprintf()` die Ausgabe von Argumenten, deren Anzahl und Datentyp zum Übersetzungszeitpunkt nicht bekannt sind.
`vprintf()` wird innerhalb von Funktionen benutzt, an die der Aufrufer jeweils eine andere Formatzeichenkette sowie andere auszugebende Argumente übergeben kann. Die Formalparameterliste der Funktionsdefinition sieht dafür eine Formatzeichenkette *format* und eine variable Argumentenliste ", ..." vor.
format ist eine Formatzeichenkette wie bei `printf()` mit ANSI-Funktionalität beschrieben (siehe dort).

`vprintf()` arbeitet eine Argumentenliste *arg* mit internen `va_arg`-Aufrufen sukzessive ab und schreibt die Argumente gemäß der Formatzeichenkette *format* auf die Standardausgabe `stdout`. Die variable Argumentenliste *arg* muss vor dem Aufruf von `vprintf()` mit dem Makro `va_start` initialisiert worden sein.

Returnwert

Anzahl der ausgegebenen Zeichen	bei Erfolg.
Integer< 0	bei Fehler.

Hinweise

`vprintf()` beginnt in der variablen Argumentenliste immer mit dem ersten Argument. Die Ausgabe ab einem beliebigen Argument lässt sich mit entsprechend vielen `va_arg`-Aufrufen vor Aufruf der Funktion `vprintf()` erreichen. Jeder `va_arg`-Aufruf positioniert die Argumentenliste um ein Argument weiter.

`vprintf()` ruft nicht das Makro `va_end` auf. Da `vprintf()` das Makro `va_arg` benutzt, ist der Wert von *arg* nach der Rückkehr unbestimmt.

Siehe auch `vfprintf()`, `vsprintf()`

vsprintf - Formatierte Ausgabe in eine Zeichenkette

Syntax #include <stdio.h>

int vsprintf(char **s*, const char **format*, va_list *arg*);

Beschreibung

`vsprintf()` gleicht der Funktion `sprintf()`. Im Unterschied zu `sprintf()` erlaubt `vsprintf()` die Ausgabe von Argumenten, deren Anzahl und Datentyp zum Übersetzungszeitpunkt nicht bekannt sind.
`vsprintf()` wird innerhalb von Funktionen benutzt, an die der Aufrufer jeweils eine andere Formatzeichenkette sowie andere auszugebende Argumente übergeben kann. Die Formalparameterliste der Funktionsdefinition sieht dafür eine Formatzeichenkette *format* und eine variable Argumentenliste ", ..." vor.

`vsprintf()` arbeitet eine Argumentenliste *arg* mit internen va_arg-Aufrufen sukzessive ab und schreibt die Argumente gemäß der Formatzeichenkette *format* in die Zeichenkette *s*. Die variable Argumentenliste *arg* muss vor dem Aufruf von `vsprintf()` mit dem Makro `va_start` initialisiert worden sein.

Die Funktion verfügt über folgende Parameter:

char **s*
: Zeiger auf die Ergebniszeichenkette. `vsprintf()` schließt die Zeichenkette mit dem Nullbyte (\0) ab.

const char **format*
: Formatzeichenkette wie bei `printf()` mit ANSI-Funktionalität (Beschreibung siehe dort).

  Es gibt nur bzgl. der Steuerzeichen für Zwischenraum (\n, \t, etc.) folgenden Unterschied: `vsprintf()` trägt in die Ergebniszeichenkette den EBCDIC-Wert des Steuerzeichens ein. Erst bei der Ausgabe in Textdateien werden die Steuerzeichen je nach Art der Textdatei in ihre entsprechende Wirkung umgesetzt (siehe Abschnitt „Zwischenraumzeichen“ auf Seite 85).

va_list *arg*
: Zeiger auf die variable Argumentenliste, die mit `va_start` initialisiert wurde.

Returnwert Anzahl der in *s* gespeicherten Zeichen. Das durch `vsprintf()` generierte abschließende Nullbyte (\0) wird dabei nicht mitgezählt.

Hinweise `vsprintf()` beginnt in der variablen Argumentenliste immer mit dem ersten Argument. Die Ausgabe ab einem beliebigen Argument lässt sich mit entsprechend vielen `va_arg`-Aufrufen vor Aufruf der Funktion `vsprintf()` erreichen. Jeder `va_arg`-Aufruf positioniert die Argumentenliste um ein Argument weiter.

`vsprintf()` ruft nicht das Makro `va_end` auf. Da `vsprintf()` das Makro `va_arg` benutzt, ist der Wert von *arg* nach der Rückkehr unbestimmt.

Bei sich überlappenden Speicherbereichen ist das Verhalten undefiniert.

Siehe auch `vfprintf()`, `vprintf()`

vswprintf - Langzeichen formatiert ausgeben

Syntax
```
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>

int vswprintf(wchar_t *s, size_t n, const wchar_t *format, va_list arg);
```

Beschreibung

Ausführliche Beschreibung siehe `fwprintf()`.

vwprintf - Langzeichen formatiert ausgeben

Syntax
```
#include <stdarg.h>
#include <wchar.h>

int vwprintf(const wchar_t *format, va_list arg);
```

Beschreibung

Ausführliche Beschreibung siehe `fwprintf()`.

wait, waitpid - auf Halt oder Ende eines Kindprozesses warten

Syntax

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

Optional
#include <sys/types.h> ❑

pid_t wait (int *stat_loc);
pid_t waitpid (pid_t pid, int *stat_loc, int options);
```

Beschreibung

`wait()` und `waitpid()` erlauben es dem aufrufenden Prozess, den Status eines seiner Kindprozesse zu ermitteln. Wenn der Status für zwei oder mehr Kindprozesse verfügbar ist, ist die Reihenfolge, in der diese Informationen geliefert werden, undefiniert.

`wait()` unterbricht die Ausführung des aufrufenden Prozesses, bis der Endestatus für einen Kindprozess verfügbar ist, oder bis zur Zustellung eines Signals, dessen Signalaktion entweder die Ausführung einer Signalbehandlungsfunktion oder `SIG_DFL` ist. Wenn der Status bereits vor dem Aufruf von `wait()` verfügbar ist, erfolgt eine sofortige Rückkehr von `wait()`.

`waitpid()` verhält sich identisch zu `wait()`, wenn *pid* den Wert `(pid_t)`-1 und *options* den Wert 0 hat. Andernfalls wird das Verhalten von `waitpid()` durch die Werte der Argumente *pid* und *options* verändert.

pid gibt eine Menge von Kindprozessen an, für die der Status ermittelt werden soll. `waitpid()` liefert nur den Status eines Kindprozesses aus dieser Menge zurück:

- Wenn *pid* gleich `(pid_t)`-1 ist, wird der Status irgendeines Kindprozesses abgefragt. In dieser Hinsicht ist `waitpid()` dann äquivalent zu `wait()`.
- Wenn *pid* größer als 0 ist, gibt dieses Argument die Prozessnummer eines einzelnen Kindprozesses an, für den der Status abgefragt wird.
- Wenn *pid* gleich 0 ist, wird der Status für irgendeinen Kindprozess abgefragt, dessen Prozessgruppennummer gleich der des aufrufenden Prozesses ist.
- Wenn *pid* kleiner als `(pid_t)`-1 ist, wird der Status irgendeines Kindprozesses abgefragt, dessen Prozessgruppennnumer gleich der des Absolutbetrags von *pid* ist.

options wird durch die bitweise inklusive Oder-Verknüpfung mit einem der folgenden Flags gebildet, die in der Include-Datei `sys/wait.h` definiert sind.

`WCONTINUED` `waitpid()` ermittelt den Status für einen durch *pid* angegebenen, fortgesetzten Kindprozess, dessen Status noch nicht abgefragt wurde, seit er nach einem Job Control Stop fortgesetzt wurde.

`WNOHANG` `waitpid()` hält die Ausführung des aufrufenden Prozesses nicht an, wenn der Status für einen der durch *pid* angegebenen Kindprozesse nicht unmittelbar verfügbar ist.

`WUNTRACED` Der Status aller durch *pid* angegebenen Prozesse, die angehalten wurden und deren Status seit dem Anhalten noch nicht geliefert wurde, wird ebenfalls an den aufrufenden Prozess gemeldet.

Wenn `wait()` oder `waitpid()` zurückkehren, weil der Status eines Kindprozesses verfügbar ist, ist der Returnwert dieser Funktionen gleich der Kindprozessnummer. Wenn dann der Wert von *stat_loc* nicht der Nullzeiger ist, wird der Status an der Stelle abgelegt, auf die *stat_loc* zeigt.

Wenn der Endestatus eines Kindprozess geliefert wird, der den Wert 0 aus `main()` zurückgegeben oder der den Wert 0 als Argument an `_exit()` oder `exit()` übergeben hat, ist auch der Wert, der an der Adresse *stat_loc* abgelegt wird, gleich 0. Gleichgültig, welchen Wert diese Information hat, sie kann mit Hilfe der folgenden Makros interpretiert werden, die in `sys/wait.h` definiert sind und die ganzzahlige Ausdrücke berechnen; *stat_val* ist ein ganzzahliger Wert, auf den *stat_loc* zeigt.

`WIFEXITED(`*stat_val*`)`
Berechnet einen Wert ungleich 0 (wahr in C), wenn der Status für einen Kindprozess geliefert wurde, der sich normal beendet hat.

`WEXITSTATUS(`*stat_val*`)`
Wenn der Wert von `WIFEXITED(`*stat_val*`)` ungleich 0 ist, berechnet dieses Makro die niederwertigen 8 Bit des Endestatus, den der Kindprozess an `_exit()` oder `exit()` übergeben hat, bzw. des Werts, den der Kindprozess aus `main()` zurückgegeben hat.

`WIFSIGNALED(`*stat_val*`)`
Berechnet einen Wert ungleich 0, wenn der Status für einen Kindprozess geliefert wurde, der sich durch den Empfang eines Signals beendete, das nicht abgefangen wurde (siehe auch `signal.h`).

`WTERMSIG(`*stat_val*`)`
Wenn der Wert von `WIFSIGNALED(`*stat_val*`)` ungleich 0 ist, berechnet dieses Makro die Signalnummer, die den Abbruch des Kindprozesses verursacht hat.

`WIFSTOPPED(`*stat_val*`)`
Berechnet einen Wert ungleich 0, wenn der Status für einen Kindprozess geliefert wurde, der zurzeit angehalten ist.

`WSTOPSIG(`*stat_val*`)`
Wenn der Wert von `WIFSTOPPED(`*stat_val*`)` ungleich 0 ist, berechnet dieses Makro die Nummer des Signals, das den Kindprozess angehalten hat.

WIFCONTINUED(*stat_val*)
: Berechnet einen Wert ungleich 0, wenn der Status für einen Kindprozess geliefert wurde, der nach einem Job Control Stop fortgesetzt wurde.

Wenn der Status, der an der Stelle *stat_loc* abgelegt ist, dort von einem Aufruf von waitpid() abgelegt wurde, der

- das Flag WUNTRACED , nicht aber das Flag WCONTINUED angegeben hatte, liefert genau eines der Makros WIFEXITED(**stat_loc*), WIFSIGNALED(**stat_loc*) oder WIFSTOPPED(**stat_loc*) einen Wert ungleich 0.
- die Flags WUNTRACED und WCONTINUED angegeben hatte, liefert genau eines der Makros WIFEXITED(**stat_loc*), WIFSIGNALED(**stat_loc*) und WIFSTOPPED(**stat_loc*) und WIFCONTINUED(**stat_loc*) einen Wert ungleich 0.
- weder das Flag WUNTRACED noch das Flag WCONTINUED angegeben hatte, oder von einem Aufruf der Funktion wait() abgelegt wurde, liefert genau eines der Makros WIFEXITED(**stat_loc*) und WIFSIGNALED(**stat_loc*) einen Wert ungleich 0.
- das Flag WCONTINUED, nicht aber das Flag WUNTRACED angegeben hatte, oder von einem Aufruf der Funktion wait(), liefert genau eines der Makros WIFEXITED(**stat_loc*), WIFSIGNALED(**stat_loc*) und WIFCONTINUED(**stat_loc*) einen Wert ungleich 0.

Wenn sich ein Vaterprozess beendet, ohne auf alle seine Kindprozesse zu warten, wird den verbleibenden Kindprozessen eine neue Vaterprozessnummer zugeordnet, nämlich die des Systemprozesses init.

Werden Threads verwendet, so wirken sich die Funktionen wait() und waitpid() auf den Prozess oder auf einen Thread wie folgt: Der rufende Thread wird suspendiert, bis die Statusinformation verfügbar ist.

Returnwert

Kindprozessnummer
: wenn wait() oder waitpid() zurückkehren, weil der Status eines Kindprozesses verfügbar ist.

-1
: wenn wait() oder waitpid() auf Grund der Zustellung eines Signals zurückkehren. errno wird auf EINTR gesetzt.

0
: wenn waitpid() mit dem Flag WNOHANG im Argument *options* aufgerufen wurde und die Funktion mindestens einen Kindprozess durch *pid* angegeben hat.

(pid_t)-1
: bei Fehler. errno wird gesetzt, um den Fehler anzuzeigen.

Fehler `wait()` schlägt fehl, wenn gilt:

`ECHILD`	Der aufrufende Prozess besitzt keine Kindprozessse, auf die nicht gewartet wird.
`EINTR`	Die Funktion wurde von einem Signal unterbrochen. Der Wert des Objekts, auf das *stat_loc* zeigt, ist dann undefiniert.

`waitpid()` schlägt fehl, wenn gilt:

`ECHILD`	Der mit *pid* angegebene Prozess oder die Prozessgruppe existiert nicht, oder er ist kein Kindprozess des aufrufenden Prozesses.
`EINTR`	Die Funktion wurde von einem Signal unterbrochen. Der Wert des Objekts, auf das *stat_loc* zeigt, ist dann undefiniert.
`EINVAL`	*options* ist nicht gültig.

Siehe auch `exec`, `exit()`, `fork()`, `sys/types.h`, `sys/wait.h`.

wait3 - auf Zustandsänderung von Kindprozessen warten

Syntax #include <sys/wait.h>

pid_t wait3(int **stat_loc*, int *options*, struct rusage **resource_usage*);

Beschreibung

`wait3()` liefert dem aufrufenden Prozess Status-Informationen über den angegebenen Kindprozess.

Der Aufruf

```
wait3(stat_loc, options, resource_usage);
```

ist äquivalent zu dem Aufruf

```
waitpid( (pid_t)-1, stat_loc, options);
```

nur dass bei erfolgreicher Ausführung in der angegebenen `rusage`-Struktur *resource_usage* die Status-Informationen für den Kindprozess eingetragen werden, der durch den Rückgabewert identifiziert wird.

`wait3()` ist nicht threadsicher.

Werden Threads verwendet, so wirkt sich diese Funktion auf den Prozess oder auf einen Thread wie folgt aus: `wait3()` - liefert dem aufrufenden Thread Status-Informationen über den angegebenen Kindprozess.

Returnwert siehe `waitpid()`.

Zusätzlich zu den bei `waitpid()` angegebenen Fehlern schlägt `wait3()` fehl, wenn gilt:

`ECHILD` Für den aufrufenden Prozess existieren keine Kindprozesse, auf die nicht gewartet wird oder
die durch das Argument *pid* spezifizierte Gruppe von Prozessen kann nie in den durch *options* angegebenen Status kommen.

Hinweis Wenn ein Vaterprozess beendet wird, ohne auf seine Kindprozesse zu warten, übernimmt der Initialisierungsprozess (Prozess-ID = 1) die Kindprozesse.

Siehe auch `exec`, `exit()`, `fork()`, `pause()`, `sys/wait.h`.

waitid - auf Zustandsänderung von Kindprozessen warten

Syntax #include <wait.h>

int waitid(idtype_t *idtype*, id_t *id*, siginfo_t **infop*, int *options*);

Beschreibung

Der aufrufende Prozess wird durch `waitid()` solange angehalten, bis einer der Kindprozesse den Zustand ändert. Der aktuelle Zustand des betreffenden Kindprozesses wird in die Struktur eingetragen, auf die *infop* zeigt. Wenn ein Kindprozess den Zustand vor dem Aufruf von `waitid()` geändert hat, kehrt `waitid()` sofort zurück.

Die Argumente *idtype* und *id* geben an, auf welche Kindprozesse `waitid()` warten soll.

- Wenn *idtype* gleich `P_PID` ist, wartet `waitid()` auf den Kindprozess, der die Prozessnummer `(pid_t)` *id* hat.
- Wenn *idtype* gleich `P_PGID` ist, wartet `waitid()` auf einen der Kindprozesse mit der Prozessgruppennummer `(pid_t)`*id*.
- Wenn *idtype* gleich `P_ALL` ist, wartet `waitid()` auf einen beliebigen Kindprozess, und *id* wird ignoriert.

Das Argument *options* wird verwendet, um anzugeben, auf welche Zustandsänderungen `waitid()` warten soll. Die Zustandsänderungen werden durch bitweise ODER-Verknüpfung der folgenden Flags angegeben:

`WEXITED` wartet darauf, dass Prozesse terminieren (exit).

`WTRAPPED` wartet darauf, dass ablaufverfolgte Prozesse auf Unterbrechungen stoßen oder einen Haltepunkt erreichen (siehe `ptrace()`).

`WSTOPPED` wartet und liefert den Prozessstatus eines Kindprozesses, welcher nach dem Empfang eines Signals gestoppt hat.

`WCONTINUED` liefert den Status für einen Kindprozess, der angehalten und wieder aufgenommen wurde.

`WNOHANG` kehrt sofort zurück, falls keine Kindprozesse vorhanden sind, auf die gewartet werden müsste.

`WNOWAIT` hält den Prozess, dessen Status in *infop* zurückgegeben wurde, in einem Wartezustand. Der Status dieses Prozesses wird dadurch nicht berührt, es kann erneut auf diesen Prozess gewartet werden, wenn der Aufruf abgeschlossen ist.

infop muss auf eine `siginfo_t`-Struktur zeigen, wie sie in `siginfo()` definiert wird. Kehrt `waitid()` zurück, weil es einen Kindprozess gefunden hat, der die in *idtype* und *options* spezifizierten Bedingungen erfüllt, wird vom System in `siginfo_t` der Zustand dieses Prozesses eingetragen. Das Strukturelement `si_signo` hat immer den Wert `SIGCHILD`.

Werden Threads verwendet, so wirkt sich diese Funktion auf den Prozess oder auf einen Thread wie folgt aus: Der aufrufende Thread wird solange suspendiert, bis einer der Kindprozesse den Zustand ändert.

Returnwert

0	wenn `waitid()` auf Grund der Zustandsänderung eines Kindprozesses zurückkehrt
-1	sonst. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `waitid()` schlägt fehl, wenn wenigstens eine der folgenden Bedingungen erfüllt ist:

`ECHILD`	Für den aufrufenden Prozess existieren keine Kindprozesse, auf die nicht gewartet wird.
`EINTR`	`waitid()` wurde unterbrochen, weil der aufrufende Prozess ein Signal empfangen hat.
`EINVAL`	Für *options* wurde ein ungültiger Wert übergeben oder *idtype* und *id* geben eine ungültige Prozessmenge an.
`EFAULT`	*infop* zeigt auf eine ungültige Adresse.

Siehe auch `exec`, `exit()`, `wait()`, `sys/wait.h`

wcrtomb - Langzeichen in Multibyte-Zeichen umwandeln

Syntax #include <wchar.h>

size_t wcrtomb(char *s, wchar_t *wc*, mbstate_t **ps*);

Beschreibung

Wenn *s* ein Nullzeiger ist, entspricht `wcrtomb()` dem Aufruf
`wcrtomb(buf, L'\0', ps)`
wobei *buf* einen internen Puffer bezeichnet.

Wenn *s* kein Nullzeiger ist, bestimmt `wcrtomb()` die Anzahl der Bytes, die unter Berücksichtigung eventueller Umschalt-Sequenzen zur Darstellung des *wc* entsprechenden Multibyte-Zeichens benötigt werden. Die Ergebnisbytes werden in das Feld geschrieben, auf dessen erstes Element *s* zeigt. Es werden maximal {`MB_CUR_MAX`} Bytes geschrieben.
Ist *wc* ein Nullzeichen, wird ein Nullbyte geschrieben, dem eine Umschalt-Sequenz vorausgehen kann, die den „initial shift“-Zustand wiederherstellt.

Der Ergebniszustand entspricht dem „initial conversion“ Zustand.

Returnwert `(size_t)-1` wenn *wc* kein gültiges Langzeichen darstellt. In `errno` wird der Wert des Makros `EILSEQ` geschrieben. Der Konversions-Zustand ist undefiniert.

Anzahl der in das Feld **s* geschriebenen Bytes
sonst.

Hinweis In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Siehe auch `mblen()`, `mbtowc()`, `wcstombs()`, `wctomb()`

wcscat - zwei Langzeichenketten zusammenfügen

Syntax #include <wchar.h>

wchar_t *wcscat(wchar_t **ws1*, const wchar_t **ws2*);

Beschreibung

`wcscat()` hängt eine Kopie der Langzeichenkette *ws2* an das Ende der Langzeichenkette *ws1* an und liefert einen Zeiger auf *ws1* zurück.

Das Null-Langzeichen (`\0`) am Ende der Langzeichenkette *ws1* wird vom ersten Zeichen der Langzeichenkette *ws2* überschrieben.
`wcscat()` schließt die Langzeichenkette mit dem Null-Langzeichen (`\0`) ab.

Returnwert Zeiger auf die Ergebnis-Langzeichenkette *ws1*.

Hinweis Als Argumente werden Langzeichenketten erwartet, die mit dem Null-Langzeichen (`\0`) abgeschlossen sind.
`wcscat()` überprüft nicht, ob *ws1* groß genug für das Ergebnis ist.

Bei sich überlappenden Speicherbereichen ist das Verhalten undefiniert.

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Siehe auch `wcsncat()`, `wchar.h`.

wcschr - Langzeichenkette nach Langzeichen durchsuchen

Syntax

```
#include <wchar.h>

wchar_t *wcschr(const wchar_t *ws, wint_t wc);
```

Beschreibung

`wcschr()` sucht das erste Vorkommen des Zeichens *wc* in der Langzeichenkette *ws* und liefert bei Erfolg einen Zeiger auf die gesuchte Position in *ws*. Der Wert von *wc* muss einem Zeichen des Typs `wchar_t` entsprechen und muss ein Langzeichen sein, das einem gültigen Zeichen in der aktuellen Lokalität entspricht.

Das abschließende Null-Langzeichen (`\0`) wird als Zeichen mitberücksichtigt.

Returnwert

Zeiger	auf die Position von *wc* in der Langzeichenkette *ws*.
Nullzeiger	wenn *wc* in der Langzeichenkette *ws* nicht enthalten ist.

Hinweis

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Siehe auch `wcsrchr()`, `wchar.h`.

wcscmp - zwei Langzeichenketten vergleichen

Syntax

```
#include <wchar.h>

int wcscmp(const wchar_t *ws1, const wchar_t *ws2);
```

Beschreibung

`wcscmp()` vergleicht zwei Langzeichenketten *ws1* und *ws2* lexikalisch, z.B.:

"Zirkel" ist lexikalisch kleiner als "Zirkus".
"Busse" ist lexikalisch größer als "Bus".

Returnwert Ganzzahliger Wert, und zwar:

< 0	*ws1* ist lexikalisch kleiner als *ws2*.
= 0	*ws1* und *ws2* sind lexikalisch gleich groß.
> 0	*ws1* ist lexikalisch größer als *ws2*.

Hinweis Als Argumente werden Langzeichenketten erwartet, die mit dem Null-Langzeichen (`\0`) abgeschlossen sind.
Es gilt die Ordnung des EBCDIC-Zeichensatzes.

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Siehe auch `wcsncmp()`, `wchar.h`.

wcscoll - zwei Langzeichenketten gemäß LC_COLLATE vergleichen

Syntax #include <wchar.h>

int wcscoll(const wchar_t *ws1, const wchar_t *ws2);

Beschreibung

`wcscoll()` vergleicht zwei Langzeichenketten *ws1* und *ws2* lexikalisch unter Berücksichtigung der in `LC_COLLATE` für die Lokalität festgelegten Sortierreihenfolge.

Returnwert Ganzzahliger Wert, und zwar:

< 0	*ws1* ist bezüglich der festgelegten Sortierreihenfolge kleiner als *ws2*.
= 0	*ws1* und *ws2* sind bezüglich der festgelegten Sortierreihenfolge gleich groß.
> 0	*ws1* ist bezüglich der festgelegten Sortierreihenfolge größer als *ws2*.

Fehler `wcscoll()` schlägt fehl, wenn gilt:

`EINVAL`	Eine der beiden Langzeichenketten lässt sich nicht in eine Multibyte-Zeichenkette umwandeln.

Hinweis Da es im Standard keinen festlegten Wert für den Fehlerfall gibt, wird empfohlen, `errno` auf den Wert 0 zu setzen, dann `wcscoll()` aufzurufen und nach dem Aufruf `errno` zu überprüfen. Falls `errno` ungleich 0 ist, kann angenommen werden, dass ein Fehler aufgetreten ist.

Zum Sortieren großer Listen sollten die Funktionen `wcsxfmr()` und `wcscmp()` verwendet werden.

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Siehe auch `wcsncmp()`, `wcsxfrm()`, `wchar.h`.

wcscpy - Langzeichenkette kopieren

Syntax

```
#include <wchar.h>

wchar_t *wcscpy(wchar_t *ws1, const wchar_t *ws2);
```

Beschreibung

`wcscpy()` kopiert die Langzeichenkette *ws2* einschließlich des Null-Langzeichens (`\0`) in den Speicherbereich, auf den *ws1* zeigt. *ws1* muss groß genug sein, um die Langzeichenkette *ws2* einschließlich des Null-Langzeichens (`\0`) aufnehmen zu können.

Returnwert Zeiger auf die Ergebnis-Langzeichenkette *ws1*.

Hinweis Als Argumente werden Langzeichenketten erwartet, die mit dem Null-Langzeichen (`\0`) abgeschlossen sind.
`wcscpy()` überprüft nicht, ob *ws1* groß genug für das Ergebnis ist.
Bei sich überlappenden Speicherbereichen ist das Verhalten undefiniert.

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Siehe auch `wcsncpy()`, `wchar.h`.

wcscspn - Länge einer komplementären Langzeichenteilkette ermitteln

Syntax #include <wchar.h>

size_t wcscspn(const wchar_t *ws1, const wchar_t *ws2);

Beschreibung

`wcscspn()` berechnet ab Beginn der Langzeichenkette *ws1* die Länge des Segmentes, das kein einziges Zeichen aus der Langzeichenkette *ws2* enthält. Das abschließende Null-Langzeichen (`\0`) gilt nicht als Teil der Langzeichenkette *ws2*.
Sobald ein Zeichen in *ws1* mit einem Zeichen in *ws2* übereinstimmt, wird die Funktion beendet und die Segmentlänge zurückgeliefert.
Wenn bereits das erste Zeichen in *ws1* mit einem Zeichen in *ws2* übereinstimmt, ist die Segmentlänge gleich 0.

Returnwert Ganzzahliger Wert

der die Segmentlänge (Anzahl ungleicher Zeichen) ab Beginn der Langzeichenkette *ws1* angibt.

Hinweis *Einschränkung*
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Siehe auch `wcsspn()`, `wchar.h`.

wcsftime - Datum und Uhrzeit in Langzeichenkette umwandeln

Syntax #include <wchar.h>

size_t wcsftime(wchar_t *_wcs_, size_t _maxsize_, const wchar_t *_format_,
const struct tm *_timptr_);

Beschreibung

`wcsftime()` schreibt Langzeichen-Codes gemäß dem in *format* angegebenen String in das Feld, auf das *wcs* zeigt.

Die Funktion verhält sich so, als ob eine von `strftime()` erzeugte Zeichenkette als Argument an `mbtowcs()` übergeben worden wäre und `mbtowcs()` das Ergebnis wiederum als Langzeichenkette mit maximal *maxsize* Langzeichen-Codes an `wcsftime()` übergibt.

Falls zwischen sich überlappenden Objekten kopiert wird, ist das Ergebnis undefiniert.

Returnwert Ganzzahliger Wert

der die Anzahl der in das Feld geschriebenen Langzeichen-Codes angibt (ohne abschließende Null), wenn die Anzahl der Langzeichen-Codes inklusive der abschließenden Null kleiner oder gleich *maxsize* ist.

0 sonst. In diesem Falle ist der Feldinhalt unbestimmt.

Fehler `wcsftime()` schlägt fehl, wenn gilt:

`ENOMEM` Es steht nicht genügend Speicherplatz für die internen Verwaltungsdaten zur Verfügung.

Siehe auch `strftime()`, `mbtowcs()`, `wchar.h`.

wcslen - Länge einer Langzeichenkette ermitteln

Syntax #include <wchar.h>

size_t wcslen(const wchar_t *ws);

Beschreibung

`wcslen()` bestimmt die Länge der Langzeichenkette *ws*, ohne das abschließende Null-Langzeichen (`\0`).

Returnwert

Länge	der Langzeichenkette *ws*. Das abschließende Null-Langzeichen (`\0`) wird nicht mitgezählt.

Hinweis

Als Argument wird eine Langzeichenkette erwartet, die mit dem Null-Langzeichen (`\0`) abgeschlossen ist.

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Siehe auch `wchar.h`.

wcsncat - zwei Langzeichenteilketten zusammenfügen

Syntax

```
#include <wchar.h>

wchar_t *wcsncat(wchar_t *ws1, const wchar_t *ws2, size_t n);
```

Beschreibung

`wcsncat()` hängt maximal *n* Zeichen der Langzeichenkette *ws2* an das Ende der Langzeichenkette *ws1* an und liefert einen Zeiger auf *ws1* zurück.

Das Null-Langzeichen (`\0`) am Ende der Langzeichenkette *ws1* wird vom ersten Zeichen der Langzeichenkette *ws2* überschrieben.

Wenn die Langzeichenkette *ws2* weniger als *n* Zeichen enthält, werden nur die Zeichen aus *ws2* an *ws1* angehängt. Wenn die Langzeichenkette *ws2* mehr als *n* Zeichen enthält, werden nur die führenden *n* Zeichen von *ws2* an *ws1* angehängt.

`wcsncat()` schließt die Langzeichenkette mit dem Null-Langzeichen (`\0`) ab.

Returnwert

Zeiger	auf die Ergebnis-Langzeichenkette *ws1*.

Hinweis

Als Argumente werden Langzeichenketten erwartet, die mit dem Null-Langzeichen (`\0`) abgeschlossen sind.

`wcsncat()` überprüft nicht, ob *ws1* groß genug für das Ergebnis ist.
Bei sich überlappenden Speicherbereichen ist das Verhalten undefiniert.

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Siehe auch `wcscat()`, `wchar.h`.

wcsncmp - zwei Langzeichenteilketten vergleichen

Syntax #include <wchar.h>

int wcsncmp(const wchar_t *ws1*, const wchar_t *ws2,* size_t *n*);

Beschreibung

`wcsncmp()` vergleicht die Langzeichenketten *ws1* und *ws2* bis zur maximalen Länge *n* lexikalisch; z.B liefert

```
wcsncmp("Sie","Siemens",3)
```

das Ergebnis 0 (gleich), weil die beiden Argumente in den ersten drei Zeichen übereinstimmen.

Returnwert Ganzzahliger Wert, und zwar:

< 0	*ws1* ist in den ersten *n* Zeichen lexikalisch kleiner als *ws2*.
0	*ws1* und *ws2* sind in den ersten *n* Zeichen lexikalisch gleich groß.
> 0	*ws1* ist in den ersten *n* Zeichen lexikalisch größer als *ws2*.

Hinweis Als Argumente werden Langzeichenketten erwartet, die mit dem Null-Langzeichen (`\0`) abgeschlossen sind.

Es gilt die Ordnung des EBCDIC-Zeichensatzes.

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Siehe auch `wcscmp()`, `wchar.h`.

wcsncpy - Langzeichenteilkette kopieren

Syntax

```
#include <wchar.h>

wchar_t *wcsncpy(wchar_t *ws1, const wchar_t *ws2, size_t n);
```

Beschreibung

`wcsncpy()` kopiert maximal *n* Zeichen der Langzeichenkette *ws2* in den Speicherbereich, auf den *ws1* zeigt.

Wenn die Langzeichenkette *ws2* weniger als *n* Zeichen enthält, wird nur in der Länge von *ws2* (wcslen + 1) kopiert, *ws1* wird dann bis zur Länge *n* mit Null-Langzeichen aufgefüllt.

Wenn die Langzeichenkette *ws2 n* Zeichen (ohne das Null-Langzeichen) oder mehr enthält, ist die Langzeichenkette *ws1* nicht automatisch mit dem Null-Langzeichen abgeschlossen.

Wenn die Langzeichenkette *ws1* mehr als *n* Zeichen enthält und das letzte kopierte Zeichen aus *ws2* ist nicht das Null-Langzeichen, bleiben ggf. restliche Daten in *ws1* erhalten.

`wcsncpy()` schließt *ws1* nicht automatisch mit dem Null-Langzeichen ab.

Returnwert Zeiger auf die Ergebnis-Langzeichenkette *ws1*.

Hinweis `wcsncpy()` überprüft nicht, ob der Speicherbereich *ws1* groß genug für das Ergebnis ist!

Da `wcsncpy()` die Ergebnis-Langzeichenkette nicht automatisch mit dem Null-Langzeichen abschließt, kann es häufig notwendig sein, *ws1* explizit mit einem Null-Langzeichen abzuschließen. Das ist z.B. der Fall, wenn nur ein Teilstück aus *ws2* kopiert wird und auch *ws2* kein Null-Langzeichen enthält.

Bei sich überlappenden Speicherbereichen ist das Verhalten undefiniert.

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Siehe auch `wcscpy()`, `wchar.h`.

wcspbrk - erstes Vorkommen eines Langzeichens in Langzeichenkette ermitteln

Syntax #include <wchar.h>

wchar_t *wcspbrk(const wchar_t **ws1*, const wchar_t **ws2*);

Beschreibung

`wcspbrk()` sucht das erste Zeichen in der Langzeichenkette *ws1*, das mit irgendeinem Zeichen aus der Langzeichenkette *ws2* übereinstimmt. Das abschließende Null-Langzeichen (`\0`) gilt nicht als Teil der Langzeichenkette *ws2*.

Returnwert

Zeiger	auf das erste gefundene Zeichen in *ws1*.
Nullzeiger	falls keinerlei Übereinstimmung vorliegt.

Hinweis Als Argumente werden Langzeichenketten erwartet, die mit dem Null-Langzeichen (`\0`) abgeschlossen sind.

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Siehe auch `wcschr()`, `wcsrchr()`, `wchar.h`.

wcsrchr - letztes Vorkommen eines Langzeichens in Langzeichenkette ermitteln

Syntax #include <wchar.h>

wchar_t *wcsrchr(const wchar_t *ws, wint_t wc);

Beschreibung

`wcsrchr()` sucht das letzte Vorkommen des Zeichens *wc* in der Langzeichenkette *ws* und liefert bei Erfolg einen Zeiger auf die gesuchte Position in *ws*.

Das abschließende Null-Langzeichen (`\0`) wird als Zeichen mitberücksichtigt.

Returnwert

Zeiger	auf die Position von *wc* in der Langzeichenkette *ws*.
Nullzeiger	wenn *wc* in der Langzeichenkette *ws* nicht enthalten ist.

Hinweis *Einschränkung*
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Siehe auch `wcschr()`, `wchar.h`.

wcsrtombs - Langzeichenkette in Multibyte-Zeichenkette umwandeln

Syntax

```
#include <wchar.h>

size_t wcsrtombs(char *dst, const wchar_t **src, size_t len, mbstate_t *ps);
```

Beschreibung

`wcsrtombs()` konvertiert eine Folge von Langzeichen aus dem Feld, auf das *src* indirekt zeigt, in Multibyte-Zeichen. `mbsrtowcs()` beginnt die Umwandlung mit dem Konvertierungszustand, der in **ps* beschrieben wird. Die konvertierten Zeichen werden in das Feld geschrieben, auf das *dst* zeigt, sofern *dst* kein Nullzeiger ist. Jedes einzelne Zeichen wird so konvertiert, als sei die Funktion `wcrtomb()` aufgerufen worden.

Die Umwandlung ist beendet, wenn ein abschließendes Nullzeichen auftritt. Das Nullzeichen wird ebenfalls umgewandelt und in das Feld geschrieben.

Die Umwandlung wird vorher abgebrochen, wenn

- eine Bytefolge auftritt, zu der kein gültiges Multibyte-Zeichen korrespondiert oder
- *dst* kein Nullzeiger ist und das nächste Multibyte-Zeichen die Gesamtlänge ***len*** der in das Feld zu schreibenden Bytes übersteigen würde.

Wenn *dst* kein Nullzeiger ist, wird dem Zeigerobjekt, auf das *src* zeigt, einer der beiden folgenden Werte zugewiesen:

- ein Nullzeiger, falls die Umwandlung mit dem Erreichen eines Nullzeichens beendet wurde
- die Adresse direkt hinter dem letzten umgewandelten Langzeichen.

Wenn *dst* kein Nullzeiger ist und die Umwandlung mit dem Erreichen eines Nullzeichens beendet wurde, entspricht der Ergebniszustand dem „initial conversion“ Zustand.

Returnwert

`(size_t)-1` wenn ein Konvertierungsfehler auftritt, das heißt eine Folge von Bytes, zu der kein gültiges Multibyte-Zeichen korrespondiert. In `errno` wird der Wert des Makros `EILSEQ` geschrieben. Der Konversions-Zustand ist undefiniert.

Anzahl der Bytes in der konvertierten Multibyte-Zeichenkette (ohne abschließendes Nullzeichen) sonst.

Siehe auch `mblen()`, `mbtowc()`, `wcstombs()`, `wctomb()`

wcsspn - Länge einer Langzeichenteilkette ermitteln

Syntax

```
#include <wchar.h>

size_t wcsspn(const wchar_t *ws1, const wchar_t *ws2);
```

Beschreibung

`wcsspn()` berechnet ab Beginn der Langzeichenkette *ws1* die Länge des Segmentes, das ausschließlich Zeichen aus der Langzeichenkette *ws2* enthält.

Sobald ein Zeichen in *ws1* mit keinem Zeichen in *ws2* übereinstimmt, wird die Funktion beendet und die Segmentlänge zurückgeliefert.

Wenn bereits das erste Zeichen in *ws1* mit keinem Zeichen in *ws2* übereinstimmt, ist die Segmentlänge gleich 0.

Returnwert Ganzzahliger Wert
der die Segmentlänge (Anzahl passender Zeichen) ab Beginn der Langzeichenkette *ws1* angibt.

Hinweis Als Argumente werden Langzeichenketten erwartet, die mit dem Null-Langzeichen (`\0`) abgeschlossen sind.

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Siehe auch `wcscspn()`, `wchar.h`.

wcsstr - erstes Vorkommen einer Langzeichenkette suchen

Syntax #include <wchar.h>

wchar_t *wcsstr(const wchar_t *ws1, const wchar_t *ws2);

Beschreibung

`wcsstr()` sucht das erste Vorkommen der Langzeichenkette *ws2* (ohne das abschließende Nullzeichen) in der Langzeichenkette *ws1*.

Returnwert Zeiger auf den Beginn der gefundenen Zeichenkette
wenn *ws2* in *ws1* gefunden wird.

Nullzeiger wenn *ws2* in *ws1* nicht gefunden wird.

ws1 wenn *ws2* ein Nullzeiger ist.

Hinweis Für C++ gelten die beiden folgenden Prototypen für die Funktion `wcsstr()`:
const wchar_t* wcsstr(const wchar_t *ws1, const wchar_t *ws2);
wchar_t* wcsstr(wchar_t *ws1, const wchar_t *ws2);

Siehe auch `strstr()`, `wmemcmp()`, `wmemcpy()`, `wmemchr()`

wcstod - Langzeichenkette in Gleitkommazahl (double) umwandeln

Syntax #include <wchar.h>

double wcstod(const wchar_t **nptr*, wchar_t ***endptr*);

Beschreibung

`wcstod()` wandelt den ersten Teil der Zeichenkette aus Langzeichenwerten, auf die *nptr* zeigt, in eine Darstellung mit doppelter Genauigkeit um. Zuerst wird die Eingabe-Zeichenkette aus Langzeichenwerten in drei Teile zerlegt:

– eine möglicherweise leere Folge von Zwischenraumzeichen als Langzeichenwerte (entsprechend der Angabe durch `iswspace()`) am Anfang,

– eine Folge, die als Gleitkommakonstante interpretiert wird,

– und schließlich eine Zeichenkette aus Langzeichenwerten mit einem oder mehr nicht erkannten Langzeichenwerten, einschließlich abschließendem Nullbyte der Eingabe-Zeichenkette aus Langzeichenwerten.

Dann wird versucht, die mittlere Folge in eine Gleitkommazahl umzuwandeln. Anschließend wird das Ergebnis zurückgegeben.

Es wird erwartet, dass diese mittlere Folge folgendes Format hat:
Das Vorzeichen + oder - (optional), eine nichtleere Folge von Ziffern, die optional ein Dezimalzeichen enthalten kann, und schließlich ein optionaler Exponententeil. Ein Exponententeil besteht aus dem Zeichen `e` bzw. `E`, gefolgt von einem Vorzeichen (optional) und einer oder mehreren dezimalen Ziffern. Diese mittlere Folge ist als die längste Teilfolge der Eingabe-Zeichenkette aus Langzeichenwerten definiert. Sie beginnt mit dem ersten Langzeichenwert, der kein Zwischenraumzeichen ist und das erwartete Format aufweist. Diese Folge enthält keine Langzeichenwerte, wenn die Eingabe-Zeichenkette aus Langzeichenwerten leer ist oder nur aus Langzeichenwerten besteht, die Zwischenraumzeichen sind, bzw. wenn der erste Langzeichenwert, der kein Zwischenraumzeichen ist, etwas anderes ist als ein Vorzeichen, eine Ziffer oder ein Dezimalzeichen.

Wenn diese mittlere Folge das erwartete Format aufweist, wird die Folge der Langzeichenwerte, die mit der ersten Ziffer oder dem Dezimalzeichen beginnt (je nachdem, was zuerst steht), als Gleitkommakonstante entsprechend der Definition in der Sprache C interpretiert. Der Unterschied besteht darin, dass das Dezimalzeichen statt des Punktes verwendet wird und, wenn weder ein Exponententeil noch ein Dezimalzeichen erscheint, nach der letzten Ziffer in der Zeichenkette aus Langzeichenwerten ein Dezimalzeichen angenommen wird. Wenn die Folge mit einem Minuszeichen beginnt, ist das Ergebnis der Umwandlung negativ. Ein Zeiger auf die letzte Zeichenkette aus Langzeichenwerten wird in dem Objekt abgelegt, auf das *endptr* zeigt, wenn *endptr* kein Nullzeiger ist.

Das Dezimalzeichen ist in der Lokalität des Programms definiert (Kategorie `LS_NUMERIC`). In der `POSIX`-Lokalität, bzw. in einer Lokalität, in der das Dezimalzeichen nicht definiert ist, ist das Dezimalzeichen standardmäßig der Punkt (.).

In einer anderen als der POSIX-Lokalität können andere Formate für die mittlere Folge zulässig sein. Wenn diese mittlere Folge leer ist oder nicht das erwartete Format aufweist, wird keine Umwandlung durchgeführt. Der Wert von *nptr* wird in dem Objekt abgelegt, auf das *endptr* zeigt, wenn *endptr* kein Nullzeiger ist.

Returnwert konvertierter Wert bei Erfolg.

0	wenn keine Umwandlung durchgeführt werden konnte.
`HUGE_VAL`	wenn der richtige Wert außerhalb des Bereichs der darstellbaren Werte liegt (entsprechend dem Vorzeichen des Wertes). `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `wcstod()` schlägt fehl, wenn gilt:

`ERANGE`	Der Wert, der zurückgegeben werden soll, würde zu einem Überlauf oder einem Unterlauf führen.

Hinweis Da 0 sowohl bei einem Fehler zurückgegeben wird als auch bei Erfolg einen gültigen Returnwert darstellt, muss eine Anwendung, die auf Fehler prüfen will, die folgenden Aktionen ausführen: `errno` wird auf 0 gesetzt, `wcstod()` aufgerufen und der Wert von `errno` überprüft. Falls dieser Wert ungleich null ist, wird angenommen, dass ein Fehler aufgetreten ist.

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Siehe auch `iswspace()`, `localeconv()`, `scanf()`, `setlocale()`, `wcstol()`, `wchar.h`.

wcstok - Langzeichenkette in Langzeichenteilkette zerlegen

Syntax

```
#include <wchar.h>

wchar_t *wcstok(wchar_t *ws1, const wchar_t *ws2);
```

Beschreibung

Mit `wcstok()` lässt sich eine Gesamtlangzeichenkette *ws1* in Langzeichenteilketten - sog. "Tokens" - zerlegen, z.B. ein Satz in die einzelnen Wörter oder eine Quellprogrammanweisung in die kleinsten syntaktischen Einheiten. Der Zeiger auf *ws1* darf nur beim ersten `wcstok`-Aufruf übergeben werden. Ab dem zweiten Aufruf ist ein Nullzeiger anzugeben.

Beginn- und Endekriterium für jedes Token sind Trennzeichen (Separatoren), die in einer zweiten Langzeichenkette *ws2* anzugeben sind. Token können durch einen oder mehrere dieser Separatoren begrenzt sein, bzw. durch Beginn und Ende der Gesamtlangzeichenkette *ws1*. Typische Separatoren zwischen den Wörtern eines Satzes sind z.B. Leerzeichen, Doppelpunkt, Komma etc.

Pro Aufruf bearbeitet `wcstok()` genau eine Langzeichenteilkette. Der erste Aufruf liefert einen Zeiger auf den Beginn der ersten gefundenen Langzeichenteilkette, die weiteren Aufrufe jeweils einen Zeiger auf den Beginn der nächsten Langzeichenteilketten. Jede Langzeichenteilkette schließt `wcstok()` mit dem Null-Langzeichen (`\0`) ab.

Bei jedem Aufruf kann eine andere Trennzeichenfolge *ws2* angegeben werden.

Returnwert Zeiger auf den Beginn einer Langzeichenteilkette.

Beim ersten Aufruf ein Zeiger auf die erste Langzeichenteilkette, beim nächsten Aufruf ein Zeiger auf die nachfolgende Langzeichenteilkette etc. `wcstok()` schließt jede Langzeichenteilkette in *ws1* mit einem Null-Langzeichen (`\0`) ab, wobei das jeweils erste gefundene Trennzeichen mit dem Null-Langzeichen (`\0`) überschrieben wird.

Nullzeiger, falls keine bzw. keine weitere Langzeichenteilkette gefunden wurde.

Hinweis *Einschränkung*

In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Siehe auch `wchar.h`.

wcstol - Langzeichenkette in ganze Zahl (long) umwandeln

Syntax #include <wchar.h>

long int wcstol(const wchar_t *_nptr_, wchar_t **_endptr_, int _base_);

Beschreibung

`wcstol()` wandelt den ersten Teil der Zeichenkette aus Langzeichenwerten, auf die *nptr* zeigt, in die Darstellung `long int` um. Zuerst wird die Eingabe-Zeichenkette aus Langzeichenwerten in drei Teile zerlegt:

- eine möglicherweise leere Folge von Zwischenraumzeichen als Langzeichenwerte (entsprechend der Angabe durch `iswspace()`) am Anfang,
- ein Folge, die als ganze Zahl mit einer Dezimalzeichen-Darstellung interpretiert wird, die durch den Wert von *base* bestimmt wird,
- und schließlich eine Zeichenkette aus Langzeichenwerten mit einem oder mehr nicht erkannten Langzeichenwerten, einschließlich abschließendem Nullbyte der Eingabe-Zeichenkette aus Langzeichenwerten.

Dann wird versucht, die mittlere Folge in eine ganze Zahl umzuwandeln. Anschließend wird das Ergebnis zurückgegeben.

Wenn der Wert von *base* gleich null ist, wird als Format der mittleren Folge eine dezimale Konstante, oktale Konstante oder hexadezimale Konstante erwartet. Dieser kann + bzw. - vorangestellt sein. Eine dezimale Konstante beginnt mit einer Ziffer ungleich Null und besteht aus einer Folge dezimaler Ziffern. Eine oktale Konstante besteht aus dem Präfix 0 und optional einer Folge nur dezimaler Ziffern. Eine hexadezimale Konstante besteht aus dem Präfix 0x bzw. 0X und einer Folge dezimaler Ziffern und der Buchstaben a (bzw. A) bis f (bzw. F) mit den Werten 10 bis 15.

Wenn der Wert von *base* zwischen 2 und 36 liegt, wird als Format der Folge eine Folge von Buchstaben und Ziffern erwartet, die eine ganze Zahl mit der Basis, die durch *base* bestimmt wird, darstellt (allerdings keine ganze Zahl mit Suffix). Optional kann das Vorzeichen + bzw. - vorangestellt sein. Den Buchstaben von a (bzw. A) bis einschließlich z (bzw. Z) sind die Werte 10 bis 35 zugeordnet. Es sind nur Buchstaben zulässig, deren Wert kleiner ist als der Wert von *base*. Ist der Wert von *base* gleich 16, können die Darstellungen 0x bzw. 0X für Langzeichenwerte, gegebenenfalls mit Vorzeichen, der Zeichen- und Buchstabenfolge voranstehen.

Diese mittlere Folge ist als die längste beginnende Teilfolge der Eingabe-Zeichenkette aus Langzeichenwerten definiert. Sie beginnt mit dem ersten Langzeichenwert, der kein Zwischenraumzeichen ist und das erwartete Format aufweist. Diese Folge enthält keine Langzeichenwerte, wenn die Eingabe-Zeichenkette aus Langzeichenwerten leer ist oder nur aus

Langzeichenwerten besteht, die Zwischenraumzeichen sind, bzw. wenn der erste Langzeichenwert, der kein Zwischenraumzeichen ist, etwas anderes als ein Vorzeichen oder ein zulässiger Buchstabe bzw. eine zulässige Ziffer ist.

Wenn diese mittlere Folge das erwartete Format aufweist und der Wert von *base* gleich null ist, wird die Folge der Langzeichenwerte, die mit der ersten Ziffer beginnt, als Integer-Konstante interpretiert. Wenn diese mittlere Folge das erwartete Format aufweist und der Wert von *base* zwischen 2 und 36 liegt, wird sie als Grundlage für die Umwandlung verwendet. Jedem Buchstaben wird sein Wert (siehe oben) zugeordnet. Wenn die Folge mit einem Minuszeichen beginnt, ist das Ergebnis der Umwandlung negativ. Wenn *endptr* kein Nullzeiger ist, wird ein Zeiger auf die abschließende Zeichenkette aus Langzeichenwerten in dem Objekt abgelegt, auf das *endptr* zeigt.

Wenn diese mittlere Folge leer ist oder nicht das erwartete Format aufweist, wird keine Umwandlung durchgeführt. Der Wert von *nptr* wird in dem Objekt abgelegt, auf das *endptr* zeigt, wenn *endptr* kein Nullzeiger ist.

Returnwert konvertierter Wert
bei Erfolg.

0 wenn keine Umwandlung durchgeführt werden konnte.

`LONG_MAX, LONG_MIN`
wenn der richtige Wert außerhalb des Bereichs der darstellbaren Werte liegt (entsprechend dem Vorzeichen des Wertes).
`errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `wcstol()` schlägt fehl, wenn gilt:

`EINVAL` Der Wert von *base* wird nicht unterstützt.

`ERANGE` Der Wert, der zurückgegeben werden soll, ist nicht darstellbar.

Hinweis Da 0, `LONG_MIN` und `LONG_MAX` sowohl bei einem Fehler zurückgegeben werden als auch bei Erfolg gültige Returnwerte darstellen, muss eine Anwendung, die auf Fehler prüfen will, die folgenden Aktionen ausführen: `errno` wird auf 0 gesetzt, `wcstol()` aufgerufen und der Wert von `errno` überprüft. Falls dieser Wert ungleich null ist, wird angenommen, dass ein Fehler aufgetreten ist.

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Siehe auch `iswalpha()`, `scanf()`, `wcstod()`, `wchar.h`.

wcstoll - Langzeichenkette in ganze Zahl (long long) umwandeln

Syntax #include <wchar.h>

long long int wcstoll(const wchar_t *restrict *nptr*, wchar_t **restrict *endptr*, int *base*);

Beschreibung

`wcstoll()` wandelt den ersten Teil der Zeichenkette aus Langzeichenwerten, auf die *nptr* zeigt, in die Darstellung `long long int` um. Zuerst wird die Eingabe-Zeichenkette aus Langzeichenwerten in drei Teile zerlegt:

– eine möglicherweise leere Folge von Zwischenraumzeichen als Langzeichenwerte (entsprechend der Angabe durch `iswspace()`) am Anfang,

– eine Folge, die als ganze Zahl mit einer Dezimalzeichen-Darstellung interpretiert wird, die durch den Wert von *base* bestimmt wird,

– und schließlich eine Zeichenkette aus Langzeichenwerten mit einem oder mehr nicht erkannten Langzeichenwerten, einschließlich abschließendem Nullbyte der Eingabe-Zeichenkette aus Langzeichenwerten.

Dann wird versucht, die mittlere Folge in eine ganze Zahl umzuwandeln. Anschließend wird das Ergebnis zurückgegeben.

Wenn der Wert von *base* gleich null ist, wird als Format der mittleren Folge eine dezimale Konstante, oktale Konstante oder hexadezimale Konstante erwartet. Dieser kann + bzw. - vorangestellt sein. Eine dezimale Konstante beginnt mit einer Ziffer ungleich Null und besteht aus einer Folge dezimaler Ziffern. Eine oktale Konstante besteht aus dem Präfix 0 und optional einer Folge nur dezimaler Ziffern. Eine hexadezimale Konstante besteht aus dem Präfix 0x bzw. 0X und einer Folge dezimaler Ziffern und der Buchstaben a (bzw. A) bis f (bzw. F) mit den Werten 10 bis 15.

Wenn der Wert von *base* zwischen 2 und 36 liegt, wird als Format der mittleren Folge eine Sequenz von Buchstaben und Ziffern erwartet, die eine ganze Zahl darstellt mit der Basis, die durch *base* bestimmt wird (allerdings keine ganze Zahl mit Suffix). Optional kann das Vorzeichen + bzw. - vorangestellt sein. Den Buchstaben von a (bzw. A) bis einschließlich z (bzw. Z) sind die Werte 10 bis 35 zugeordnet. Es sind nur Buchstaben zulässig, deren Wert kleiner ist als der Wert von *base*. Ist der Wert von *base* gleich 16, können die Darstellungen 0x bzw. 0X für Langzeichenwerte, gegebenenfalls mit Vorzeichen, der Zeichen- und Buchstabenfolge voranstehen.

Diese mittlere Folge ist als die längste beginnende Teilfolge der Eingabe-Zeichenkette aus Langzeichenwerten definiert. Sie beginnt mit dem ersten Langzeichenwert, der kein Zwischenraumzeichen ist und das erwartete Format aufweist. Diese Folge enthält keine Langzeichenwerte, wenn die Eingabe-Zeichenkette aus Langzeichenwerten leer ist oder nur aus

Langzeichenwerten besteht, die Zwischenraumzeichen sind, bzw. wenn der erste Langzeichenwert, der kein Zwischenraumzeichen ist, etwas anderes als ein Vorzeichen oder ein zulässiger Buchstabe bzw. eine zulässige Ziffer ist.

Wenn diese mittlere Folge das erwartete Format aufweist und der Wert von *base* gleich null ist, wird die Folge der Langzeichenwerte, die mit der ersten Ziffer beginnt, als Integer-Konstante interpretiert. Wenn diese mittlere Folge das erwartete Format aufweist und der Wert von *base* zwischen 2 und 36 liegt, wird sie als Grundlage für die Umwandlung verwendet. Jedem Buchstaben wird sein Wert (siehe oben) zugeordnet. Wenn die Folge mit einem Minuszeichen beginnt, ist das Ergebnis der Umwandlung negativ. Wenn *endptr* kein Nullzeiger ist, wird ein Zeiger auf die abschließende Zeichenkette aus Langzeichenwerten in dem Objekt abgelegt, auf das *endptr* zeigt.

Wenn diese mittlere Folge leer ist oder nicht das erwartete Format aufweist, wird keine Umwandlung durchgeführt. Der Wert von *nptr* wird in dem Objekt abgelegt, auf das *endptr* zeigt, wenn *endptr* kein Nullzeiger ist.

Returnwert

konvertierter Wert
: bei Erfolg.

`0`
: wenn keine Umwandlung durchgeführt werden konnte. `errno` wird auf `EINVAL` gesetzt, wenn der Wert von *base* nicht unterstützt wird.

`LLONG_MAX`, `LLONG_MIN`
: abgängig vom Vorzeichen des Wertes.

`ULLONG_MAX`
: wenn der richtige Wert außerhalb des Bereichs der darstellbaren Werte liegt. `errno` wird auf `ERANGE` gesetzt, um den Fehler anzuzeigen.

Fehler

Da 0 sowohl bei einem Fehler zurückgegeben wird als auch bei Erfolg einen gültigen Returnwert darstellt, muss eine Anwendung, die auf Fehler prüfen will, die folgenden Aktionen ausführen: `errno` wird auf 0 gesetzt, `wcstoll()` aufgerufen und der Wert von `errno` überprüft. Falls dieser Wert ungleich null ist, wird angenommen, dass ein Fehler aufgetreten ist.

Hinweise

In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Siehe auch

`iswalpha()`, `iswspace()`, `scanf()`, `strtol()`, `strtoll()`, `strtoul()`, `strtoull()`, `wcstod()`, `wcstol()`, `wcstoul()`

wcstombs - Langzeichenkette in Zeichenkette umwandeln

Syntax #include <stdlib.h>

size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);

Beschreibung

`wcstombs()` wandelt eine Folge von `wchar_t`-Werten in *pwcs* in die entsprechenden Multibyte-Zeichen um und speichert diese in die Zeichenkette *s*.
n gibt die maximale Anzahl Bytes an, die in *s* abgespeichert werden sollen.

In dieser Version sind Zeichen, die aus mehreren Bytes bestehen, nicht realisiert. Multibyte-Zeichen haben immer die Länge 1 Byte und `wchar_t`-Werte sind immer `long`-Werte. Mit `wcstombs()` wird jeder `wchar_t`-Wert (Typ `long`) in *pwcs* einem 1 Byte langen Bereich in der Zeichenkette *s* zugewiesen.

Die Zuweisung wird beendet, wenn:

– der `wchar_t`-Wert 0 in *pwcs* auftritt,
– bereits *n* Bytes zugewiesen wurden oder
– ein `wchar_t`-Wert nicht in einem Byte dargestellt werden kann.

Returnwert Anzahl der zugewiesenen Bytes
bei erfolgreicher Umwandlung.

`(size_t)-1` wenn ein `wchar_t`-Wert nicht in ein Multibyte-Zeichen umgewandelt werden kann.

Hinweis Wenn ein `wchar_t`-Wert in *pwcs* nicht in ein Multibyte-Zeichen umgewandelt werden kann, werden die bereits vorher umgewandelten `wchar_t`-Werte in *s* abgespeichert.

Bei sich überlappenden Speicherbereichen ist das Verhalten undefiniert.

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Siehe auch `mblen()`, `mbtowc()`, `mbstowcs()`, `wctomb()`, `stdlib.h`.

wcstoul - Langzeichenkette in ganze Zahl (unsigned long) umwandeln

Syntax #include <wchar.h>

unsigned long int wcstoul(const wchar_t **nptr*, wchar_t ***endptr*, int *base*);

Beschreibung

`wcstoul()` wandelt den ersten Teil der Zeichenkette aus Langzeichenwerten, auf die *nptr* zeigt, in die Darstellung `unsigned long int` um. Zuerst wird die Eingabe-Zeichenkette aus Langzeichenwerten in drei Teile zerlegt:

– eine möglicherweise leere Folge von Zwischenraumzeichen als Langzeichenwerte (entsprechend der Angabe durch `iswspace()`) am Anfang,

– ein Folge, die als ganze Zahl mit einer Dezimalzeichen-Darstellung interpretiert wird, die durch den Wert von *base* bestimmt wird,

– und schließlich eine Zeichenkette aus Langzeichenwerten mit einem oder mehr nicht erkannten Langzeichenwerten, einschließlich abschließendem Nullbyte der Eingabe-Zeichenkette aus Langzeichenwerten.

Dann wird versucht, die mittlere Folge in eine ganze Zahl umzuwandeln. Anschließend wird das Ergebnis zurückgegeben.

Wenn der Wert von *base* gleich null ist, so wird als Format der mittleren Folge eine dezimale Konstante, oktale Konstante oder hexadezimale Konstante erwartet. Dieser kann + bzw. - vorangestellt sein. Eine dezimale Konstante beginnt mit einer Ziffer ungleich Null und besteht aus einer Folge dezimaler Ziffern. Eine oktale Konstante besteht aus dem Präfix 0 und optional einer Folge nur dezimaler Ziffern. Eine hexadezimale Konstante besteht aus dem Präfix 0x bzw. 0X und einer Folge dezimaler Ziffern und der Buchstaben a (bzw. A) bis f (bzw. F) mit den Werten 10 bis 15.

Wenn der Wert von *base* zwischen 2 und 36 liegt, wird als Format der Folge eine Folge von Buchstaben und Ziffern erwartet, die eine ganze Zahl mit der Basis, die durch *base* bestimmt wird, darstellt (allerdings keine ganze Zahl mit Suffix). Optional kann das Vorzeichen + bzw. - vorangestellt sein. Den Buchstaben von a (bzw. A) bis einschließlich z (bzw. Z) sind die Werte 10 bis 35 zugeordnet. Es sind nur Buchstaben zulässig, deren Wert kleiner ist als der Wert von *base*. Ist der Wert von *base* gleich 16, können die Darstellungen 0x bzw. 0X für Langzeichenwerte, gegebenenfalls mit Vorzeichen, der Zeichen- und Buchstabenfolge voranstehen.

Diese mittlere Folge ist als die längste beginnende Teilfolge der Eingabe-Zeichenkette aus Langzeichenwerten definiert. Sie beginnt mit dem ersten Langzeichenwert, der kein Zwischenraumzeichen ist und das erwartete Format aufweist. Diese Folge enthält keine Langzeichenwerte, wenn die Eingabe-Zeichenkette aus Langzeichenwerten leer ist oder nur aus

Langzeichenwerten besteht, die Zwischenraumzeichen sind, bzw. wenn der erste Langzeichenwert, der kein Zwischenraumzeichen ist, etwas anderes als ein Vorzeichen oder ein zulässiger Buchstabe bzw. eine zulässige Ziffer ist.

Wenn diese mittlere Folge das erwartete Format aufweist und der Wert von *base* gleich null ist, wird die Folge der Langzeichenwerte, die mit der ersten Ziffer beginnt, als Integer-Konstante interpretiert. Wenn diese mittlere Folge das erwartete Format aufweist und der Wert von *base* zwischen 2 und 36 liegt, wird sie als Grundlage für die Umwandlung verwendet. Jedem Buchstaben wird sein Wert (siehe oben) zugeordnet. Wenn die Folge mit einem Minuszeichen beginnt, ist das Ergebnis der Umwandlung negativ. Wenn *endptr* kein Nullzeiger ist, wird ein Zeiger auf die abschließende Zeichenkette aus Langzeichenwerten in dem Objekt abgelegt, auf das *endptr* zeigt.

Wenn diese mittlere Folge leer ist oder nicht das erwartete Format aufweist, wird keine Umwandlung durchgeführt. Der Wert von *nptr* wird in dem Objekt abgelegt, auf das *endptr* zeigt, wenn *endptr* kein Nullzeiger ist.

Returnwert konvertierter Wert
bei Erfolg.

0	wenn keine Umwandlung durchgeführt werden konnte.
`ULONG_MAX`	wenn der richtige Wert außerhalb des Bereichs der darstellbaren Werte liegt (entsprechend dem Vorzeichen des Wertes). `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `wcstoul()` schlägt fehl, wenn gilt:

`EINVAL`	Der Wert von *base* wird nicht unterstützt.
`ERANGE`	Der Wert, der zurückgegeben werden soll, ist nicht darstellbar.

Hinweis Da 0 und `ULONG_MAX` sowohl bei einem Fehler zurückgegeben werden, 0 aber auch bei Erfolg einen gültigen Returnwert darstellt, muss eine Anwendung, die auf Fehler prüfen will, die folgenden Aktionen ausführen: `errno` wird auf 0 gesetzt, `wcstoul()` aufgerufen und der Wert von `errno` überprüft. Falls dieser Wert ungleich null ist, wird angenommen, dass ein Fehler aufgetreten ist. Anders als `wcstod()` und `wcstol()`, muss `wcstoul()` immer eine nicht negative Zahl zurückgeben. Die Verwendung des Returnwerts von `wcstoul()` für Zahlen, die außerhalb des Bereichs liegen, für die Funktion `wcstoul()` kann also zu schwerwiegenderen Problemen führen als zu Genauigkeitsproblemen, wenn diese Zahlen negativ sein können.

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Siehe auch `iswalpha()`, `scanf()`, `wcstod()`, `wcstol()`, `wchar.h`.

wcstoull - Langzeichenkette in ganze Zahl (unsigned long long) umwandeln

Syntax #include <wchar.h>

unsigned long long int wcstoull(const wchar_t *restrict *nptr*, wchar_t **restrict *endptr*, int *base*);

Beschreibung

`wcstoull()` wandelt den ersten Teil der Zeichenkette aus Langzeichenwerten, auf die *nptr* zeigt, in die Darstellung `unsigned long int` um. Zuerst wird die Eingabe-Zeichenkette aus Langzeichenwerten in drei Teile zerlegt:

– eine möglicherweise leere Folge von Zwischenraumzeichen als Langzeichenwerte (entsprechend der Angabe durch `iswspace()`) am Anfang,

– ein Folge, die als ganze Zahl mit einer Dezimalzeichen-Darstellung interpretiert wird, die durch den Wert von *base* bestimmt wird,

– und schließlich eine Zeichenkette aus Langzeichenwerten mit einem oder mehr nicht erkannten Langzeichenwerten, einschließlich abschließendem Nullbyte der Eingabe-Zeichenkette aus Langzeichenwerten.

Dann wird versucht, die mittlere Folge in eine ganze Zahl vom Typ `unsigned long int` umzuwandeln. Anschließend wird das Ergebnis zurückgegeben.

Wenn der Wert von *base* gleich null ist, wird als Format der mittleren Folge eine dezimale Konstante, oktale Konstante oder hexadezimale Konstante erwartet. Dieser kann + bzw. - vorangestellt sein. Eine dezimale Konstante beginnt mit einer Ziffer ungleich Null und besteht aus einer Folge dezimaler Ziffern. Eine oktale Konstante besteht aus dem Präfix 0 und optional einer Folge nur dezimaler Ziffern. Eine hexadezimale Konstante besteht aus dem Präfix 0x bzw. 0X und einer Folge dezimaler Ziffern und der Buchstaben a (bzw. A) bis f (bzw. F) mit den Werten 10 bis 15.

Wenn der Wert von *base* zwischen 2 und 36 liegt, wird als Format der mittleren Folge eine Sequenz von Buchstaben und Ziffern erwartet, die eine ganze Zahl darstellt mit der Basis, die durch *base* bestimmt wird (allerdings keine ganze Zahl mit Suffix). Optional kann das Vorzeichen + bzw. - vorangestellt sein. Den Buchstaben von a (bzw. A) bis einschließlich z (bzw. Z) sind die Werte 10 bis 35 zugeordnet. Es sind nur Buchstaben zulässig, deren Wert kleiner ist als der Wert von *base*. Ist der Wert von *base* gleich 16, können die Darstellungen 0x bzw. 0X für Langzeichenwerte, gegebenenfalls mit Vorzeichen, der Zeichen- und Buchstabenfolge voranstehen.

Diese mittlere Folge ist als die längste beginnende Teilfolge der Eingabe-Zeichenkette aus Langzeichenwerten definiert. Sie beginnt mit dem ersten Langzeichenwert, der kein Zwischenraumzeichen ist und das erwartete Format aufweist. Diese Folge enthält keine Langzeichenwerte, wenn die Eingabe-Zeichenkette aus Langzeichenwerten leer ist oder nur aus

Langzeichenwerten besteht, die Zwischenraumzeichen sind, bzw. wenn der erste Langzeichenwert, der kein Zwischenraumzeichen ist, etwas anderes als ein Vorzeichen oder ein zulässiger Buchstabe bzw. eine zulässige Ziffer ist.

Wenn diese mittlere Folge das erwartete Format aufweist und der Wert von *base* gleich null ist, wird die Folge der Langzeichenwerte, die mit der ersten Ziffer beginnt, als Integer-Konstante interpretiert. Wenn diese mittlere Folge das erwartete Format aufweist und der Wert von *base* zwischen 2 und 36 liegt, wird sie als Grundlage für die Umwandlung verwendet. Jedem Buchstaben wird sein Wert (siehe oben) zugeordnet. Wenn die Folge mit einem Minuszeichen beginnt, ist das Ergebnis der Umwandlung negativ. Wenn *endptr* kein Nullzeiger ist, wird ein Zeiger auf die abschließende Zeichenkette aus Langzeichenwerten in dem Objekt abgelegt, auf das *endptr* zeigt.

Wenn diese mittlere Folge leer ist oder nicht das erwartete Format aufweist, wird keine Umwandlung durchgeführt. Der Wert von *nptr* wird in dem Objekt abgelegt, auf das *endptr* zeigt, wenn *endptr* kein Nullzeiger ist.

Returnwert

konvertierter Wert
: bei Erfolg.

`0`
: wenn keine Umwandlung durchgeführt werden konnte. `errno` wird auf `EINVAL` gesetzt, wenn der Wert von *base* nicht unterstützt wird.

`LLONG_MAX, LLONG_MIN`
: abgängig vom Vorzeichen des Wertes.

`ULLONG_MAX`
: wenn der richtige Wert außerhalb des Bereichs der darstellbaren Werte liegt. `errno` wird auf `ERANGE` gesetzt, um den Fehler anzuzeigen.

Fehler

Da 0 sowohl bei einem Fehler zurückgegeben wird als auch bei Erfolg einen gültigen Returnwert darstellt, muss eine Anwendung, die auf Fehler prüfen will, die folgenden Aktionen ausführen: `errno` wird auf 0 gesetzt, `wcstoull()` aufgerufen und der Wert von `errno` überprüft. Falls dieser Wert ungleich null ist, wird angenommen, dass ein Fehler aufgetreten ist.

Hinweise

In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Siehe auch

`iswalpha(), iswspace(), scanf(), strtoul(), wcstod(), wcstol()`

wcswcs - Langzeichenteilkette in Langzeichenkette ermitteln

Syntax #include <wchar.h>

wchar_t *wcswcs(const wchar_t **ws1*, const wchar_t **ws2*);

Beschreibung

`wcswcs()` sucht das erste Vorkommen der Langzeichenkette *ws2* (ohne das abschließende Null-Langzeichen) in der Langzeichenkette *ws1*.

Returnwert

Zeiger	auf den Beginn der gefundenen Langzeichenkette in *ws1*.
Nullzeiger	wenn *ws2* in *ws1* nicht enthalten ist.
Zeiger	auf den Beginn von *ws1*, wenn *ws2* die Länge 0 hat.

Hinweis Als Argumente werden Langzeichenketten erwartet, die mit dem Null-Langzeichen (`\0`) abgeschlossen sind.

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Siehe auch `wcschr()`, `wchar.h`.

wcswidth - Spaltenanzahl einer Langzeichenkette ermitteln

Syntax #include <wchar.h>

int wcswidth(const wchar_t **pwcs*, size_t *n*);

Beschreibung

`wcswidth()` bestimmt die Anzahl der Spaltenpositionen, die für *n* Zeichen in der Langzeichenkette *pwcs* benötigt werden. Falls vor dem Erreichen von *n* Zeichen ein Null-Langzeichen (`\0`) angetroffen wird, werden weniger als *n* Zeichen bearbeitet.

Returnwert Anzahl der Spaltenpositionen für die Langzeichenkette *pwcs*.

0	falls *pwcs* auf ein Null-Langzeichen zeigt.
-1	falls *pwcs* ein nicht abdruckbares Zeichen enthält.

Hinweis *Einschränkung*
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Siehe auch `wchar.h`.

wcsxfrm - Langzeichenkette transformieren

Syntax #include <wchar.h>

size_t wcsxfrm(wchar_t *ws1, const wchar_t *ws2, size_t n);

Beschreibung

`wcsxfrm()` transformiert die Langzeichenkette, auf die *ws2* zeigt, und schreibt das Ergebnis der Transformation in das Feld, auf das *ws1* zeigt. Die Transformation wird so durchgeführt, dass die Funktion `wcscmp()` für zwei transformierte Langzeichenketten denselben Returnwert (größer, gleich oder kleiner null) liefert, wie die Funktion `wcscoll()` für die beiden ursprünglichen, nicht transformierten Langzeichenketten.
Es werden maximal *n* Langzeichen-Codes in das Feld geschrieben (inklusive des abschließenden Null-Zeichens).

Wenn *n* den Wert 0 hat, darf *wc1* ein Nullzeiger sein.

Falls zwischen sich überlappenden Objekten kopiert wird, ist das Ergebnis undefiniert.

Returnwert Ganzzahliger Wert < n
der die Anzahl der in das Feld geschriebenen Langzeichen-Codes angibt (ohne abschließende Null).

Ganzzahliger Wert ≥ n
In diesem Falle ist der Inhalt des Feldes *ws1* unbestimmt.

(size_t) - 1 bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `wcsxfrm()` schlägt fehl, wenn gilt:

`EINVAL` Die Langzeichenkette, auf die *ws2* zeigt, enthält Langzeichen-Codes, die außerhalb des Wertebereichs der gewählten Sortierfolge liegen.

`ENOMEM` Es steht nicht genügend Speicherplatz für die internen Verwaltungsdaten zur Verfügung.

Hinweis Es wird so transformiert, dass zwei transformierte Langzeichenketten von `wcscmp()` gemäß der in `LC_COLLATE` festgelegten Sortierfolge geordnet werden.

Die Tatsache, dass *ws1* ein Nullzeiger sein darf, wenn *n* den Wert 0 hat, ist nützlich, wenn die Größe des Feldes vor der Transformation bestimmt werden soll.

Da es im Standard keinen festlegten Wert für den Fehlerfall gibt, wird empfohlen, `errno` auf den Wert 0 zu setzen, dann `wcscoll()` aufzurufen und nach dem Aufruf `errno` zu überprüfen. Falls `errno` ungleich 0 ist, kann angenommen werden, dass ein Fehler aufgetreten ist.

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Siehe auch `wcscmp()`, `wcscoll()`, `wchar.h`.

wctob - Langzeichen in (1-Byte) Multibyte-Zeichen umwandeln

Syntax
```
#include <stdio.h>
#include <wchar.h>

int wctob(wint_t c);
```

Beschreibung
`wctob()` überprüft, ob das Zeichen *c* zu einem Element des erweiterten Zeichensatzes korrespondiert, dessen Multibyte-Darstellung im „initial shift“-Zustand aus einem Byte besteht.

Returnwert
`EOF` falls zu *c* kein korrespondierendes Multibyte-Zeichen der Länge eins im „initial shift“-Zustand existiert.

Multibyte-Zeichen der Länge 1 Byte, das zu *c* korrespondiert.
sonst.

Siehe auch `mblen()`, `mbtowc()`, `wcstombs()`, `wctomb()`

wctomb - Langzeichen in Zeichen umwandeln

Syntax

```
#include <stdlib.h>

int wctomb(char *s, wchar_t wchar);
```

Beschreibung

`wctomb()` wandelt den `wchar_t`-Wert *wchar* in das entsprechende Multibyte-Zeichen um und speichert dieses in die Zeichenkette *s*.

In dieser Version sind Zeichen, die aus mehreren Bytes bestehen, nicht realisiert. Multibyte-Zeichen haben immer die Länge 1 Byte und `wchar_t`-Werte sind immer `long`-Werte. Mit `wctomb()` wird der Wert *wchar* (Typ `long`) dem 1 Byte langen Bereich *s* zugewiesen.

Keine Zuweisung erfolgt, wenn *s* ein Nullzeiger ist oder wenn der `wchar_t`-Wert nicht in einem Byte dargestellt werden kann.

Returnwert

0	falls *s* ein Nullzeiger ist.
-1	falls der `wchar_t`–Wert nicht in ein Multibyte-Zeichen umgewandelt werden kann.
1	sonst in allen anderen Fällen.

Hinweis

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Siehe auch `mblen()`, `mbstowcs()`, `mbtowc()`, `wcstombs()`, `stdlib.h`.

wctrans - Abbildung zwischen Langzeichen definieren

Syntax

```
#include <wctype.h>

wctrans_t wctrans(const char *property);
```

Beschreibung

`wctrans()` konstruiert aus *property* einen Wert des Typs `wctrans_t`, der eine Abbildung zwischen Langzeichen beschreibt.

Die beiden Zeichenketten "`tolower`" und "`toupper`" sind in allen Lokalitäten als Werte des Arguments *property* zugelassen.

Wenn *property* eine Abbildung identifiziert, die gemäß der `LC_CTYPE`-Kategorie der aktuellen Lokalität gültig ist, gibt `wctrans()` einen Wert ungleich 0 zurück, der als gültiges zweites Argument der Funktion `towctrans()` verwendet werden kann.

Returnwert

Wert ≠ `0`	wenn *property* eine gültige Abbildung identifiziert.
`0`	sonst.

Hinweis

In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Siehe auch `towctrans()`

wctype - Langzeichenklasse definieren

Syntax

```
#include <wchar.h>

wctype_t wctype(const char *charclass);
```

Beschreibung

`wctype()` ist für gültige Namen von Zeichenklassen definiert, wie sie in der aktuellen Umgebung festgelegt sind. *charclass* ist eine Zeichenkette, die eine generische Zeichenklasse angibt, für die zeichensatzspezifische Typinformationen benötigt werden. Die folgenden Namen von Zeichenklassen sind in jeder Umgebung definiert: `"alnum"`, `"alpha"`, `"blank"`, `"cntrl"`, `"digit"`, `"graph"`, `"lower"`, `"print"`, `"punct"`, `"space"`, `"upper"` und `"xdigit"`.

Es können weitere Namen von Zeichenklassen angegeben werden, wenn sie in der Definitionsdatei der Umgebung definiert sind (Kategorie `LC_CTYPE`).

Die Funktion gibt einen Wert vom Typ `wctype_t` zurück. Dieser Wert kann als zweites Argument für einen Darauf folgenden Aufruf von `iswctype()` verwendet werden. `wctype()` bestimmt entsprechend den Regeln des durch die Zeichentyp-Informationen der Umgebung (Kategorie `LC_CTYPE`) definierten Zeichensatzes `wctype_t`-Werte. Die von `wctype()` zurückgegebenen Werte sind solange gültig, bis ein Aufruf von `setlocale()` die Kategorie `LC_CTYPE` modifiziert.

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Returnwert

0	wenn der Name der Zeichenklasse in der aktuellen Lokalität nicht gültig ist (Kategorie `LC_CTYPE`).
$\neq 0$	εσ wird ein Objekt vom Typ `wctype_t` zurückgegeben, das in Aufrufen von `iswctype()` verwendet werden kann.

Siehe auch `iswctype()`, `wchar.h`.

wcwidth - Spaltenanzahl eines Langzeichens ermitteln

Syntax

```
#include <wchar.h>

int wcwidth(wint_t wc);
```

Beschreibung

`wcwidth()` bestimmt die Anzahl der Spaltenpositionen, die für den Langzeichenwert *wc* benötigt werden. Der Wert von *wc* muss ein Zeichen sein, das als `wchar_t` darstellbar ist. Außerdem muss dieser Wert ein Langzeichenwert sein, der in der aktuellen Lokalität einem gültigen Zeichen entspricht.

Returnwert

-1	wenn *wc* keinem darstellbaren Langzeichenwert entspricht.
0	wenn *wc* ein Langzeichen-Nullbyte ist.
1	wenn *wc* einem darstellbaren Langzeichenwert entspricht.

Hinweis

Einschränkung
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt. Sie sind vom Typ `wchar_t` (siehe `stddef.h`). ❑

Siehe auch `wchar.h`.

wmemchr - Langzeichenkette nach Langzeichen durchsuchen

Syntax #include <wchar.h>

wchar_t *wmemchr(const wchar_t *ws, wchar_t *wc, size_t n);

Beschreibung

`wmemchr()` sucht das erste Vorkommen des Langzeichens *wc* in den ersten *n* Bytes der Langzeichenkette *ws* und liefert bei Erfolg einen Zeiger auf die gesuchte Position in *ws*.

Returnwert Zeiger auf die Position von *wc* in *ws*
bei Erfolg,

Nullzeiger sonst.

Hinweise In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Für C++ gelten die beiden folgenden Prototypen für die Funktion `wmemchr()`:
const wchar_t* wmemchr(const wchar_t *ws, wchar_t *wc, size_t n);
wchar_t* wmemchr(wchar_t *ws, wchar_t *wc, size_t n);

Siehe auch `memchr()`, `wcsstr()`, `wmemcmp()`, `wmemcpy()`

wmemcmp - zwei Langzeichenketten vergleichen

Syntax #include <wchar.h>

int wmemcmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);

Beschreibung

`wmemcmp()` vergleicht die ersten *n* Bytes der beiden Langzeichenketten *ws1* und *ws2* lexikalisch.

Returnwert

< 0	*ws1* ist lexikalisch kleiner als *ws2*.
= 0	*ws1* und *ws2* sind lexikalisch gleich groß.
> 0	*ws1* ist lexikalisch größer als *ws2*.

Hinweis In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Siehe auch `memcmp(), wcsstr(), wmemchr(), wmemcpy()`

wmemcpy - Langzeichenkette kopieren

Syntax #include <wchar.h>

wchar_t *wmemcpy(wchar_t *ws1, const wchar_t *ws2, size_t n);

Beschreibung

`wmemcpy()` kopiert die ersten *n* Bytes der Langzeichenkette *ws2* in die ersten *n* Bytes der Langzeichenkette *ws1*.

Returnwert Zeiger auf die Langzeichenkette *ws1*.

Hinweis In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Siehe auch `memcmp(), wmemmove(), wmemset()`

wmemmove - Langzeichenkette in überlappenden Bereich kopieren

Syntax #include <wchar.h>

wchar_t *wmemmove(wchar_t *_ws1_, const wchar_t *_ws2_, size_t _n_);

Beschreibung

`wmemmove()` kopiert die ersten _n_ Bytes der Langzeichenkette _ws2_ in die ersten _n_ Bytes der Langzeichenkette _ws1_. Das Kopieren findet statt, als ob die _n_ Langzeichen zuerst in ein temporäres Feld kopiert würden, das weder _ws1_ noch _ws2_ überlappt, und anschließend von diesem Feld in _ws1_.

Returnwert Zeiger auf die Langzeichenkette _ws1_.

Hinweis In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Siehe auch `memmove()`, `wmemcpy()`, `wmemset()`

wmemset - erste _n_ Langzeichen in Langzeichenkette setzen

Syntax #include <wchar.h>

wchar_t *wmemset(wchar_t *_ws_, wchar_t *_c_, size_t _n_);

Beschreibung

`wmemset()` setzt die ersten _n_ Langzeichen in der Langzeichenkette _ws_ auf den Wert _c_.

Returnwert Zeiger auf _ws_.

Hinweis In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Siehe auch `memset()`, `wmemcpy()`, `wmemmove()`

wprintf - Langzeichen formatiert ausgeben

Syntax #include <wchar.h>

int wprintf(const wchar_t *_format_ [, _arglist_]);

Beschreibung

Ausführliche Beschreibung siehe `fwprintf()`.

write - Bytes in Datei schreiben

Syntax #include <unistd.h>

BS2000
#include <stdio.h> ❑

ssize_t write(int *fildes*, const void **buf*, size_t *nbyte*);

Beschreibung

`write()` versucht, *nbyte* Bytes aus dem Puffer, auf den *buf* zeigt, in die dem Dateideskriptor *fildes* zugeordnete Datei zu schreiben.

BS2000
SAM-Dateien werden mit elementaren Funktionen stets als Textdateien verarbeitet. ❑

In einer Datei, in der positioniert werden kann, beginnt die Schreiboperation an der Stelle in der Datei, die durch den mit *fildes* verbundenen Lese-/Schreibzeiger bestimmt ist. Vor einer erfolgreichen Rückkehr von `write()` wird der Zeiger um die Anzahl von Bytes erhöht, die tatsächlich geschrieben wurden. Wenn in einer normalen Datei der erhöhte Zeiger über das Dateiende hinauszeigt, wird die Dateilänge mit der Zeigerposition gleichgesetzt.

Wenn als Dateistatus-Flag `O_SYNC` gesetzt ist und *fildes* auf eine normale Datei zeigt, kehrt ein erfolgreicher `write`-Aufruf erst dann zurück, wenn die Daten physikalisch aktualisiert wurden.

In einer Datei, in der nicht positioniert werden kann, findet das Schreiben immer an der aktuellen Position statt. Der Wert des Lese-/Schreibzeigers, der einem solchen Gerät zugeordnet ist, ist undefiniert.

Wenn als Dateistatus-Flag `O_APPEND` gesetzt ist, wird der Lese-/Schreibzeiger vor jeder Schreiboperation auf das Dateiende gesetzt. Die Datei wird zwischen dem Setzen des Lese-/Schreibzeigers und dem Beginn der `write()`-Operation nicht verändert.

Wenn `write()` mehr Bytes schreiben will, als Platz zur Verfügung steht (z.B. wegen `ulimit()` oder dem physikalischen Ende eines Mediums), werden nur so viel Bytes geschrieben, wie noch Platz haben. Angenommen, in einer Datei ist noch Platz für 20 Bytes vorhanden, bevor eine Grenze erreicht wird. Eine Schreiboperation von 512 Bytes liefert dann den Returnwert 20. Die nächste Schreiboperation mit einer Byteanzahl ungleich 0 würde dann ein Fehlerergebnis liefern (außer in den unten beschriebenen Fällen) und dem Prozess ein SIGXFSZ-Signal schicken.

Wenn `write()` von einem Signal unterbrochen wird, bevor es Daten geschrieben hat, wird -1 zurückgegeben und `errno` auf `EINTR` gesetzt.
Wenn `write()` von einem Signal unterbrochen wird, nachdem es Daten erfolgreich geschrieben hat, wird die Anzahl der geschriebenen Bytes zurückgegeben.

Wenn write() erfolgreich in eine normale Datei geschrieben hat, gilt Folgendes:

– Jeder erfolgreiche read-Aufruf von jeder Byteposition in der durch eine Schreiboperation veränderten Datei liefert die von write() für diese Position spezifizierten Daten zurück, bis diese Bytepositionen erneut geändert werden.

– Jeder folgende write-Aufruf für dieselbe Byteposition in der Datei überschreibt diese Daten.

Eine Schreibanforderung für eine Pipe oder FIFO wird genauso behandelt wie solche für eine normale Datei, mit folgenden Ausnahmen:

– Es gibt keine Dateiposition, die einer Pipe zugeordnet ist, da jede Schreibanforderung an das Ende der Datei angefügt wird.

– Schreibanforderungen von {PIPE_BUF} oder weniger Bytes werden nicht mit Daten anderer Prozesse gemischt, die auf dieselbe Datei schreiben. Eine Schreiboperation von mehr als {PIPE_BUF} Bytes kann, in bestimmten Grenzen, mit Schreiboperationen anderer Prozesse gemischt werden, gleichgültig, ob das Flag O_NONBLOCK im System-Dateistatus-Byte gesetzt ist oder nicht.

– Wenn das Flag O_NONBLOCK nicht gesetzt ist, kann eine Schreibanforderung den Prozess blockieren, aber eine normale Beendigung liefert das Ergebnis *nbyte*.

– Wenn das Flag O_NONBLOCK gesetzt ist, wird die Anforderung von write() verschieden behandelt, wie folgt:

  – write() blockiert den Prozess nicht.

  – Eine Schreibanforderung für {PIPE_BUF} oder weniger Bytes hat eine der folgenden Auswirkungen:

    a) Wenn in der Pipe genügend Platz zur Verfügung steht, überträgt write() alle Daten und gibt die Anzahl der angeforderten Bytes zurück.

    b) Wenn in der Pipe nicht genügend Platz zur Verfügung steht, überträgt write() keine Daten, liefert den Wert -1 zurück und setzt errno gleich EAGAIN.

  – Eine Schreibanforderung mit mehr als {PIPE_BUF} Bytes hat eine der folgenden Auswirkungen:

    a) Wenn mindestens 1 Byte geschrieben werden kann, überträgt write() so viel Daten wie möglich und gibt die Anzahl der geschriebenen Bytes zurück. Wenn alle vorher in eine Pipe geschriebenen Daten gelesen wurden, werden zumindest {PIPE_BUF} Bytes übertragen.

    b) Wenn keine Daten geschrieben werden können, überträgt write() keine Daten, liefert den Wert -1 und setzt errno gleich EAGAIN.

Wenn eine Anforderung größer als {PIPE_BUF} Bytes ist und alle Daten, die vorher in diese Datei geschrieben wurden, bereits gelesen sind, überträgt write() mindestens {PIPE_BUF} Bytes.

Wenn versucht wird, auf einen Dateideskriptor zu schreiben, der keine Pipe oder FIFO ist und nichtblockierendes Schreiben unterstützt, geschieht Folgendes:

– Wenn das Flag O_NONBLOCK nicht gesetzt ist, blockiert write() solange, bis die Daten akzeptiert werden können.

– Wenn das Flag O_NONBLOCK gesetzt ist, blockiert write() den Prozess nicht. Wenn einige Daten ohne ein Blockieren des Prozesses geschrieben werden können, schreibt write() so viel wie möglich und liefert die Anzahl der übertragenen Bytes. Andernfalls liefert die Funktion den Wert -1 und errno wird gleich EAGAIN gesetzt.

Bei erfolgreicher Beendigung, wobei *nbyte* größer als 0 ist, markiert write() die Strukturkomponenten st_ctime und st_mtime der Datei zur Änderung; die Dateistatus-Flags S_ISUID und S_ISGID werden gelöscht, wenn der Prozess keine Sonderrechte besitzt.

Wenn *fildes* einen STREAM bezeichnet, wird die Schreiboperation durch die minimalen und maximalen Werte für *nbyte* bestimmt („Paketgröße“), die vom STREAM akzeptiert werden. Diese Werte werden vom obersten STREAM-Modul festgelegt.
Wenn *nbyte* in der zugelassenen Paketgröße liegt, werden *nbyte* Bytes geschrieben.
Wenn *nbyte* nicht im Bereich der Paketgröße liegt, und die kleinste Paketgröße gleich 0 ist, spaltet write() den Puffer in Segmente mit maximaler Paketgröße auf, bevor die Daten stromabwärts gesendet werden (das letzte Segment kann kleiner sein).
Wenn *nbyte* nicht im Bereich der Paketgröße liegt, und die kleinste Paketgröße ungleich 0 ist, schlägt write() fehl und setzt errno auf ERANGE.

Wird ein Puffer mit Länge 0 (*nbyte* = 0) auf einen STREAM geschrieben, sendet write() eine Nachricht der Länge 0 und gibt den Wert 0 zurück. Wird jedoch ein Puffer mit Länge 0 auf eine STREAM-basierte Pipe oder eine FIFO-Datei geschrieben, wird nichts gesendet und 0 zurückgegeben. Der Prozess kann I_SWROPT ioctl() verwenden, wenn Nachrichten mit Länge 0 über die Pipe oder FIFO-Datei gesendet werden sollen.

Wenn write() auf einen STREAM schreibt, werden Nachrichten der Prioritätsklasse 0 erzeugt.
Es gelten die folgenden Regeln, wenn write() auf einen STREAM schreibt, der weder eine Pipe noch eine FIFO-Datei ist:

– Wenn das Flag O_NONBLOCK nicht gesetzt ist, und der STREAM keine Daten akzeptiert (weil die STREAM-Schreibschlange wegen interner Kontrollflussbedingungen voll ist) blockiert write() solange, bis die Daten akzeptiert werden können.

– Wenn das Flag O_NONBLOCK gesetzt ist, und der STREAM keine Daten akzeptiert, schlägt write() fehl, gibt -1 zurück und setzt errno auf EAGAIN.

– Wenn das Flag O_NONBLOCK gesetzt ist, und write() bereits einen Teil des Puffers geschrieben hat, wenn eine Bedingung auftritt, unter der der STREAM keine Daten mehr akzeptiert, beendet sich write() und gibt die Anzahl der tatsächlich geschriebenen Bytes zurück.

Werden Threads verwendet, so wirkt sich die Funktion auf den Prozess oder auf einen Thread wie folgt aus:

– Bytes in Datei schreiben

  Eine Schreibanforderung für eine Pipe oder FIFO wird genauso behandelt wie solche für eine normale Datei, mit folgenden Ausnahmen:

  – Wenn das Flag O_NONBLOCK nicht gesetzt ist, kann eine Schreibanforderung den Thread blockieren, aber eine normale Beendigung liefert das Ergebnis *nbyte*.

  – Wenn das Flag O_NONBLOCK gesetzt ist, wird die Anforderung von write() verschieden behandelt, wie folgt:

    write() blockiert den Thread nicht.

  Wenn versucht wird, auf einen Dateideskriptor zu schreiben, der keine Pipe oder FIFO ist und nichtblockierendes Schreiben unterstützt, geschieht Folgendes:

  – Wenn das Flag O_NONBLOCK nicht gesetzt ist, blockiert write() den aufrufenden Thread solange, bis die Daten akzeptiert werden können.

  – EAGAIN - das Flag O_NONBLOCK ist für den Dateideskriptor gesetzt und der Thread würde durch die Schreiboperation angehalten werden.

– Ferner wird beim EPIPE-Fehler das Signal SIGPIPE nicht an den Prozess, sondern an den aufrufenden Thread gesendet.

Returnwert Anzahl der tatsächlich geschriebenen Bytes
bei erfolgreicher Beendigung. Die Byteanzahl kann nicht größer sein als *nbyte*.

0 wenn in eine normale Datei oder einen STREAM geschrieben werden soll und *nbyte* gleich 0 ist. write() hat nichts geschrieben.

-1 bei Fehler. write() hat nichts geschrieben, weil einer der folgenden Fehler vorliegt:

  – physikalischer Ein-/Ausgabefehler

  – *fildes* ist kein gültiger Dateideskriptor

  – die Datei ist nicht vorhanden

  – es besteht kein Schreibrecht für die Datei

  – der Bereich, in dem die Daten stehen, ist nicht korrekt angegeben

`errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `write()` schlägt fehl, wenn gilt:

`EAGAIN` das Flag `O_NONBLOCK` ist für den Dateideskriptor gesetzt und der Prozess würde durch die Schreiboperation angehalten werden

`EBADF` *fildes* ist kein gültiger, zum Schreiben geöffneter Dateideskriptor.

`EFBIG` Es wurde versucht, in eine Datei zu schreiben, die die maximal mögliche Dateigröße oder deren Dateigröße die Prozessgrenze überschreitet (siehe `getrlimit()` und `ulimit()`).

Erweiterung
`EAGAIN` Der Systemspeicher, der für „raw"-Ein-/Ausgabe zur Verfügung steht, ist vorübergehend nicht ausreichend,
oder es wurde versucht, in einen Datenstrom zu schreiben, der bei gesetztem `O_NDELAY` oder `O_NONBLOCK` keine Daten akzeptieren kann,
oder es wurde versucht, `{PIPE_BUF}` Bytes oder weniger auf eine Pipe oder eine FIFO zu schreiben, und es waren weniger als *nbytes* freier Speicherplatz vorhanden. ❑

Erweiterung
`EDEADLK` Die `write`-Funktion schläft und löst dadurch einen Deadlock aus.

`EFAULT` *buf* weist über den zugewiesenen Adressraum des Prozesses hinaus. ❑

`EINTR` Die Schreiboperation wurde durch ein Signal unterbrochen, und es wurden keine Daten übertragen.

Erweiterung
`EINVAL` Es wurde versucht, in einen Datenstrom zu schreiben, der mit einem Multiplexer verbunden ist. ❑

`EIO` Ein physikalischer Ein-/Ausgabefehler ist aufgetreten,
oder der Prozess ist Mitglied einer Hintergrund-Prozessgruppe und versucht, von seinem steuernden Terminal zu lesen. Der Prozess ignoriert oder blockiert das Signal `SIGTTIN`, oder die Prozessgruppe ist verwaist.

`ENOSPC` Auf dem Gerät, das die Datei enthält, war kein freier Platz mehr übrig.

Erweiterung
`ENOSR` Es wurde versucht, in einen Datenstrom zu schreiben, für den nicht genügend Speicherplatz zur Verfügung steht. ❑

`ENXIO` Eine Anforderung für ein nicht existierendes Gerät wurde gemacht,
oder die Anforderung lag außerhalb der Fähigkeiten des Geräts.

`EPIPE` Es wurde versucht, auf eine Pipe oder FIFO-Datei zu schreiben, die für keinen Prozess zum Lesen geöffnet oder die nur an einem Ende geöffnet ist. Der Prozess erhält ein `SIGPIPE`-Signal.

`ERANGE` Es wurde versucht, in einen Datenstrom mit *nbyte* außerhalb der vorgegebenen Mindest- und Höchstgrenzen zu schreiben, und der Mindestwert ist ungleich null.

`EINVAL` Der STREAM oder Multiplexer, auf den sich *fildes* bezieht, ist stromabwärts direkt oder indirekt über einen Multiplexer angeschlossen.

`ENXIO` Es wurde versucht, auf ein nicht existierendes Gerät zuzugreifen, oder die Anforderung lag außerhalb der Fähigkeiten des Geräts.

`ENXIO` Ein Hangup ist aufgetreten, während auf den Stream geschrieben wird.

`write()` schlägt auch dann fehl, wenn vor dem Aufruf eine asynchrone Fehlermeldung am STREAM-Kopf auftritt. In diesem Falle bezieht sich der Wert von `errno` nicht auf `write()`, sondern auf den vorhergehenden STREAM-Fehler.

Hinweis Um sicherzugehen, dass Ihre Angabe in *nbyte* die Größe des Puffers nicht überschreitet, sollten Sie die Funktion `sizeof()` verwenden.

BS2000
Nach jedem `write`-Aufruf sollte die tatsächlich geschriebene Byteanzahl überprüft werden:

– Wenn das Ergebnis kleiner als die Angabe in *nbyte* ist, liegt im Allgemeinen ein Fehler vor.

– Wenn das Ergebnis größer als die Angabe in *nbyte* ist, wurden Tabulatorzeichen (`\t`) in eine Textdatei geschrieben. Tabulatorzeichen werden dabei in die entsprechenden Leerzeichen umgesetzt und bei der Ergebnisanzahl berücksichtigt.

Die Daten werden nicht sofort in die externe Datei geschrieben, sondern in einem C-internen Puffer zwischengespeichert (siehe Abschnitt „Pufferung von Datenströmen“ auf Seite 77).

Bei der Ausgabe in Textdateien werden die Steuerzeichen für Zwischenraum (`\n`, `\t`, etc.) je nach Art der Textdatei in ihre entsprechende Wirkung umgesetzt (siehe Abschnitt „Zwischenraumzeichen“ auf Seite 85). ❑

Siehe auch `creat()`, `dup()`, `fcntl()`, `lseek()`, `open()`, `pipe()`, `ulimit()`, `unistd.h`.

writev - in Datei schreiben

Syntax #include <sys/uio.h>

ssize_t writev(int *fildes*, const struct iovec **iov*, size_t *nbyte*);

Beschreibung

`writev()` macht dasselbe wie `write()`, sammelt aber die Ausgabedaten der *iovcnt*-Puffer, die durch die Mitglieder der *iov*-Felder (*iov*[0], *iov*[1], ..., *iov*[*iovcnt*-1]) festgelegt sind. Es muss gelten 0 < *iovcnt* ≤ `IOV_MAX` .

Für `writev()` enthält die Struktur `iovec` folgende Elemente:

```
caddr_t   iov_base;
int       iov_len;
```

Jeder `iovec`-Eintrag gibt die Basisadresse und die Länge eines Speicherbereichs an, aus dem Daten geschrieben werden sollen. `writev()` schreibt immer einen vollständigen Bereich, bevor es zum nächsten übergeht.

Wenn *fildes* eine reguläre Datei bezeichnet und alle Elemente des Feldes *iov* den Wert 0 haben, gibt `writev()` den Wert 0 zurück und hat sonst keine Wirkung.

Wenn die Summe der `iov_len`–Werte `SSIZE_MAX` überschreitet, schlägt `writev()` fehl, und es werden keine Daten transferiert.

weitere Beschreibung: siehe `write()`.

Returnwert Anzahl der tatsächlich geschriebenen Bytes
bei Erfolg.

-1 sonst. In diesem Falle wird der Dateizeiger nicht verändert. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler siehe `write()`. Zusätzlich zu den dort angegebenen Fehlern schlägt `writev()` fehl, wenn gilt:

`EINVAL` *iovcnt* war kleiner oder gleich 0 oder größer gleich 16, oder einer der `iov_len`-Werte im *iov*-Feld war negativ, oder die Summe der `iov_len`-Werte im *iov*-Feld erzeugt einen Überlauf bei einer 32-Bit Ganzzahl.

`EINVAL` *fildes* ist einer BS2000-Datei zugeordnet..

`writev()` schlägt auch dann fehl, wenn vor dem Aufruf eine asynchrone Fehlermeldung am STREAM-Kopf auftritt. In diesem Falle bezieht sich der Wert von `errno` nicht auf `writev()`, sondern auf den vorhergehenden STREAM-Fehler.

Siehe auch `chmode()`, `creat()`, `dup()`, `fcntl()`, `getrlimit()`, `lseek()`, `open()`, `pipe()`, `ulimit()`, `limits.h`, `stropts.h`, `sys/uio.h`, `unistd.h`.

wscanf - formatiert lesen

Syntax #include <wchar.h>

int wscanf(const wchar_t **format* [, *arglist*]);

Beschreibung

Ausführliche Beschreibung siehe `fwscanf()`.

y0, y1, yn - Besselfunktionen der zweiten Art anwenden

Syntax #include <math.h>

double y0(double x);
double y1(double x);
double yn(int n, double x);

Beschreibung

`y0()`, `y1()` und `yn()` berechnen die Besselfunktionen der zweiten Art für reelle Argumente x (> 0) und die ganzzahligen Ordnungen 0, 1 bzw. n (nur bei `yn`).

Returnwert Wert der Besselfunktion für x, wenn x >0.

-`HUGE_VAL` bei Argumenten ≤ 0.
`errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler `y0()`, `y1()` und `yn()` schlagen fehl, wenn gilt:

`EDOM` Der Wert von x ist negativ.

Siehe auch `j0()`, `j1()`, `jn()`, `math.h`.

Include-Dateien

Dieses Kapitel beschreibt die Inhalte der Include-Dateien, in denen die X/Open-konformen Funktionen bzw. Makros und externen Variablen definiert bzw. deklariert werden. Die Include-Dateien sind in alphabetischer Reihenfolge beschrieben.

Include-Dateien enthalten außerdem Definitionen symbolischer Konstanten, allgemeiner Strukturen, Präprozessor-Makros und vordefinierter Datentypen. Jede Funktions- oder Variablenbeschreibung im Kapitel „Funktionen und Variablen alphabetisch (a - m)“ gibt die Include-Dateien an, die eine Anwendung einbinden muss, damit diese Funktion benutzt werden kann. Diese Include-Dateien werden nur während des Entwicklungsprozesses benötigt; bei der Ausführung müssen sie nicht vorhanden sein.

Strukturmittel

Nach der Überschrift, die den jeweiligen Namen der Include-Datei und eine stichwortartige Beschreibung des Inhalts enthält, folgen immer dieselben Unterabschnitte:

Syntax — Darstellung der Syntax, mit der die jeweilige Include-Datei in eine Anwendung eingebunden wird.

Beschreibung
Beschreibung der Inhalte der Include-Datei.

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

Siehe auch — Querverweise auf Funktionsbeschreibungen, andere Handbuchteile oder andere Handbücher.

Für die Erweiterungen gegenüber dem Standard gibt es folgende Kennzeichnungen:

BS2000
Informationen, die sich auf Erweiterungen in Zusammenhang mit dem Zugriff auf das DVS und auf C-Laufzeitsystemversionen bis V 2.1C (CRTE V1.0B) beziehen. Das Ende eines so gekennzeichneten Abschnittes wird durch die Endemarke ❑ angezeigt.

Erweiterung
Informationen über Erweiterungen des C-Laufzeitsystems V2.2 (CRTE V 2.0). Das Ende eines so gekennzeichneten Abschnittes wird durch die Endemarke ❑ angezeigt.

assert.h - Programmbedingung überprüfen

Syntax #include <assert.h>

Beschreibung

Die Include-Datei `assert.h` definiert das Makro `assert()` und bezieht sich auf das Makro `NDEBUG`, das nicht in dieser Include-Datei definiert ist. Wenn `NDEBUG` als Makroname vor der Einbindung dieser Include-Datei definiert ist, dann wird das Makro `assert()` einfach wie folgt definiert:

```
#define assert(ignore) ((void) 0)
```

Andernfalls verhält sich das Makro so, wie unter `assert()` beschrieben.

Das Makro `assert()` ist als Makro, nicht als Funktion implementiert. Wenn die Makrodefinition unterdrückt wird, um auf eine konkrete Funktion zuzugreifen, dann ist das Verhalten undefiniert.

Siehe auch `assert()`.

cpio.h - Modus-Werte für cpio-Archive

Syntax #include <cpio.h>

Beschreibung

Werte für das Feld `c_mode` im `cpio`-Archivformat. Die Werte sind wie folgt beschrieben:

Name	Wert (oktal)	Beschreibung
C_IRUSR	0000400	Leserecht für den Benutzer
C_IWUSR	0000200	Schreibrecht für den Benutzer
C_IXUSR	0000100	Ausführrecht für den Benutzer
C_IRGRP	0000040	Leserecht für die Gruppe
C_IWGRP	0000020	Schreibrecht für die Gruppe
C_IXGRP	0000010	Ausführrecht für die Gruppe
C_IROTH	0000004	Leserecht für andere
C_IWOTH	0000002	Schreibrecht für andere
C_IXOTH	0000001	Ausführrecht für andere
C_ISUID	0004000	Benutzer-ID setzen
C_ISGID	0002000	Gruppen-ID setzen
C_ISVTX	0001000	reserviert
C_ISDIR	0040000	Dateiverzeichnis
C_ISFIFO	0010000	FIFO-Datei
C_ISREG	0100000	reguläre Datei
C_ISBLK	0060000	blockorientierte Gerätedatei
C_ISCHR	0020000	zeichenorientierte Gerätedatei
C_ISCTG	0110000	reserviert
C_ISLNK	0120000	reserviert
C_ISSOCK	0140000	reserviert

Die Include-Datei definiert die symbolische Konstante:

```
MAGIC "070707"
```

ctype.h - Zeichenklassifikation

Syntax #include <ctype.h>

Beschreibung

Folgende Namen sind als Funktionen deklariert und mit Ausnahme von `isprint()`, `tolower()` und `toupper()` auch als Makros definiert:

```
int isalnum(int c);
int isalpha(int c);
int isascii(int c);
int iscntrl(int c);
int isdigit(int c);
int isgraph(int c);
int islower(int c);
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int isupper(int c);
int isxdigit(int c);
int toascii(int c);
int tolower(int c);
int toupper(int c);
```

Folgende Makronamen sind definiert:

```
int _toupper(int c);
int _tolower(int c);
```

Siehe auch `isalnum()`, `isalpha()`, `isascii()`, `iscntrl()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, `isxdigit()`, `toascii()`, `tolower()`, `toupper()`, `locale.h`.

dirent.h - Format von Dateiverzeichnis-Einträgen

Syntax #include <dirent.h>

Beschreibung

Folgender Datentyp ist durch `typedef` definiert:

`DIR`	Ein Typ, der einen Dateiverzeichnis-Strom definiert.

Die Datentypen `ino_t` und `ino64_t` werden definiert wie in `sys/types.h`.

Die Struktur `dirent` wird mit folgenden Komponenten definiert:

`ino_t d_ino`	fortlaufende Dateinummer
`char d_name[]`	Name des Eintrags

Die Struktur `dirent64` wird mit folgenden Komponenten definiert:

`ino64_t d_ino`	fortlaufende Dateinummer
`char d_name[]`	Name des Eintrags

Der `char`-Vektor `d_name` hat keine bestimmte Größe, aber die Anzahl der Bytes vor dem abschließenden Nullbyte soll `{NAME_MAX}` (beschrieben in `limits.h`) nicht überschreiten.

Die nachfolgenden Funktionen werden deklariert:

`int closedir(DIR *`*dirp*`)`

`void rewinddir(DIR *`*dirp*`)`

`DIR *opendir(const char *`*dirname*`)`

`void seekdir(DIR *`*dirp*`, long int *`*loc*`)`

`struct dirent *readdir (DIR *`*dirp*`);`

`struct dirent64 *readdir64 (DIR *`*dirp*`);`

`int readdir_r(DIR *`*dirp*`, struct dirent *`*entry*`, struct dirent **`*result*`)`

`long int telldir(DIR *`*dirp*`)`

Siehe auch `closedir()`, `opendir()`, `readdir()`, `readdir_r( )`, `rewinddir()`, `seekdir()`, `telldir()`, `sys/types.h`.

errno.h - Fehlercodes des Systems

Syntax #include <errno.h>

Beschreibung

errno.h stellt unter anderem eine Deklaration der Variablen errno zur Verfügung.

errno.h gibt den folgenden symbolischen Konstanten eindeutige, von 0 verschiedene Werte.

Die allgemeinen Beschreibungen in der Tabelle werden im Abschnitt „Fehler“ bei den jeweiligen Funktionen, bei denen sie auftreten können, präziser angegeben. In Programmen sollten nur die symbolischen Namen verwendet werden, da die konkreten Werte für die Fehlercodes implementierungsabhängig und somit nicht portabel sind.

Fehlercodes, die zwar in errno.h definiert, aber nicht vom XPG4 Version 2 vorgeschrieben sind, sind als Erweiterung gekennzeichnet.

symbolische Fehlernummer	**Fehlermeldung**	**Bedeutung**
E2BIG	Argumentliste zu lang	Die Summe der Byteanzahl, die von der Argument- und Umgebungsliste des neuen Prozessabbilds verwendet werden, ist größer als die systemabhängige Grenze {ARG_MAX}.
EACCES	Zugriff verweigert	Es wurde versucht, auf eine Art und Weise auf eine Datei zuzugreifen, die von deren Zugriffsrechten verboten wird.
EADDRINUSE (*Erweiterung*)	Adresse bereits verwendet	Es wurde versucht, eine Adresse zu verwenden, die bereits verwendet wurde, obwohl das Protokoll dies nicht zulässt.
EADDRNOTAVAIL (*Erweiterung*)	Angegebene Adresse kann nicht verwendet werden	Es wurde versucht, einen Endpunkt für eine Transportverbindung mit einer Adresse einzurichten, die nicht auf dem aktuellen Rechner liegt.
EADV (*Erweiterung*)	Anmeldefehler	Es wurde versucht, bei der gemeinsamen Nutzung ferner Dateien (RFS) ein Betriebsmittel anzumelden, das bereits angemeldet worden ist, oder RFS zu stoppen, während noch Betriebsmittel angemeldet sind, oder ein Gerät auszuhängen, während es noch angemeldet ist.
EAFNOSUPPORT (*Erweiterung*)	Adressfamilie durch Protokollfamilie nicht unterstützt	Eine mit dem angeforderten Protokoll nicht kompatible Adresse wurde verwendet.

symbolische Fehlernummer	Fehlermeldung	Bedeutung
EAGAIN	Betriebsmittel temporär nicht verfügbar	Dies ist eine temporäre Fehlerbedingung. Spätere Aufrufe derselben Funktion können normal beendet werden. Es ist z.B. möglich, dass ein Systemaufruf wegen unzureichendem Speicherplatz gescheitert ist.
EALREADY (*Erweiterung*)	Operation bereits aktiv	Es wurde versucht, eine weitere Operation auf einem nichtblockierenden Objekt zu starten, das bereits eine Operation bearbeitet.
EBADE (*Erweiterung*)	Ungültiger Austausch	
EBADF	Ungültiger Dateideskriptor	Das Argument für einen Dateideskriptor liegt außerhalb des zulässigen Bereichs, verweist nicht auf eine offene Datei oder ein Leseversuch wurde für eine nur zum Schreiben geöffnete Datei vorgenommen bzw. umgekehrt.
EBADFD (*Erweiterung*)	Dateideskriptor in ungültigem Zustand	Entweder verweist ein Dateideskriptor nicht auf eine offene Datei, oder ein Lesezugriff wurde auf eine Datei versucht, die nur zum Schreiben geöffnet wurde.
EBADMSG (*Erweiterung*)	Unzulässige Datennachricht	Bei einem Systemaufruf von z.B. `read()` auf einem STREAMS-Gerät ist etwas an den Kopf der Warteschlange gelangt, das nicht als Datennachricht verarbeitet werden kann. Was dies ist, hängt jeweils vom verwendeten Systemaufruf ab, es kann sich um Steuer- oder Dateninformationen oder um weitergereichte Dateideskriptoren handeln.
EBADR (*Erweiterung*)	Ungültiger Anmelde-Deskriptor	
EBADRQC (*Erweiterung*)	Ungültiger Anmelde-Code	
EBADSLT (*Erweiterung*)	Ungültiger Slot	
EBFONT (*Erweiterung*)	Ungültige Zeichensatzdatei	
EBUSY	Betriebsmittel nicht verfügbar	Es wurde versucht, ein Betriebsmittel zu verwenden, das zurzeit nicht verfügbar ist, weil es von einem anderen Prozess derart belegt ist, dass dies zu einem Konflikt mit der Anforderung des aktuellen Prozesses führen würde.

symbolische Fehlernummer	Fehlermeldung	Bedeutung
ECHILD	Kein Sohnprozess vorhanden	wait() oder waitpid() wurden von einem Prozess aufgerufen, der keine Sohnprozesse besitzt, oder der bereits auf alle Sohnprozesse gewartet hat.
ECHRNG (*Erweiterung*)	Kanalnummer nicht im zulässigen Bereich	
ECOMM (*Erweiterung*)	Kommunikationsfehler beim Senden	Der aktuelle Prozess wartet bei der gemeinsamen Nutzung ferner Dateien (RFS) auf eine Nachricht von einem fernen Rechner, aber die virtuelle Verbindung scheitert.
ECONNABORTED (*Erweiterung*)	Abbruch der Verbindung durch Software	Aus internen Gründen des Hostrechners bricht die Verbindung ab.
ECONNREFUSED (*Erweiterung*)	Verbindung verweigert	Es wurde versucht, eine Verbindung zu einem Dienst herzustellen, der auf dem fernen Rechner nicht aktiv ist.
ECONNRESET (*Erweiterung*)	Verbindung durch Kommunikationspartner in Grundstellung rückgesetzt	Der Kommunikationspartner hat die Verbindung abgebrochen. Normalerweise wird dies auf Grund einer Zeitüberschreitung oder auf Grund eines Neustarts des Rechners ausgelöst.
EDEADLK	Gefahr eines Deadlocks	Es wurde versucht, ein Betriebsmittel zu sperren und diese Sperre hätte einen Deadlock verursacht.
EDEADLOCK (*Erweiterung*)	Schließen der Datei durch Deadlock-Fehler	
EDESTADDRREQ (*Erweiterung*)	Zieladresse benötigt	Die Zieladresse für eine Operation am Endpunkt einer Transportverbindung war nicht angegeben.
EDMS (*Erweiterung*)	dms Fehler bei %.8s, Fehlercode = %.8s	
EDOM	Bereichsfehler	Ein Eingabe-Argument liegt außerhalb des Definitionsbereichs einer mathematischen Funktion (definiert in *ISO C-Standard*).
EEOF (*Erweiterung*)	Versuch, nach Dateiende zu lesen (cmd %.8s)	
EEXIST	Datei bereits vorhanden	Eine vorhandene Datei ist in einem ungültigen Zusammenhang angegeben worden, z.B. beim Aufruf der link()-Funktion.

symbolische Fehlernummer	Fehlermeldung	Bedeutung
EFAULT	Ungültige Adresse	Ein Programm hat auf Daten außerhalb des zulässigen Adressraums verwiesen. Es kann jedoch nicht garantiert werden, dass dieser Fehler immer zuverlässig erkannt wird. Wird dieser Fehler nicht erkannt, so kann dies dazu führen, dass ein Signal generiert und an den Prozess gesendet wird, um die Adress-Verletzung anzuzeigen.
EFBIG	Datei zu groß	Die Länge einer Datei würde die maximal zulässige Dateigröße oder {FCHR_MAX} überschreiten.
EHOSTDOWN (*Erweiterung*)	Hostrechner nicht aktiv	Eine Operation des Transportdienstgebers ist gescheitert, weil der Zielrechner nicht aktiv ist.
EHOSTUNREACH (*Erweiterung*)	Keine Verbindung zu Hostrechner bekannt	Eine Operation des Transportdienstgebers versuchte, einen nicht erreichbaren Hostrechner zu verwenden.
EIDRM	Bezeichner entfernt	Während der Interprozesskommunikation wurde ein Bezeichner aus dem System entfernt.
EIKEYOFLW (*Erweiterung*)	ISAM-KEY Überlauf	
EILSEQ	Unzulässige Bytesequenz	Ein Langzeichen wurde entdeckt, das keinem gültigen Zeichen entspricht, oder eine Bytesequenz bildet kein gültiges Langzeichen.
EINPROGRESS (*Erweiterung*)	Operation jetzt aktiv	Es wurde versucht, eine lange dauernde Operation (z.B. connect) auf einem nichtblockierenden Objekt auszuführen.
EINTR	Unterbrochener Systemaufruf	Während der Ausführung einer unterbrechbaren Funktion wurde ein Signal durch den Prozess abgefangen. Wenn die Signalbehandlungsfunktion ein normales return ausführt, dann kann die unterbrochene Funktion diese Bedingung liefern (siehe auch signal.h.).
EINVAL	Ungültiges Argument	Ein ungültiges Argument wurde verwendet, z.B. ein undefiniertes Signal für signal() oder kill().
EINVAL (*Erweiterung*)	Zugriff auf eine BS2000-Datei	Einer Funktion, die als Argument eine POSIX-Datei erwartet, wurde eine BS2000-Datei übergeben.

symbolische Fehlernummer	Fehlermeldung	Bedeutung
EIO	Ein-/Ausgabe-Fehler	Ein physikalischer Ein-/Ausgabe-Fehler ist aufgetreten. In einigen Fällen ist es möglich, dass der Fehler erst beim nächsten Systemaufruf auftritt (z.B. kann ein Fehler bei write() nach close() gemeldet werden). Jede andere, einen Fehler verursachende Operation kann dafür sorgen, dass die Fehleranzeige für EIO verloren geht.
EIORESID (*Erweiterung*)	Block nicht vollständig übertragen	
EISCONN (*Erweiterung*)	Socket bereits verbunden	Es wurde eine Verbindung angefordert oder mit sendto oder sendmsg eine Zieladresse angegeben, obwohl der Endpunkt bereits verbunden war.
EISDIR	Dateiverzeichnis statt Datei	Es wurde versucht, ein Dateiverzeichnis zum Schreiben zu öffnen.
EISNAM (*Erweiterung*)	Benannte type Datei	
EL2HLT (*Erweiterung*)	Ebene 2 beendet	
EL2NSYNC (*Erweiterung*)	Ebene 2 nicht synchronisiert	
EL3HLT (*Erweiterung*)	Ebene 3 beendet	
EL3RST (*Erweiterung*)	Reset auf Ebene 3	
ELIBACC (*Erweiterung*)	Kein Zugriff auf gemeinsam nutzbare Bibliothek möglich	Es wurde versucht, eine ausführbare Datei zu starten, die eine gemeinsam nutzbare Bibliothek benötigt, die nicht vorhanden ist, bzw. für die der Benutzer kein Zugriffsrecht hat.
ELIBBAD (*Erweiterung*)	Zugriff auf beschädigte, gemeinsam nutzbare Bibliothek	Es wurde versucht, eine ausführbare Datei zu starten, die eine gemeinsam nutzbare Bibliothek benötigt. Dies war nicht möglich, da die gemeinsam nutzbare Bibliothek wahrscheinlich beschädigt ist.
ELIBEXEC (*Erweiterung*)	Kein direktes Ausführen einer gemeinsam nutzbaren Bibliothek möglich	Es wurde versucht, eine gemeinsam nutzbare Bibliothek mit exec auszuführen.

symbolische Fehlernummer	**Fehlermeldung**	**Bedeutung**
ELIBMAX (*Erweiterung*)	Obergrenze für gemeinsam nutzbare Bibliotheken überschritten	Es wurde versucht, eine ausführbare Datei zu starten, die mehr gemeinsam nutzbare Bibliotheken benötigt als auf Grundlage der aktuellen Konfiguration des Systems zulässig sind.
ELIBSCN (*Erweiterung*)	.lib-Abschnitt in a.out beschädigt	Es wurde versucht, eine auszuführbare Datei zu starten, die eine gemeinsam nutzbare Bibliothek benötigt, und in dem .lib-Abschnitt von a.out waren fehlerhafte Daten vorhanden. Dem .lib-Abschnitt entnimmt exec die Information, die von gemeinsam nutzbaren Bibliotheken benötigt werden.
ELNRNG (*Erweiterung*)	Verweisnummer nicht im zulässigen Bereich	
ELOOP (*Erweiterung*)	Zu viele symbolische Verweise	Die Anzahl der symbolischen Verweise während der Durchquerung eines Pfades überschreitet {MAXSYMLINKS}.
EMACRO (*Erweiterung*)	Makro-Fehler: Makro %.8s gibt %.8s zurück	
EMFILE	Zu viele offene Dateien	Es wurde versucht, mehr als von {OPEN_MAX} erlaubte Dateideskriptoren für diesen Prozess zu öffnen.
EMLIN	Zu viele Verweise	Es wurde versucht, den Verweiszähler einer Datei größer als {LINK_MAX} zu setzen.
EMODE (*Erweiterung*)	Ungültiger Eröffnungszustand (cmd: %.8s)	
EMSGSIZE (*Erweiterung*)	Nachricht zu lang	Eine Nachricht, die an den Diensterbringer der Transportverbindung gesendet wurde, war größer als der interne Nachrichtenpuffer oder irgend eine andere Begrenzung im Netzwerk.
EMULTIHOP (*Erweiterung*)	Kein Überspringen möglich	Es wurde versucht, bei der gemeinsamen Nutzung ferner Dateien (RFS) auf ferne Betriebsmittel zuzugreifen, auf die man nicht direkt zugreifen kann.
ENAME (*Erweiterung*)	Ungültiger Dateiname (cmd %.8s)	
ENAMETOOLONG	Pfadname zu lang	Die Länge eines Pfadnamens überschreitet {PATH_MAX}, oder eine Pfadnamen-Komponente ist länger als {NAME_MAX.}, und {_POSIX_NO_TRUNC} ist für diese Datei aktiv.

symbolische Fehlernummer	Fehlermeldung	Bedeutung
ENAVAIL (*Erweiterung*)	Keine XENIX Semaphoren verfügbar	
ENETDOWN (*Erweiterung*)	Netzwerk nicht aktiv	Eine Operation fand ein inaktives Netzwerk vor.
ENETRESET (*Erweiterung*)	Verbindung auf Grund eines Resets abgebrochen	Der Hostrechner ist abgestürzt und wurde neu geladen.
ENETUNREACH (*Erweiterung*)	Netzwerk nicht erreichbar	Eine Operation wurde an ein nicht erreichbares Netzwerk gerichtet.
ENFILE	Zu viele Dateien im System offen	Das System hat eine vordefinierte Grenze für die gleichzeitig offenen Dateien erreicht und kann zeitweise keine weiteren Anforderungen zum Öffnen weiterer Dateien annehmen.
ENOANO (*Erweiterung*)	Kein Anode	
ENOBUFS (*Erweiterung*)	Kein Speicherplatz für Datenpuffer verfügbar	Eine Operation auf einem Endpunkt einer Transportverbindung konnte auf Grund mangelnden Speicherplatzes für Datenpuffer oder auf Grund einer vollen Warteschlange nicht ausgeführt werden.
ENOCSI (*Erweiterung*)	Keine CSI-Struktur verfügbar	
ENODATA (*Erweiterung*)	Keine Daten verfügbar	
ENODEV	Gerät unbekannt	Es wurde versucht, eine ungültige Funktion für ein Gerät anzuwenden, z.B. von einem Gerät zu lesen, auf das nur geschrieben werden kann, wie z.B. ein Drucker.
ENOENT	Datei oder Dateiverzeichnis unbekannt	Die Komponente eines angegebenen Pfadnamens existiert nicht oder der Pfadname ist die leere Zeichenkette.
ENOEXEC	Fehler im Ausführungsformat einer Datei	Es wurde versucht, eine Datei auszuführen, die zwar über die entsprechenden Zugriffrechte verfügt, aber nicht das Format hat, das bei ausführbaren Dateien benötigt wird.
ENOLCK	Keine Datensatz-Sperren verfügbar	Eine systemabhängige Grenze für die Anzahl der gleichzeitigen Datei- und Satzsperren wurde erreicht und es sind zurzeit keine weiteren mehr verfügbar.

symbolische Fehlernummer	Fehlermeldung	Bedeutung
ENOLINK (*Erweiterung*)	Virtuelle Verbindung verloren	Bei der gemeinsamen Nutzung ferner Dateien (RFS) ist die (virtuelle) Verbindung mit einem fernen Rechner verlorengegangen.
ENOMEM	Nicht genügend Speicherplatz	Das Speicherabbild des neuen Prozesses benötigt mehr Speicherplatz, als verfügbar ist.
ENOMSG	Keine Nachricht des geforderten Typs	Die Nachrichtenwarteschlange für die Interprozesskommunikation enthält keine Nachricht des geforderten Typs.
ENONET (*Erweiterung*)	Rechner nicht an das Netzwerk angeschlossen	Bei der gemeinsamen Nutzung ferner Dateien (RFS) wurde versucht, ferne Betriebsmittel anzumelden, abzumelden, ein- oder auszuhängen; aber der Rechner hat die Startprozeduren zum Anschluss des Netzwerks (noch) nicht durchgeführt.
ENOPKG (*Erweiterung*)	Paket nicht installiert	Es wurde versucht, einen Systemaufruf zu verwenden, der Teil eines nicht installierten Pakets ist.
ENOPROTOOPT (*Erweiterung*)	Protokoll nicht verfügbar	Eine falsche Version oder eine falsche Ebene wurde beim Abfragen oder beim Setzen von Protokolloptionen verwendet.
ENOSPC	Kein Platz mehr auf Gerät	Während der Ausführung der Funktion `write()` für eine normale Datei oder bei der Erweiterung eines Dateiverzeichnisses ist kein weiterer Platz mehr auf dem Gerät verfügbar.
ENOSR (*Erweiterung*)	Betriebsmittel für Datenströme unzureichend	Während eines `open` stehen entweder keine Datenstrom-Warteschlangen oder keine Datenstrom-Kopfdatenstrukturen zur Verfügung. Dies ist eine vorübergehende Fehlersituation; sobald andere Prozesse Betriebsmittel freigeben, ist der Fehler behoben.
ENOSTR (*Erweiterung*)	Gerät ist kein Datenstrom	Es wurde versucht, die Systemaufrufe `putmsg()` oder `getmsg()` auf eine Datei anzuwenden, die kein Datenstrom ist.
ENOSUBSYS (*Erweiterung*)	POSIX-Subsystem nicht bereit	
ENOSYS	Funktion nicht implementiert	Es wurde versucht, eine Funktion zu verwenden, die unter der aktuellen Implementierung nicht verfügbar ist.

symbolische Fehlernummer	Fehlermeldung	Bedeutung
ENOTBLK (*Erweiterung*)	Keine blockorientierte Datei	Eine nicht blockorientierte Datei wurde angegeben, während ein blockorientiertes Gerät benötigt wird.
ENOTCONN (*Erweiterung*)	Socket nicht verbunden	Die Anforderung nach Senden oder Empfangen von Daten konnte nicht erfüllt werden, da der Kommunikationsendpunkt nicht verbunden war und (beim Senden von datagrams) keine Adresse angegeben wurde.
ENOTDIR	Kein Dateiverzeichnis	Eine Komponente des angegebenen Pfadnamens existiert, ist aber kein Dateiverzeichnis, obwohl ein Dateiverzeichnis erwartet wurde.
ENOTEMPTY	Dateiverzeichnis nicht leer	Es wurde ein Dateiverzeichnis angegeben, das außer . und .. noch weitere Einträge enthält, obwohl ein leeres Dateiverzeichnis erwartet wurde.
ENOTNAM (*Erweiterung*)	Keine XENIX benannte type Datei	
ENOTSOCK (*Erweiterung*)	Socket-Operation auf nicht-Socket	
ENOTTY	Ungültige Ein-/Ausgabe-Steueroperation	Eine Funktion zur Steuerung einer Datei oder einer Gerätedatei wurde aufgerufen, für die diese Operation nicht erlaubt ist.
ENOTUNIQ (*Erweiterung*)	Name im Netzwerk nicht eindeutig	
ENXIO	Gerät oder Adresse nicht verfügbar	Die Ein-/Ausgabe auf eine Gerätedatei erfolgte auf ein Gerät, das nicht existiert oder die Anforderung liegt außerhalb der Möglichkeiten dieses Geräts. Dieser Fehler kann zum Beispiel auch dann auftreten, wenn ein Bandlaufwerk nicht eingeschaltet ist.
EOPNOTSUPP (*Erweiterung*)	Operation am Endpunkt der Transportverbindung nicht unterstützt	Es wurde versucht, z.B. eine Verbindung auf einem Endpunkt einer datagram-Transportverbindung zu akzeptieren.
EOPR (*Erweiterung*)	Illegale Operation (cmd %.8s)	
EOVERFLOW (*Erweiterung*)	Wert für den definierten Datentyp zu groß	
EPERM	Nicht Systemverwalter	Es wurde versucht, eine Operation auszuführen, die Prozessen mit Sonderrechten oder dem Eigentümer einer Datei oder eines anderen Betriebsmittels vorbehalten ist.

symbolische Fehlernummer	Fehlermeldung	Bedeutung
EPFNOSUPPORT (*Erweiterung*)	Protokollfamilie nicht unterstützt	Die Protokollfamilie ist im aktuellen System nicht konfiguriert, oder es existiert dafür keine Implementierung. Der Fehler tritt bei Internet-Protokollen auf.
EPIPE	Pipe abgebrochen	Es wurde versucht, auf eine Pipe oder FIFO zu schreiben, von denen kein Prozess Daten liest.
EPROTO (*Erweiterung*)	Protokollfehler	Dieser Fehler ist gerätespezifisch, steht im Allgemeinen jedoch nicht in Zusammenhang mit einem Hardware-Ausfall.
EPROTONOSUPPORT (*Erweiterung*)	Protokoll nicht unterstützt	Das Protokoll ist im aktuellen System nicht konfiguriert, oder es existiert dafür keine Unterstützung.
EPROTOTYPE (*Erweiterung*)	Ungültiger Protokolltyp für Socket	Es wurde ein Protokoll angegeben, das die Semantik des geforderten Socket-Typs nicht unterstützt.
ERANGE	Ergebnis zu groß oder zu klein	Das Ergebnis einer Funktion ist zu groß oder zu klein für den verfügbaren Speicherplatz (definiert im *ISO C-Standard*).
EREMCHG (*Erweiterung*)	Adresse für fernen Zugriff geändert	
EREMOTE (*Erweiterung*)	Betriebsmittel nicht lokal	Dieser Fehler kann bei gemeinsamer Nutzung ferner Dateien (RFS) die folgenden Ursachen haben: – Es wurde versucht, ein Betriebsmittel anzumelden, das sich nicht auf dem lokalen Rechner befindet. – Es wurde das Ein-/Aushängen eines Gerätes versucht, das sich auf einem fernen Rechner befindet. – Es wurde das Ein-/Aushängen eines Pfadnamens versucht, der sich auf einem fernen Rechner befindet.
EREMOTEIO (*Erweiterung*)	Ferner Ein-/Ausgabefehler	
EREPL (*Erweiterung*)	Nur bei der Eröffnung auftretender Fehler	
ERESTART (*Erweiterung*)	Erneut durchführbarer Systemaufruf	

symbolische Fehlernummer	Fehlermeldung	Bedeutung
EROFS	Dateisystem nur zum Lesen	Es wurde versucht, eine Datei oder ein Dateiverzeichnis in einem Dateisystem mit Schreibschutz zu ändern.
ESHUTDOWN (*Erweiterung*)	Kein Senden nach Schließen von Socket	Das geforderte Senden von Daten war nicht möglich, da der Kommunikationsendpunkt bereits geschlossen war.
ESOCKTNOSUPPORT (*Erweiterung*)	Socket-Typ nicht unterstützt	Für den angegebenen Socket-Typ ist das aktuelle System nicht konfiguriert, oder es existiert dafür keine Unterstützung.
ESPIPE	Ungültige Positionieranweisung	Es wurde versucht, auf die Dateiposition einer Pipe oder FIFO zuzugreifen.
ESRCH	Prozess unbekannt	Zur angegebenen Prozessnummer kann kein entsprechender Prozess gefunden werden.
ESRMNT (*Erweiterung*)	Srmount-Fehler	Es wurde versucht, bei der gemeinsamen Nutzung ferner Dateien (RFS) RFS anzuhalten, während noch Betriebsmittel auf fernen Rechnern eingehängt sind, oder beim Anmelden eines Betriebsmittels eine Benutzerliste zu verwenden, die einen der fernen Rechner nicht enthält, der derzeit das Betriebsmittel eingehängt hat.
ESSNOTAVAIL (*Erweiterung*)	Subsystem nicht verfügbar	
ESTALE (*Erweiterung*)	Veraltete NFS Dateibehandlung	
ERESTART (*Erweiterung*)	Unterbrochener Systemaufruf sollte erneut gestartet werden	
ESTRPIPE (*Erweiterung*)	Fehler in Streams-Pipe (nicht extern sichtbar)	
ETIME (*Erweiterung*)	Zeituhr abgelaufen	Die für `ioctl()` gesetzte Zeituhr ist abgelaufen. Die Ursache für diesen Fehler ist gerätespezifisch und kann entweder einen Hardware- oder einen Software-Fehler anzeigen oder möglicherweise auch einen Zeitwert, der für die bestimmte Funktion zu klein ist. Der Status der `ioctl()`-Funktion ist unbestimmt.

symbolische Fehlernummer	Fehlermeldung	Bedeutung
ETIMEDOUT (*Erweiterung*)	Zeitüberschreitung für Verbindung	Ein Verbindungsaufbau oder eine Sendeanforderung scheiterte, weil der Kommunikationspartner nicht ordnungsgemäß innerhalb der vorgegebenen Zeit antwortete. Das Zeitintervall ist abhängig vom Kommunikationsprotokoll.
ETOOMANYREFS (*Erweiterung*)	Zu viele Referenzen: Verbinden nicht möglich	
ETXTBSY	Textdatei aktiv	Es wurde versucht, eine reine Prozedur auszuführen, die aktuell zum Schreiben geöffnet ist oder, eine reine Prozedur zu schreiben, die aktuell ausgeführt wird.
EUCLEAN (*Erweiterung*)	Struktur benötigt Aufräumen	
EUNATCH (*Erweiterung*)	Protokolltreiber nicht eingerichtet	
EUSERS (*Erweiterung*)	Zu viele Benutzer (für UFS)	
EWOULDBLOCK (*Erweiterung*)	siehe EAGAIN	
EXDEV	Ungültiger Verweis	Es wurde versucht, einen Verweis auf eine Datei in einem anderen Dateisystem einzurichten.
EXFULL (*Erweiterung*)	Austausch ausgeschöpft	

Siehe auch errno, perror(), strerror(), Abschnitt „Fehlerbehandlung“ auf Seite 130.

fcntl.h - Optionen für die Steuerung von Dateien

Syntax #include <fcntl.h>

Beschreibung

Die Include-Datei `fcntl.h` definiert die folgenden Anfragen und Parameter zum Gebrauch durch die Funktionen `fcntl()` und `open()`.

Eindeutige Werte für die bei `fcntl()` angegebenen Kommandos sind:

`F_DUPFD`	Dateideskriptor duplizieren
`F_GETFD`	Dateideskriptor-Flags holen
`F_SETFD`	Dateideskriptor-Flags setzen
`F_GETFL`	Dateistatus-Flags holen
`F_SETFL`	Dateistatus-Flags setzen
`F_GETLK`	Satzsperr-Informationen holen
`F_GETLK64`	Satzsperr-Informationen holen unter Verwendung der flock64-Struktur
`F_SETLK`	Satzsperr-Informationen setzen
`F_SETLK64`	Satzsperr-Informationen setzen unter Verwendung der flock64-Struktur
`F_SETLKW`	Satzsperr-Informationen setzen; warten, falls blockiert
`F_SETLKW64`	Satzsperr-Informationen setzen; warten, falls blockiert (unter Verwendung der flock64-Struktur)

Dateistatus-Flag für `open()` und `fcntl()`:

`O_LARGEFILE`	Maximaler Offset im Open File Deskriptor ist der maximale Wert, der durch ein off64_t-Objekt dargestellt werden kann.

Dateideskriptor-Flags für `fcntl()`:

`FD_CLOEXEC`	Dateideskriptor bei Ausführen einer exec-Funktion (siehe `exec()`) schließen

Eindeutige Werte für `l_type`, um mit `fcntl()` Datensätze zu sperren:

`F_RDLCK`	gemeinsam benutzte Sperre oder Lesesperre
`F_UNLCK`	entsperren
`F_WRLCK`	Exklusiv- oder Schreibsperre

Die Werte für `l_whence`, `SEEK_SET`, `SEEK_CUR` und `SEEK_END` sind definiert wie unter `unistd.h` beschrieben.

Wertegruppen für das Argument *oflag* von `open()` sind bitweise verschieden:

`O_CREAT`	Datei erstellen, falls sie nicht existiert
`O_EXCL`	exklusive Benutzung
`O_NOCTTY`	kein steuerndes Terminal zuweisen
`O_TRUNC`	Dateilänge abschneiden

Dateistatus-Marke für `open()` und `fcntl()`:

O_APPEND	Appendmodus setzen
O_NONBLOCK	Entsperrmodus (POSIX)
O_SYNC	synchrones Schreiben

Maske für die Anwendung von Dateizugriffsmodus:

`O_ACCMODE`	Maske für Dateizugriffsmodus

Dateizugriffsverfahren für `open()` und `fcntl()`:

`O_RDONLY`	Öffnen zum Lesen
`O_RDWR`	Öffnen zum Lesen und Schreiben
`O_WRONLY`	Öffnen zum Schreiben

Die symbolischen Namen der Werte für `mode_t` sind definiert wie unter `sys/types.h` beschrieben.

Die Struktur `flock` beschreibt eine Dateisperre. Sie enthält die folgenden Komponenten:

```
short   l_type;    /* Sperrtyp */
short   l_whence;  /* Offsettyp */
off_t   l_start;   /* Relativer Offset in Bytes */
off_t   l_len;     /* Größe; wenn 0, dann bis EOF (Dateiende) */
long    l_sysid;   /* zurückgegeben mit F_GETLK */
pid_t   l_pid;     /* Prozess-ID des Prozesses, der die Sperre hält; */
                   /* zurückgegeben mit F_GETKL */
```

Die Struktur `flock64` ist die die 64-Bit-Version von `flock` und beschreibt eine Dateisperre. Sie enthält die folgenden Komponenten:

```
short   l_type;    /* Sperrtyp */
short   l_whence;  /* Offsettyp */
off64_t l_start;   /* Relativer Offset in Bytes */
off64_t l_len;     /* Größe; wenn 0, dann bis EOF (Dateiende) */
long    l_sysid;   /* zurückgegeben mit F_GETLK */
pid_t   l_pid;     /* Prozess-ID des Prozesses, der die Sperre hält; */
                   /* zurückgegeben mit F_GETKL */
```

Die Datentypen `mode_t`, `off_t` und `pid_t` sind definiert wie unter `sys/stat.h` beschrieben.

Die folgenden Namen sind als Funktionen deklariert:

```
int creat(const *path, mode_t mode);
int creat64(const char *path, mode_t mode)
int fcntl(int fildes, int cmd, ...);
int open(const char *path, int oflag, ...);
int open64(const char *path, int oflag, ...);
```

float.h - Typen für Gleitpunktzahlen

Syntax #include <float.h>

Beschreibung

Die Merkmale von Gleitpunktzahl-Typen sind definiert in den Begriffen eines Modells, das die Darstellung von Gleitpunktzahlen beschreibt, und Werten, die Informationen über die Gleitpunkt-Arithmetik einer Implementation bereitstellen.

Die folgenden Parameter werden benutzt, um das Modell für jeden Gleitpunktzahl-Typ zu definieren:

- s Vorzeichen (±1)
- b Basis oder Wurzel der Exponentialdarstellung (eine ganze Zahl > 1)
- e Exponent (eine ganze Zahl zwischen einem Minimum e_{min} und einem Maximum e_{max})
- p Genauigkeit (die Anzahl der signifikanten Stellen zur Basis b)
- f_k nicht-negative ganze Zahlen kleiner b (die signifikanten Stellen)

Eine normalisierte Gleitpunktzahl x*(f1 > 0 wenn x ≠ 0)* wird nach dem folgenden Modell definiert:

$$x = s \times b^{e} \times \sum_{k=1}^{p} f_k \times b^{-k}, e_{min} \le e \le e_{min}$$

`FLT_RADIX` ist ein konstanter Ausdruck, passend für den Gebrauch in den `#if`-Anweisungen des Präprozessors. Außer `FLT_RADIX` und `FLT_ROUNDS` haben alle Ausdrücke eigene Namen für alle drei Gleitpunkt-Typen. Die Gleitpunkt-Modell-Darstellung wird für alle Makronamen außer `FLT_ROUNDS` bereitgehalten.

Der Rundungsmodus für Gleitpunkt-Addition wird durch den Wert von `FLT_ROUNDS` bestimmt:

- -1 unbestimmbar
- 0 gegen Null
- 1 zur nächsten Zahl
- 2 gegen positiv unendlich
- 3 gegen negativ unendlich

Alle anderen Werte für `FLT_ROUNDS` bestimmen ein implementationsabhängiges Rundungsverhalten. In `float.h` ist `FLT_ROUNDS` mit dem Wert 0 definiert.

Die Makronamen in der folgenden Liste werden als Ausdrücke definiert.

Name	Wert	Beschreibung
FLT_RADIX	16	Wurzel der Exponentialdarstellung b
FLT_MANT_DIG	6	Anzahl der signifikanten Stellen zur Basis FLT_RADIX, p
DBL_MANT_DIG	14	
LDBL_MANT_DIG	ANSI 28 KR 14	
FLT_DIG	6	Anzahl der Dezimalstellen q, die für eine beliebige Gleitpunktzahl mit q Dezimalstellen umgewandelt werden kann in eine Gleitpunktzahl mit p Stellen zur Basis b und wieder zurück ohne Änderung der q Dezimalstellen. $[(p-1)\times\log_{10}b]+k$ Dabei gilt: k = 1, wenn b eine ganzzahlige Potenz von 10 ist, sonst k = 0.
DBL_DIG	15	
LDBL_DIG	ANSI 32 KR 15	
FLT_MIN_EXP	-64	Kleinste negative ganze Zahl, mit der FLT_RADIX potenziert mit dieser Zahl minus 1 eine normalisierte Gleitpunktzahl ergibt. e_{min}
DBL_MIN_EXP	-64	
LDBL_MIN_EXP	-64	
FLT_MIN_10_EXP	-79	Kleinste negative ganze Zahl, mit der 10 potenziert mit dieser Zahl minus 1 eine Zahl im Bereich der normalisierten Gleitpunktzahlen ergibt. $\left\lceil \log_{10} b^{e_{min}^{-1}} \right\rceil$
DBL_MIN_10_EXP	-79	
LDBL_MIN_10_EXP	-79	
FLT_MAX_EXP	63	Größte ganze Zahl, mit der FLT_RADIX potenziert mit dieser Zahl minus 1 eine darstellbare endliche Gleitpunktzahl ergibt. e_{max}
DBL_MAX_EXP	63	
LDBL_MAX_EXP	63	
FLT_MAX_10_EXP	75	Größte ganze Zahl, mit der 10 potenziert mit dieser Zahl minus 1 eine Zahl im Bereich der darstellbaren endlichen Gleitpunktzahlen ergibt. $[\log_{10}((1-b^{-p})\times b^{e_{max}})]$
DBL_MAX_10_EXP	75	
LDBL_MAX_10_EXP	75	

Die hier angegebenen besonders großen oder kleinen Werte sind angenäherte Auflösungen der in `float.h` definierten Ausdrücke.

Name	Wert	Beschreibung
`FLT_MAX`	7,2370e75	Größte darstellbare endliche Gleitpunktzahl $(1 - b^{-p}) \times b^{e_{max}}$
`DBL_MAX`	7,2370e75	
`LDBL_MAX`	7,2370e75	

Die hier angegebenen besonders großen oder kleinen Werte sind angenäherte Auflösungen der in `float.h` definierten Ausdrücke.

Name	Wert	Beschreibung
`FLT_EPSILON`	9,53674e-7	Die Differenz zwischen 1,0 und dem kleinsten im entsprechenden Gleitpunkt-Typ darstellbaren Wert größer als 1,0 $b^{(1-p)}$
`DBL_EPSILON`	2,22045e-16	
`LDBL_EPSILON`	3,08149e-33 ANSI 9 KR 16	
`FLT_MIN`	5,3976054e-79	Angenäherte kleinste mit `float` darstellbare normalisierte positive Gleitpunktzahl $b^{(e_{min}-1)}$
`DBL_MIN`	5,39760534693402 8e-79	Kleinste mit `double` und `long double` darstellbare positive Gleitpunktzahl $b^{(e_{min}-1)}$
`LDBL_MIN`	5,39760534693402 78909e-79	

fmtmsg.h - Struktur der Meldungsanzeige

Syntax #include <fmtmsg.h>

Beschreibung

fmtmsg.h definiert folgende Makros, die expandiert konstante, ganzzahlige Ausdrücke ergeben:

MM_HARD	Ursprung eines Zustandes ist die Hardware.
MM_SOFT	Ursprung eines Zustandes ist die Software.
MM_FIRM	Ursprung eines Zustandes ist die Firmware.
MM_APPL	Meldungsursprung in einer Anwendung.
MM_UTIL	Meldungsursprung in einem Hilfsprogramm.
MM_OPSYS	Meldungsursprung im Betriebssystem.
MM_RECOVER	Stabilisierbarer Fehler.
MM_NRECOV	Nicht stabilisierbarer Fehler.
MM_HALT	Schwerwiegender Fehler, der die Bearbeitung der Anwendung anhält.
MM_ERROR	Anwendung hat einen nicht schwer wiegenden Fehler erkannt.
MM_WARNING	Anwendung hat einen ungewöhnlichen Zustand erkannt (der evtl. ein Fehler sein könnte).
MM_INFO	Information über einen Zustand, der keinen Fehler darstellt.
MM_NOSEV	Für die Meldung existiert keine Warnstufe.
MM_PRINT	Meldung auf Standard-Fehlerausgabe ausgeben.
MM_CONSOLE	Meldung auf Systemkonsole ausgeben.

Die folgende Tabelle zeigt die Nullwerte und Bezeichner für die Argumente von fmtmsg():

Argument	Typ	Nullwert	Bezeichner
label	char*	(char*)0	MM_NULLLBL
severity	int	0	MM_NULLSEV
class	long int	0L	MM_NULLMC
text	char*	(char*)0	MM_NULLTXT
action	char*	(char*)0	MM_NULLACT
tag	char*	(char*)0	MM_NULLTAG

fmtmsg.h definiert folgende Makros als Rückgabewerte von fmtmsg():

MM_OK	Die Funktion wurde erfolgreich ausgeführt.
MM_NOTOK	Die Funktion wurde nicht erfolgreich ausgeführt.
MM_NOMSG	Die Funktion konnte keine Meldung auf der Standard-Fehlerausgabe generieren, wurde aber ansonsten erfolgreich ausgeführt.

MM_NOCON Die Funktion konnte keine Meldung auf der Systemkonsole generieren, wurde aber ansonsten erfolgreich ausgeführt.

Folgende Funktion ist definiert:

```
int fmtmsg(long classification, const char *label, int severity,
           const char *text, const char *action, const char *tag);
```

Siehe auch fmtmsg().

ftw.h - Durchsuchen eines Dateibaums

Syntax #include <ftw.h>

Beschreibung

Die Include-Datei ftw.h definiert Codes für das dritte Argument der benutzerdefinierten Funktion, die als zweites Argument an ftw() übergeben wird:

FTW_F Datei

FTW_D Dateiverzeichnis

FTW_DNR Dateiverzeichnis ohne Leserecht

FTW_NS Unbekannter Typ, stat() fehlgeschlagen

Deklariert den folgenden Namen als Funktion:

```
int ftw(const char *path,
   int (*fn) (const char *, const struct stat *, int), int ndirs);

int ftw64(const char *path,
   int (*fn) (const char *, const struct stat64 *, int), int ndirs);

int nftw64(const char *path,
   int (*fn) (const char *, const struct stat64 *, int, struct FTW *),
   int depth, int flags);
```

ftw.h definiert die Struktur stat sowie die symbolischen Namen für st_mode und die Makros für den Dateityp-Test wie in sys/stat.h beschrieben.

Hinweis Einbeziehen von ftw.h kann alle Symbole aus sys/stat.h sichtbar machen.

Siehe auch ftw(), sys/stat.h.

grp.h - Gruppenstruktur

Syntax #include <grp.h>

Beschreibung

Die Include-Datei `grp.h` deklariert die Struktur `struct group`, welche die folgenden Komponenten enthält:

`char`	`*gr_name`	Gruppenname
`gid_t`	`gr_gid`	Gruppennummer
`char`	`**gr_mem`	Zeiger auf einen mit dem Nullzeiger abgeschlossenen Vektor von Zeiger auf die Mitgliedsnamen

Der Datentyp `gid_t` ist definiert wie unter `sys/types.h` beschrieben.

Die folgenden Namen sind als Funktionen deklariert:

`struct group *getgrgid(gid_t` ***gid***`);`

`struct group *getgrgid_r(gid_t` ***gid***, struct group *_gr_, char *_name2_, size_t _siz_, struct group **_grp_,);

`struct group *getgrnam(const char *`***name***`);`

`struct group *getgrnam_r(const char *`***name***, struct group *_gr_, char *_name2_, size_t _siz_, struct group **_grp_`);`

Siehe auch `getgrgid()`, `getgrgid()_r`,`getgrnam()`, `getgrnam()_r`, `sys/types.h`.

iconv.h - Zeichensatz-Umwandlung

Syntax #include <iconv.h>

Beschreibung

Folgender Datentyp ist durch `typedef` definiert:

`iconv_t` Gibt die Umwandlung von einem in einen anderen Zeichensatz an.

Folgende Namen sind als Funktionen deklariert:

```
iconv_t iconv_open(const char *tocode, const char *fromcode);

size_t iconv(iconv_t cd, char **inbuf, size_t *inbytesleft, char **outbuf,
             size_t *outbytesleft);

int iconv_close(iconv_t cd);
```

Siehe auch `iconv_open()`, `iconv()`, `iconv_close()`.

iso646.h - Alternative Schreibweisen für Operatoren

Syntax #include <iso646.h>

Beschreibung

Die Include-Datei iso646.h enthält die folgenden 11 Makros, die zu den jeweils dahinterstehenden Schreibweisen expandiert werden und damit alternative Schreibweisen für die Operatoren darstellen:

and	&&	compl	~	or_eq	|=
and_eq	&=	not	!	xor	^
bitand	&	not_eq	!=	xor_eq	^=
bitor	|	or	||		

langinfo.h - Konstanten für Sprachinformation

Syntax #include <langinfo.h>

Beschreibung

`langinfo.h` enthält die Konstanten, die verwendet werden, um `langinfo`-Daten zu identifizieren (siehe auch `nl_langinfo()`). Der Typ dieser Konstanten ist in `nl_types.h` gegeben.

Folgende Konstanten sind auf allen X/Open-kompatiblen Systemen definiert. Die Einträge in der Spalte „Kategorie“ geben an, zu welcher Kategorie von `setlocale()` jeder Eintrag definiert ist.

Konstante	Kategorie	Bedeutung
CODESET	LC_CTYPE	Zeichensatzname
D_T_FMT	LC_TIME	Zeichenkette zur Formatierung von Datum und Zeit
D_FMT	LC_TIME	Datumsformat-Zeichenkette
T_FMT	LC_TIME	Zeitformat-Zeichenkette
T_FMT_AMPM	LC_TIME	Formatzeichenkette für Vormittag (a.m.) oder Nachmittag (p.m.)
AM_STR	LC_TIME	Kennzeichen für Vormittag (a.m.)
PM_STR	LC_TIME	Kennzeichen für Nachmittag (p.m.)
DAY_1	LC_TIME	Name des ersten Wochentags (z.B. Sonntag)
DAY_2	LC_TIME	Name des zweiten Wochentags (z.B. Montag)
DAY_3	LC_TIME	Name des dritten Wochentags (z.B. Dienstag)
DAY_4	LC_TIME	Name des vierten Wochentags (z.B. Mittwoch)
DAY_5	LC_TIME	Name des fünften Wochentags (z.B. Donnerstag)
DAY_6	LC_TIME	Name des sechsten Wochentags (z.B. Freitag)
DAY_7	LC_TIME	Name des siebten Wochentags (z.B. Samstag)
ABDAY_1	LC_TIME	abgekürzter Name des ersten Wochentags
ABDAY_2	LC_TIME	abgekürzter Name des zweiten Wochentags
ABDAY_3	LC_TIME	abgekürzter Name des dritten Wochentags
ABDAY_4	LC_TIME	abgekürzter Name des vierten Wochentags
ABDAY_5	LC_TIME	abgekürzter Name des fünften Wochentags
ABDAY_6	LC_TIME	abgekürzter Name des sechsten Wochentags
ABDAY_7	LC_TIME	abgekürzter Name des siebten Wochentags
MON_1	LC_TIME	Name des ersten Monats
MON_2	LC_TIME	Name des zweiten Monats

Konstante	Kategorie	Bedeutung
MON_3	LC_TIME	Name des dritten Monats
MON_4	LC_TIME	Name des vierten Monats
MON_5	LC_TIME	Name des fünften Monats
MON_6	LC_TIME	Name des sechsten Monats
MON_7	LC_TIME	Name des siebten Monats
MON_8	LC_TIME	Name des achten Monats
MON_9	LC_TIME	Name des neunten Monats
MON_10	LC_TIME	Name des zehnten Monats
MON_11	LC_TIME	Name des elften Monats
MON_12	LC_TIME	Name des zwölften Monats
ABMON_1	LC_TIME	abgekürzter Name des ersten Monats
ABMON_2	LC_TIME	abgekürzter Name des zweiten Monats
ABMON_3	LC_TIME	abgekürzter Name des dritten Monats
ABMON_4	LC_TIME	abgekürzter Name des vierten Monats
ABMON_5	LC_TIME	abgekürzter Name des fünften Monats
ABMON_6	LC_TIME	abgekürzter Name des sechsten Monats
ABMON_7	LC_TIME	abgekürzter Name des siebten Monats
ABMON_8	LC_TIME	abgekürzter Name des achten Monats
ABMON_9	LC_TIME	abgekürzter Name des neunten Monats
ABMON_10	LC_TIME	abgekürzter Name des zehnten Monats
ABMON_11	LC_TIME	abgekürzter Name des elften Monats
ABMON_12	LC_TIME	abgekürzter Name des zwölften Monats
ERA	LC_TIME	Beschreibungsteile für Ära
ERA_D_FMT	LC_TIME	Formatzeichenkette für Äradatum
ERA_D_T_FMT	LC_TIME	Formatzeichenkette für Äradatum und -uhrzeit
ERA_T_FMT	LC_TIME	Formatzeichenkette für Äruhrzeit
ALT_DIGITS	LC_TIME	alternative Symbole für Ziffern
RADIXCHAR	LC_NUMERIC	Dezimalpunkt
THOUSEP	LC_NUMERIC	Tausender-Trennzeichen
YESEXPR	LC_MESSAGES	Ausdruck für positive Anwort
NOEXPR	LC_MESSAGES	Ausdruck für negative Anwort
YESSTR	LC_MESSAGES	Positive Antwort für Ja/Nein-Abfragen Wird zukünftig vom X/Open-Standard nicht mehr unterstützt.

Konstante	Kategorie	Bedeutung
NOSTR	LC_MESSAGES	Negative Antwort für Ja/Nein-Abfragen Wird zukünftig vom X/Open-Standard nicht mehr unterstützt.
CRNCYSTR	LC_MONETARY	Währungssymbol; wenn das Symbol vor dem Wert erscheinen soll, dann muss das Zeichen '-' vorangestellt werden; wenn es nach dem Wert erscheinen soll, dann muss das Zeichen '+' voranstehen; wenn es den Dezimalpunkt ersetzen soll, dann muss das Zeichen '.' vorangestellt werden.

Deklariert den folgenden Namen als Funktion:

```
char *nl_langinfo(nl_item);
```

Hinweis Wann immer dies möglich ist, sollten Benutzer die Funktionen verwenden, die kompatibel zu denen in ISO C-Standard sind, um auf `langinfo`–Daten zuzugreifen. Insbesondere sollte die Funktion `strftime()` verwendet werden, um auf die Datums- und Zeitinformationen der Kategorie `LC_TIME` zuzugreifen. Die `localeconv`-Funktion sollte verwendet werden, um auf Informationen in Verbindung mit `RADIXCHAR`, `THOUSEP` und `CRNCYSTR` zuzugreifen.

Siehe auch `nl_langinfo()`, `localeconv()`, `strfmon()`, `strftime()`, Abschnitt „Lokalität“ auf Seite 52.

libgen.h - Funktionsdefinitionen für Mustervergleich

Syntax #include <libgen.h>

Beschreibung

`libgen.h` definiert folgende externe Variable:

```
extern char* _ _loc1
```

(WIRD DEMNÄCHST NICHT MEHR UNTERSTÜTZT)
Die Variable wird von `regex()` verwendet, um die Adresse eines Musters anzugeben.

Folgende Funktionen sind definiert:

```
char  *regcmp(const char *string1, ... );
char  *basename (const char *path);
char  *dirname (const char *path);
char  *regex(const char *re1, const char *subject, ... );
```

Siehe auch `regcmp()`.

limits.h - implementierungsabhängige Konstanten

Syntax #include <limits.h>

Beschreibung

`limits.h` definiert verschiedene symbolische Namen. Die unten angeführten Tabellen gruppieren diese Namen in verschiedene Kategorien. Die Namen repräsentieren verschiedene Grenzwerte von Betriebsmitteln, die das System Anwendungen zur Verfügung stellt.

Symbolische Konstantennamen, die mit `_POSIX` beginnen, werden unter `unistd.h` beschrieben.

Anwendungen sollten sich bei Grenzwerten nicht auf einen konkreten Wert verlassen. Um ein Maximum an Portabilität zu erreichen, sollte eine Anwendung bei den Betriebsmitteln möglichst mit dem Wert auskommen, der in der Spalte „kleinster akzeptabler Wert“ angegeben ist.
Falls eine Anwendung aber die Betriebsmittel vollständig ausschöpfen will, die in der jeweiligen Implementierung zur Verfügung stehen, sollte der in `limits.h` definierte symbolische Name benutzt werden, der den jeweiligen Maximalwert der Implementierung enthält (siehe Spalte 1 der folgenden Tabelle). Dabei ist zu beachten, dass viele der in `limits.h` aufgeführten Grenzwerte variabel sind, und dass zur Laufzeit der tatsächliche Wert von dem Wert abweichen kann, der in `limits.h` vorgegeben ist. Dies hat folgende Gründe:

– Der Grenzwert hängt von einem Pfadnamen ab.
– Der Grenzwert unterscheidet sich auf Übersetzungszeit- und Laufzeit-Rechnern.

Aus diesen Gründen kann eine Anwendung `fpathconf()`, `pathconf()` und `sysconf()` verwenden, um den aktuellen Wert einer Grenze zur Laufzeit zu ermitteln.

Die Einträge in der Liste, die auf `_MIN` enden, stellen die kleinsten negativen Werte der mathematischen Typen dar, die garantiert dargestellt werden können. Kleinere negative Werte können unter einigen Systemen unterstützt werden, wie in der jeweiligen Include-Datei `limits.h` des Systems angegeben, aber Anwendungen, die solche Zahlen benötigen, sind nicht unbedingt auf alle Systeme portabel.

Das Symbol '*' in der Spalte „kleinster akzeptabler Wert“ gibt an, dass es keinen garantierten Wert für alle X/Open-kompatiblen Systeme gibt.

Zur Laufzeit unveränderbare Werte

Name	Beschreibung	kleinster akzeptabler Wert
ARG_MAX	Maximale Länge des Arguments der exec-Funktionen, einschließlich der Umgebungsdaten	_POSIX_ARG_MAX
CHILD_MAX	Maximale Anzahl der Prozesse je Benutzernummer	25
OPEN_MAX	Maximalzahl der gleichzeitig offenen Dateien je Prozess	20
PASS_MAX	Maximalzahl signifikanter Zeichen in einem Kennwort, ohne abschließendes Nullbyte (Wird zukünftig vom X/Open-Standard nicht mehr unterstützt.)	8
PTHREAD_DESTRUCTOR_ITERATIONS		
	Maximalzahl an Versuchen, die unternommen wurden, um die thread-spezifische Datenwerte eines Threads beim Thread-Ende zu löschen.	_POSIX_THREAD_DESTRUCTOR_ITERATIONS
PTHREAD_KEYS_MAX		
	Maximalzahl der Datenschlüssel, die von einem Prozess erzeugt werden können.	_POSIX_THREAD_KEYS_MAX
PTHREAD_STACK_MIN		
	Minimale Stack-Speicher-Größe (in Byte).	0
PTHREAD_THREADS_MAX		
	Maximalzahl der Threads, die pro Prozess erzeugt werden können.	_POSIX_THREAD_THREADS_MAX
STREAM_MAX	Maximalzahl der Datenströme, die ein Prozess gleichzeitig offen haben kann. Entspricht dem Wert von FOPEN_MAX (siehe stdio.h)	_POSIX_STREAM_MAX
TZNAME_MAX	Maximalzahl der Bytes, die von einem Zeitzonennamen unterstützt wird (nicht von der Variablen TZ)	_POSIX_TZNAME_BUF

Werte für Pfadnamen-Variablen

Name	Beschreibung	kleinster akzeptabler Wert
LINK_MAX	Maximale Anzahl von Verweisen auf eine einzelne Datei	_POSIX_LINK_MAX
MAX_CANON	Maximale Anzahl von Bytes in einer Eingabezeile der Datensichtstation bei standardmäßiger Eingabeverarbeitung	_POSIX_MAX_CANON
MAX_INPUT	Maximale Anzahl von Byte, die in der Eingabeschlange einer Datensichtstation erlaubt sind	_POSIX_MAX_INPUT
NAME_MAX	Maximale Anzahl der Zeichen in einem Dateinamen ohne das abschließende Nullbyte	_POSIX_NAME_MAX
PATH_MAX	Maximale Anzahl der Zeichen in einem Pfadnamen mit abschließendem Nullbyte	_POSIX_PATH_MAX
PIPE_BUF	Maximale Anzahl der Bytes die beim Schreiben auf eine Pipe garantiert atomar sind	_POSIX_PIPE_BUF

Zur Laufzeit erhöhbare Werte

Die Grenzwerte in der folgenden Tabelle werden durch die jeweilige Implementierung festgelegt. Eine Anwendung sollte davon ausgehen, dass der von der Implementierung in `limits.h` vorgegebene Wert das Minimum ist, das für alle Anwendungen zutrifft, die unter dieser Implementierung ablaufen.
Instanzen dieser Implementierung können den in `limits.h` vorgegebenen Wert zur Laufzeit jedoch erhöhen. Der tatsächliche Wert, der von einer bestimmten Instanz unterstützt wird, wird durch `sysconf()` zur Verfügung gestellt.

Name	Beschreibung	kleinster akzeptabler Wert
`BC_BASE_MAX`	Maximalwert für *obase*, der vom `bc`-Kommando erlaubt wird	`_POSIX2_BC_BASE_MAX`
`BC_DIM_MAX`	Maximale Anzahl von Elementen, die vom `bc`-Kommando in einem Vektor erlaubt sind	`_POSIX2_BC_DIM_MAX`
`BC_SCALE_MAX`	Maximalwert für *scale*, der vom `bc`-Kommando erlaubt ist	`_POSIX2_BC_SCALE_MAX`
`BC_STRING_MAX`	Maximale Länge einer Zeichenkettenkonstanten, die vom `bc`-Kommando akzeptiert wird	`_POSIX2_BC_STRING_MAX`
`COLL_WEIGHTS_MAX`	Maximale Anzahl von Gewichten, die einem Eintrag des `LC_COLLATE`-Schlüsselwortes `order` in der Datei für die Lokalitätsdefinition zugewiesen werden kann (siehe auch Abschnitt „Lokalität“ auf Seite 52)	`_POSIX2_COLL_WEIGHTS_MAX`
`EXPR_NEST_MAX`	Maximale Schachtelungstiefe von geklammerten Ausdrücken im `expr`-Kommando	`_POSIX2_EXPR_NEST_MAX`
`LINE_MAX`	Maximale Länge in Bytes einer Kommando-Eingabezeile (entweder Standardeingabe oder eine andere Datei), wenn das Kommando Textverarbeitung zulässt. Die Länge schließt auch Platz für das Zeilenendezeichen ein.	`_POSIX2_LINE_MAX`
`NGROUPS_MAX`	Maximale Anzahl gleichzeitig vorhandener weiterer Gruppennummern je Prozess	8
`RE_DUP_MAX`	Maximale Anzahl von wiederholten Vorkommen eines regulären Ausdrucks, die erlaubt sind, wenn man die Intervallschreibweise \{*m*,*n*\} benutzt (siehe auch „Reguläre Ausdrücke“ im Handbuch „POSIX Kommandos (BS2000/OSD)“)	`_POSIX2_RE_DUP_MAX`

Minimalwerte

Die symbolischen Konstanten in der folgenden Tabelle werden in `limits.h` mit den dargestellten Werten definiert. Es handelt sich um symbolische Namen für die kleinsten Werte eines bestimmten Merkmals für Systeme, die X/Open-konform sind. Symbolische Konstanten, die damit verbunden sind, werden an anderer Stelle in diesem Handbuch definiert, wo die tatsächliche Implementierung berücksichtigt wird. Eine X/Open-konforme Implementierung muss Minimalwerte unterstützen, die nicht kleiner sind, als die in dieser Tabelle angegebenen Werte. Eine portable Anwendung sollte keine größeren Werte benötigen, wenn sie korrekt ablaufen soll. Die folgenden Konstanten sind durch POSIX festgelegt und werden immer in `limits.h` definiert. Sie sind fest.

Name	Beschreibung	Wert
`_POSIX_ARG_MAX`	Länge der Argumentzeichenkette für die `exec`-Funktionen in Bytes einschließlich der Umgebungsdaten	4 096
`_POSIX_CHILD_MAX`	Anzahl gleichzeitiger Prozesse je realer Benutzernummer	6
`_POSIX_LINK_MAX`	Maximalwert des Verweiszählers einer Datei	8
`_POSIX_MAX_CANON`	Anzahl der Bytes in der Eingabeschlange einer Datensichtstation bei standardmäßiger Eingabeverarbeitung	255
`_POSIX_MAX_INPUT`	Anzahl von Bytes, für die in der Eingabeschlange einer Datensichtstation Platz verfügbar ist	255
`_POSIX_NAME_MAX`	Anzahl der Bytes in einem Dateinamen, ohne abschließendes Nullbyte	14
`_POSIX_NGROUPS_MAX`	Anzahl gleichzeitiger weiterer Gruppennummern je Prozess	0
`_POSIX_OPEN_MAX`	Anzahl von Dateien, die ein Prozess gleichzeitig offen haben kann	16
`_POSIX_PATH_MAX`	Anzahl der Bytes in einem Pfadnamen	255
`_POSIX_PIPE_BUF`	Anzahl der Bytes, die beim Schreiben auf eine Pipe atomar geschrieben werden kann	512
`_POSIX_SSIZE_MAX`	Wert, der einem Objekt vom Datentyp `ssize_t` zugewiesen werden kann	32 767
`_POSIX_STREAM_MAX`	Anzahl der Datenströme, die ein Prozess gleichzeitig offen haben kann	8
`_POSIX_TZNAME_BUF`	Maximale Anzahl von Bytes, die für einen Zeitzonennamen unterstützt wird (nicht von `TZ`-Variable)	3
`_POSIX2_BC_BASE_MAX`	Maximalwert von *obase*, der vom `bc`-Kommando erlaubt wird	99

Name	Beschreibung	Wert
_POSIX2_BC_DIM_MAX	Maximale Anzahl von Elementen, die in einem Vektor vom bc-Kommando erlaubt werden	2 048
_POSIX2_BC_SCALE_MAX	Maximalwert von *scale*, der vom bc-Kommando erlaubt wird	99
_POSIX2_BC_STRING_MAX	Maximale Länge einer Zeichenkettenkonstanten, die vom bc-Kommando erlaubt wird	1 000
_POSIX2_COLL_WEIGHTS_MAX	Maximale Anzahl von Gewichten, die einem Eintrag des LC_COLLATE-Schlüsselwortes order in der Datei für die Lokalitätsdefinition zugewiesen werden kann (siehe Abschnitt „Lokalität“ auf Seite 52)	2
_POSIX2_EXPR_NEST_MAX	Maximale Schachtelungstiefe von geklammerten Ausdrücken, im expr-Kommando	32
_POSIX2_LINE_MAX	Maximale Länge in Bytes einer Kommando-Eingabezeile (entweder Standardeingabe oder eine andere Datei), wenn das Kommando Textverarbeitung zulässt. Die Länge schließt auch Platz für das Zeilenendezeichen ein.	2 048
_POSIX2_RE_DUP_MAX	Maximale Anzahl von wiederholten Vorkommen eines regulären Ausdrucks, die erlaubt sind, wenn man die Intervallschreibweise \{*m*,*n*\} benutzt (siehe auch „Reguläre Ausdrücke“ im Handbuch „POSIX Kommandos (BS2000/OSD)“)	255

Numerische Grenzwerte

Die Werte in den folgenden Tabellen sind in limits.h als konstante Ausdrücke definiert, für den Gebrauch in #if-Präprozessor-Anweisungen. Außerdem werden, mit Ausnahme von CHAR_BIT, DBL_DIG, DBL_MAX, FLT_MAX, LONG_BIT, WORD_BIT und MB_LEN_MAX, die symbolischen Namen als Ausdrücke mit dem korrekten Datentyp definiert. Wenn der Wert eines Objekts vom Typ char in einem Ausdruck als ganzzahliger Wert mit Vorzeichen behandelt wird, haben CHAR_MIN und SCHAR_MIN bzw. CHAR_MAX und SCHAR_MAX den gleichen Wert. Andernfalls hat CHAR_MIN den Wert 0, und CHAR_MAX hat den gleichen Wert wie UCHAR_MAX.

Maximalwerte

Name	Beschreibung	kleinster akzeptabler Wert
CHAR_BIT	Anzahl der Bits in einem char	8
CHAR_MAX	Maximalwert für Datentyp char	UCHAR_MAX oder SCHAR_MAX
DBL_DIG	Signifikante Stellen für Datentyp double (Wird zukünftig vom X/Open-Standard nicht mehr unterstützt.)	10
DBL_MAX	Maximaler Dezimalwert eines double (Wird zukünftig vom X/Open-Standard nicht mehr unterstützt.)	1E+37
FLT_DIG	Signifikante Stellen für Datentyp float (Wird zukünftig vom X/Open-Standard nicht mehr unterstützt.)	6
FLT_MAX	Maximaler Dezimalwert eines float (Wird zukünftig vom X/Open-Standard nicht mehr unterstützt.)	1E+37
INT_MAX	Maximalwert für Datentyp int	32 767
LONG_BIT	Anzahl der Bits in einem long	32
LONG_MAX	Maximalwert für Datentyp long	+2 147 483 647
MB_LEN_MAX	Maximale Byte-Anzahl für ein Zeichen für jede unterstütze Lokalität	1
SCHAR_MAX	Maximalwert für Datentyp signed char	+127
SHRT_MAX	Maximalwert für Datentyp short	+32 767
SSIZE_MAX	Maximalwert für Datentyp ssize_t	_POSIX_SSIZE_MAX
UCHAR_MAX	Maximalwert für Datentyp unsigned char	255
UINT_MAX	Maximalwert für Datentyp unsigned int	65 535
ULONG_MAX	Maximalwert für Datentyp unsigned long	4 294 967 295
USHRT_MAX	Maximalwert für Datentyp unsigned short	65 535
WORD_BIT	Bitanzahl eines Worts oder für Datentyp int	16

Minimalwerte

Name	Beschreibung	größter akzeptabler Wert
CHAR_MIN	Minimalwert für Datentyp char	SCHAR_MIN oder 0
INT_MIN	Minimalwert für Datentyp int	-32 767
LONG_MIN	Minimalwert für Datentyp long	-2 147 483 647
SCHAR_MIN	Minimalwert für Datentyp signed char	-127
SHRT_MIN	Minimalwert für Datentyp short	-32 767

Andere unveränderliche Werte

Folgende Konstanten sind in allen Systemen in limits.h definiert.

Name	Beschreibung	kleinster akzeptabler Wert
CHARCLASS_NAME_MAX	Maximale Byte-Anzahl in einem Namen für eine Zeichenklasse	14
NL_ARGMAX	Maximalwert einer Ziffer in printf()- und scanf()-Aufrufen	9
NL_LANGMAX	Maximale Byte-Anzahl in einem LANG-Namen	14
NL_MSGMAX	Maximale Meldungsnummer	32 767
NL_NMAX	Maximale Byte-Anzahl in N-zu-1 Sortierabbildung	*
NL_SETMAX	Maximale Mengennummer	255
NL_TEXTMAX	Maximale Anzahl von Zeichen in einem Meldungstext	_POSIX2_LINE_MAX
NZERO	Standard-Prozesspriorität	20
TMP_MAX	Maximalzahl eindeutiger Namen, die von tmpnam() generiert werden können (wird zukünftig vom X/Open-Standard nicht mehr unterstützt)	10 000

Hinweis TMP_MAX wird auch in stdio.h definiert, um den ISO C Standard zu erfüllen.

DBL_DIG, DBL_MAX, DBL_MIN, FLT_DIG, FLT_MAX und FLT_MIN werden auch in float.h definiert.

Siehe auch fpathconf(), pathconf(), sysconf(), unistd.h.

locale.h - Kategorie-Makros

Syntax #include <locale.h>

Beschreibung

`locale.h` definiert die Struktur `lconv`, die mindestens folgende Komponenten enthält:

```
char *currency_symbol
char *decimal_point
char frac_digits
char *grouping
char *int_curr_symbol
char int_frac_digits
char *mon_decimal_point
char *mon_grouping
char *mon_thousands_sep
char *negative_sign
char n_cs_precedes
char n_sep_by_space
char n_sign_posn
char *positive_sign
char p_cs_precedes
char p_sep_by_space
char p_sign_posn
char *thousands_sep
```

`locale.h` definiert `NULL` und mindestens folgende Makros:

```
LC_ALL
LC_COLLATE
LC_CTYPE
LC_MESSAGES
LC_MONETARY
LC_NUMERIC
LC_TIME
```

Sie werden zu verschiedenen ganzzahligen konstanten Ausdrücken expandiert, um als erstes Argument der Funktion `setlocale()` verwendet zu werden.

Folgende Namen sind als Funktionen deklariert:

struct lconv *localeconv(void);

char setlocale(int *category*, const char **locale*);

Siehe auch `localeconv()`, `setlocale()`, Abschnitt „Umgebungsvariablen“ auf Seite 71.

math.h - mathematische Funktionen und Konstanten

Syntax #include <math.h>

Beschreibung

Diese Header-Datei enthält Deklarationen aller Funktionen der mathematischen Bibliothek sowie verschiedener Funktionen der C-Bibliothek, die Gleitpunktwerte zurückgeben.

Erweiterung

`HUGE` Höchstwert einer Gleitpunktzahl einfacher Genauigkeit ❑

Die nachstehenden mathematischen Konstanten sind zur einfacheren Anwendung durch den Benutzer wie folgt definiert:

`M_E`	Basis des natürlichen Logarithmus (e)
`M_LOG2E`	Logarithmus zur Basis 2 von e
`M_LOG10E`	Logarithmus zur Basis 10 von e
`M_LN2`	natürlicher Logarithmus von 2
`M_LN10`	natürlicher Logarithmus von 10
`M_PI`	π, Verhältnis des Umfangs eines Kreises zu seinem Durchmesser
`M_PI_2`	$\pi/2$
`M_PI_4`	$\pi/4$
`M_1_PI`	$1/\pi$
`M_2_PI`	$2/\pi$
`M_2_SQRTPI`	2/Wurzel aus π
`M_SQRT2`	positive Quadratwurzel von 2
`M_SQRT1_2`	1/positive Quadratwurzel von 2

Die folgenden mathematischen Konstanten sind auch in dieser Include-Datei definiert:

`MAXFLOAT`	Maximalwert einer Gleitpunktzahl einfacher Genauigkeit
`HUGE_VAL`	positives Unendlich vom Typ `double`

Das Makro `HUGE_VAL` ist definiert, um Fehlerwerte zu repräsentieren, die von den Mathematikfunktionen geliefert werden. Da der Wert $+\infty$ nicht auf der Hardware darstellbar ist, ist `HUGE_VAL = {DBL_MAX}`.

Die folgenden Namen sind als Funktionen deklariert:

```
double acos(double x);
```

```
double asin(double x);
```

```
double atan2(double x);
double atan(double x);
double ceil(double x);
double cos(double x);
double cosh(double x);
double exp(double x);
double fabs(double x);
double floor(double x);
double fmod(double x);
double frexp(double x);
double ldexp(double x);
double log10(double x);
double log(double x);
double modf(double x);
double pow(double x);
double sin(double x);
double sinh(double x);
double sqrt(double x);
double tan(double x);
double tanh(double x);
double erf(double x);
double erfc(double x);
double gamma(double x);
double hypot(double x, double y);
double j0(double x);
double j1(double x);
double jn(int n, double x);
double lgamma(double x);
double y0(double x);
```

```
double y1(double x);

double yn(int n, double x);

int isnan(double x);
```

Die folgende externe Variable wird definiert:

```
extern int signgam;
```

Zur Definition von verschiedenen maschinenabhängigen Konstanten siehe `values()`.

Siehe auch `matherr()`, `values()`, `acos()`, `asin()`, `atan()`, `atan2()`, `ceil()`, `cos()`, `cosh()`, `erf()`, `exp()`, `fabs()`, `floor()`, `fmod()`, `frexp()`, `gamma()`, `hypot()`, `isnan()`, `j0()`, `ldexp()`, `lgamma()`, `log()`, `log10()`, `modf()`, `pow()`, `sin()`, `sinh()`, `sqrt()`, `tan()`, `tanh()`, `y0()`.

monetary.h - Typen für monetäre Werte

Syntax #include <monetary.h>

Beschreibung

Die Include-Datei `monetary.h` definiert die folgenden Datentypen durch `typedef`:

`size_t` Wie in `stddef.h` beschrieben.

`ssize_t` Wie in `sys/types.h` beschrieben.

Der folgende Name ist als Funktion deklariert:

```
ssize_t strfmon(char *s, size_t maxsize, const char *format, ...);
```

Siehe auch `strfmon()`.

ndbm.h - Definitionen von Operationen für die ndbm-Datenbank

Syntax #include <ndbm.h>

Beschreibung

`ndbm.h` definiert den Typ `datum` als folgende Struktur:

```
typedef struct {
        void  *dptr;              /* Zeiger auf die Anwendungsdaten */
        size_t  dsize;            /* Länge des Objekts, auf das dptr zeigt */
} datum
```

`ndbm.h` definiert den Typ `DBM` durch folgende Struktur:

```
typedef struct {
       int dbm_dirf;                /* Verzeichnis öffnen */
       int dbm_pagf;                /* Page-Datei öffnen  */
       int dbm_flags;               /* Flags, siehe unten  */
       long dbm_maxbno;             /* etztes ‚Bit' in Verzeichnis */
       long dbm_bitno;              /* Nummer des aktuellen Bits  */
       long dbm_hmask;              /* Hash-Maske */
       long dbm_blkptr;             /* aktueller Block für dbm_nextkey  */
       int dbm_keyptr;              /* aktueller Key für dbm_nextkey  */
       long dbm_blkno;              /* aktuelle Seite für read bzw. write  */
       long dbm_pagbno;             /* aktuelle Seite im Seitenpuffer  */
       char dbm_pagbuf[_PBLKSIZ];   /* Page-Datei Blockpuffer  */
       long dbm_dirbno;             /* aktueller Block in dirbuf  */
       char dbm_dirbuf[_DBLKSIZ];   /* Blockpuffer für Dateiverzeichnis  */
} DBM;
```

Ferner sind die folgenden Konstanten definiert, die mögliche Werte des Parameters *store_mode* von `dbm_store()` bezeichnen:

`DBM_INSERT` es dürfen nur neue Einträge eingefügt werden
`DBM_REPLACE` bestehende Einträge dürfen ersetzt werden

Folgende Funktionen sind definiert:

```
int   dbm_clearerr(DBM *db);
void  dbm_close(DBM *db);
int   dbm_delete(DBM *db, datum key);
int   dbm_error(DBM *db);
datum dbm_fetch(DBM *db, datum key);
datum dbm_firstkey(DBM *db);
datum dbm_nextkey(DBM *db);
DBM  *dbm_open(const char *file, int open_flags, mode_t file_mode);
int   dbm_store(DBM *db, datum key, datum content, int store_mode);
```

Folgende Funktionen werden auch als Makros angeboten:

```
int   dbm_clearerr(DBM *db);
int   dbm_error(DBM *db);
```

Siehe auch `dbm_clearerr()`.

nl_types.h - Datentypen für NLS

Syntax #include <nl_types.h>

Beschreibung

Folgende Datentypen sind durch `typedef` definiert:

`nl_catd`	Wird von den Meldungskatalog-Funktionen `catopen()`, `catgets()` und `catclose()` verwendet, um einen Katalog-Deskriptor zu identifizieren.
`nl_item`	Wird von `nl_langinfo()` verwendet, um `langinfo`-Daten zu identifizieren. Werte für Objekte des Typs `nl_item` werden in `langinfo.h` definiert.

Folgende Konstanten sind definiert:

`NL_CAT_LOCALE`	Wert, der als *oflag*-Argument an `catopen()` übergeben werden muss, um sicherzustellen, dass die Auswahl des Meldungskatalogs von der Lokalitätskategorie `LC_MESSAGES` abhängt und nicht direkt von der Umgebungsvariablen `LANG`.
`NL_SETD`	Wird von `gencat` verwendet, wenn keine `$set`-Direktive in einer Meldungstext-Quelldatei angegeben wird. Diese Konstante kann als Wert von `set_id` an nachfolgende Aufrufe von `catgets()` verwendet werden, um Meldungen aus der Standard-Meldungsmenge zu ermitteln. Der Wert von `NL_SETD` ist implementierungsabhängig.

Folgende Namen sind als Funktionen deklariert:

```
int catclose(nl_catd catd);

char *catgets(nl_catd catd, int set_id, int msg_id, const char *str);

nl_catd catopen(const char *name, int oflag);
```

Siehe auch `catclose()`, `catgets()`, `catopen()`, `nl_langinfo()`, `langinfo.h`, Kommando `gencat` im Handbuch „POSIX Kommandos (BS2000/OSD)“.

poll.h - Definitionen für die poll()-Funktion

Syntax #include <poll.h>

poll.h definiert die pollfd Struktur:

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

poll.h definiert den folgenden Typ mit typedef:

nfds_t	Vorzeichenloser, ganzzahliger Datentyp, der die Anzahl der Dateideskriptoren angibt.

Durch ODER-Verknüpfung der nachfolgenden symbolischen Konstanten in beliebiger Kombination werden die Parameter events und revents in der pollfd Struktur aufgebaut (zulässig ist auch die Angabe 0):

POLLIN	Wie POLLRDNORM | POLLRDBAND.
POLLRDNORM	Daten mit Priorität = 0 können gelesen werden.
POLLRDBAND	Daten mit Priorität > 0 können gelesen werden.
POLLPRI	Daten mit hoher Priorität können gelesen werden.
POLLOUT	wie POLLWRNORM.
POLLWRNORM	Daten mit Priorität = 0 können geschrieben werden.
POLLERR	Eine Fehlermeldung liegt vor (gilt nur für revents).
POLLHUP	Verbindung zum Gerät ist unterbrochen (gilt nur für revents).
POLLNVAL	Ungültiges fd Element (gilt nur für revents).

poll.h definiert folgende Funktionen:

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

Siehe auch poll().

pwd.h - Kennwortstruktur

#include <pwd.h>

Die Include-Datei `pwd.h` stellt eine Definition für die Struktur `struct passwd` zur Verfügung, welche die folgenden Komponenten enthält:

`char`	`*pw_name`	Benutzerkennung
`uid_t`	`pw_uid`	Benutzernummer
`gid_t`	`pw_gid`	Gruppennummer
`char`	`*pw_dir`	HOME-Dateiverzeichnis
`char`	`*pw_shell`	Startprogramm
`char`	`*pw_passwd`	LOGON-Kennwort (BS2000)
`char`	`*pw_age`	Kennwortalter (SINIX, wird aus Kompatibilitätsgründen unterstützt)
`char`	`*pw_comment`	Kommentarzeile (zB.: Adresse, Raum-Nr.)
`char`	`*pw_gecos`	Sicherheitsattribut (SINIX, wird aus Kompatibilitätsgründen unterstützt.)

Folgende Namen sind als Funktionen deklariert:

struct passwd *getpwnam(const char `*`*name*`)`;

int getpwnam_r(const char `*`*name*, struct passwd * *pw*, char * *name1*, size_t *siz*, struct passwd **ppw);

`struct passwd *getpwuid(uid_t` ***uid*** `)`;

int getpwuid_r`(uid_t` *uid*, struct passwd * *pw*, char * *name1*, size_t *siz*, struct passwd **ppw`)`;

re_comp.h - Vergleich regulärer Ausdrücke bei re_comp

#include <re_comp.h>

Die folgenden Funktionen sind deklariert:

char *re_comp(const char **string*);

int re_exec(const char **string*);

Diese Include-Datei wird in der nächsten Version nicht mehr angeboten.

`re_comp().`

regexp.h - Deklarationen für reguläre Ausdrücke

Syntax #include <regexp.h>

Beschreibung

Folgende Namen sind als Funktionen deklariert:

int advance(const char **string*, const char **expbuf*);
char *compile(char **instring*, const char **expbuf*);
int step(const char **string*, const char **expbuf*);

Folgende externe Variablen sind definiert:

extern char *loc1;
extern char *loc2;
extern char *locs;

Wird zukünftig vom X/Open-Standard nicht mehr unterstützt.

`regexp()`.

search.h - Tabellen durchsuchen

Syntax #include <search.h>

Beschreibung

Die Include-Datei `search.h` führt ein `typedef ENTRY` für den Datentyp `struct entry` durch, der die folgenden Komponenten enthält:

```
char        *key;

char        *data;
```

Die Aufzählungstypen `ACTION` und `VISIT` werden mit `typedef` wie folgt definiert:

```
enum { FIND, ENTER } ACTION;
enum { preorder, postorder, endorder, leaf } VISIT;
```

Die folgenden Namen sind als Funktionen deklariert:

```
int hcreate(size_t ne1);

void hdestroy(void);

ENTRY *hsearch(ENTRY item, ACTION action);

void *lfind(const void *key, const void *base, size_t *ne1p, size_t width,
      int (*compar)(const void *, const void*));

void *lsearch(const void *key, const void *base, size_t *ne1p, size_t
      width, int (*compar)(const void *, const void*))

void *tsearch(const void *key, void **rootp,
      int (*compar)(const void *, const void*))

void *tdelete(const void *key, void **rootp,
      int (*compar)(const void *, const void*))

void *tfind(const void *key, void **rootp,
      int (*compar)(const void *, const void*))

void *twalk(const void *root, void (*action)(const void *, VISIT, int))
```

Siehe auch `hsearch()`, `lsearch()`, `tsearch()`, `sys/types.h`.

setjmp.h - Deklarationen für Stackumgebung

Syntax #include <setjmp.h>

Beschreibung

Die Include-Datei `setjmp.h` enthält Typdefinitionen mit `typedef` für die Datentypen `jmp_buf` und `sigjmp_buf`.

`longjmp()` und `siglongjmp()` werden als Funktionen deklariert:

```
void longjmp(jmp_buf env, int val);
```

```
void siglongjmp(sigjmp_buf env, int val);
```

`setjmp()` wird als Funktion deklariert und `sigsetjmp()` als Makro definiert:

```
int setjmp(jmp_buf env);
```

```
int sigsetjmp(sigjmp_buf env, int savemask);
```

Siehe auch `longjmp()`, `setjmp()`, `siglongjmp()`, `sigsetjmp()`.

signal.h - Signale

Syntax #include <signal.h>

Beschreibung

signal.h deklariert Konstanten für die Darstellung im System auftretender Signale. Sie beginnen mit den Buchstaben SIG. Jedes Signal besitzt einen eindeutigen, positiven ganzzahligen Wert. Der Wert 0 ist für das Nullsignal reserviert (siehe auch kill()). Eine Implementierung kann zusätzliche Signale, die im System auftreten können, definieren.

Als Signalaktionen sind folgende symbolische Konstanten definiert:

SIG_DFL	Anforderung der voreingestellten Signalbehandlung
SIG_ERR	Returnwert von signal() im Fehlerfall
SIG_HOLD	Anforderung, den Prozess anzuhalten
SIG_IGN	Anforderung, das Signal zu ignorieren

Jede dieser Konstanten expandiert zu einem eindeutigen Ausdruck des Typs void (*)(int), dessen Wert keiner deklarierbaren Funktion entspricht.

Folgende Datentypen werden durch typedef definiert:

sig_atomic_t	ganzzahliger Datentyp eines Objekts, auf das als unteilbare Einheit zugegriffen werden kann, sogar im Fall von asynchronen Unterbrechungen
sigset_t	ganzzahliger Datentyp oder Struktur eines Objektes, das eine Signalmenge darstellt
pid_t	siehe sys/types.h

Folgende symbolische Namen für Signalnummern - kurz Signale genannt - werden von den in diesem Handbuch beschriebenen Bibliotheksfunktionen unterstützt:

Signal	**Beschreibung**	**voreingestellte Signalbehandlung**
SIGABRT	Prozessabbruch	Abnormales Prozessende; zusätzlich können vorher Signalbehandlungsfunktionen ausgeführt werden.
SIGALRM	Alarmuhr abgelaufen STXIT-Klasse: RTIMER	Abnormales Prozessende mit allen Konsequenzen von _exit() außer, dass der Status für wait() zur Verfügung steht und waitpid() das anormale Prozessende durch das Signal anzeigt.
SIGBUS *Erweiterung*	Adressfehler	Abnormales Prozessende und Speicherabzug.

Signal	Beschreibung	voreingestellte Signalbehandlung
SIGCHLD	Sohnprozess wurde beendet oder angehalten	Signal ignorieren.
SIGCONT	Fortsetzen, wenn angehalten	Angehaltenen Prozess fortsetzen, andernfalls Signal ignorieren.
SIGDVZ *Erweiterung*	Division durch 0; (wird abgebildet auf SIGFPE) STXIT-Klasse: PROCHK	siehe SIGABRT
SIGEMT *Erweiterung*	Unterbrechung für Emulator	Abnormales Prozessende und Speicherabzug.
SIGFPE	Fehler bei Gleitpunktoperation STXIT-Klasse: PROCHK STXIT-Gewichte: 0x64: Exponentenüberlauf 0x68: Divisionsfehler 0x6c: Mantisse=0 0x70: Exponentenunterlauf 0x74: Dezimalüberlauf 0x78: Fixpunktüberlauf	siehe SIGABRT
SIGHUP	Verbindung zum Terminal unterbrochen (hang up) STXIT-Klasse: ABEND	siehe SIGALRM
SIGILL	Unzulässiger Befehl (illegal instruction) STXIT-Klasse: PROCHK	siehe SIGABRT; dieses Signal wird nach dem Abfangen nicht zurückgesetzt.
SIGINT	Unterbrechung am Terminal STXIT-Klasse: ESCPBRK	siehe SIGALRM
SIGIO *Erweiterung*	Ein-/Ausgabe auf Socket möglich (entspricht SIGPOLL)	Signal ignorieren.
SIGIOT *Erweiterung*	Abbruch	Abnormales Prozessende und Speicherabzug.
SIGKILL	Unbedingter Abbruch, der weder abgefangen noch ignoriert werden kann (kill)	siehe SIGALRM
SIGPIPE	Schreiben auf Pipe, die nicht lesbar ist	siehe SIGALRM
SIGPOLL *Erweiterung*	Pollbares Ereignis	Signal ignorieren.
SIGPROF *Erweiterung*	Ablauf eines Zeitgebers zur Laufzeitanalyse	Exit

Signal	Beschreibung	voreingestellte Signalbehandlung
SIGPWR *Erweiterung*	Spannungsunterbrechung/Neustart	Signal ignorieren.
SIGQUIT	Beenden des Terminals	siehe SIGABRT
SIGSEGV	Unerlaubter Zugriff auf Speichersegment STXIT-Klasse: ERROR	siehe SIGABRT
SIGSTOP	Ausführung anhalten (kann nicht abgefangen oder ignoriert werden)	Prozess anhalten.
SIGSYS *Erweiterung*	Fehlerhafter Systemaufruf STXIT-Gewicht: 0xb0: Validierungsfehler	Abnormales Prozessende und Speicherabzug.
SIGTERM	Prozessbeendigung STXIT-Klasse: TERM	siehe SIGALRM
SIGTIM *Erweiterung*	Zeitgeber; (wird abgebildet auf SIGVTALRM) STXIT-Klasse: TIMER	Exit
SIGTRAP *Erweiterung*	Unterbrechung für Ablaufverfolgung	Abnormales Prozessende und Speicherabzug; dieses Signal wird nach dem Abfangen nicht zurückgesetzt.
SIGTSTP	Stopsignal für Terminal	Prozess anhalten.
SIGTTIN	Hintergrundprozess versucht zu lesen	Prozess anhalten.
SIGTTOU	Hintergrundprozess versucht zu schreiben	Prozess anhalten.
SIGURG *Erweiterung*	Dringender Vorfall an Socket	Signal ignorieren.
SIGUSR1	Benutzerdefiniertes Signal 1	siehe SIGALRM
SIGUSR2	Benutzerdefiniertes Signal 2	siehe SIGALRM
SIGVTALRM *Erweiterung*	STXIT-Gewicht: 0x20: SETIC-Intervall für CPU-Zeit abgelaufen	Exit
SIGWINCH *Erweiterung*	Änderung der Fenstergröße	Signal ignorieren.
SIGXCPU *Erweiterung*	CPU-Zeitbegrenzung überschritten STXIT-Klasse: RUNOUT	Abnormales Prozessende und Speicherabzug.
SIGXFSZ *Erweiterung*	Datei-Speicherbegrenzung überschritten	Abnormales Prozessende und Speicherabzug.

`signal.h` deklariert die Struktur `sigaction`, die zumindest folgende Komponenten enthält:

`void (*sa_handler)(int)`
: Signalbehandlungsfunktion

`sigset_t sa_mask`
: Signalmenge, die während der Ausführung der Signalbehandlungsfunktion blockiert ist

`int sa_flags`
: Spezielle Marken

Als Flag-Bits sind folgende symbolische Konstanten definiert:

`SA_NOCLDSTOP`
: Wenn Sohnprozess angehalten wird, `SIGCHLD` nicht generieren.

`SIG_BLOCK`
: Die Ergebnismenge ist die Vereinigung der aktuellen Menge und der Signalmenge, auf die `set` zeigt.

`SIG_UNBLOCK`
: Die Ergebnismenge ist die Schnittmenge der aktuellen Menge und der Signalmenge, auf die `set` zeigt.

`SIG_SETMASK`
: Die Ergebnismenge ist die Signalmenge, auf die `set` zeigt.

Folgende Namen sind als Funktionen deklariert:

```
int kill(pid_t pid, int sig);
int pthread_kill(p_thread pthr, int signo);
int pthread_sigmask(int signo, const sigset_t *cset, sigset_t *set);
int raise(int sig);
int sigaction(int sig, const struct sigactions *act, struct sigaction *oact);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigismember(const sigset_t *set, int signo);
void (*signal(int sig, void(*func)(int)))(int);
int sigpending(sigset_t *set);
int sigprocmask(int how, cont sigset_t *set, sigset_t *oset);
int sigsuspend(cont sigset_t *sigmask);
```

Siehe auch `alarm()`, `kill()`, `pthread_kill()`, `pthread_sigmask()`, `raise()`, `sigaction()`, `sigaddset()`, `sigdelset()`, `sigemptyset()`, `sigfillset()`, `sigismember()`, `signal()`, `sigpending()`, `sigprocmask()`, `sigsuspend()`, `wait()`, `sys/types.h`.

stdarg.h - variable Argumentenliste bearbeiten

Syntax #include <stdarg.h>

Syntax void va_start(va_list *ap*, *argN*);
type va_arg(va_list *ap*, type);
void va_end(va_list *ap*);

Beschreibung

`stdarg.h` enthält Makros, die die Entwicklung von portablen Prozeduren erlauben, welche eine variable Anzahl von Argumenten verschiedener Typen akzeptieren. Routinen mit variablen Argumentenlisten (wie `printf()`), die diese Makros nicht verwenden, sind nicht portabel, da verschiedene Maschinen unterschiedliche Konventionen der Argumentübergabe verwenden.

`va_list` ist für Variablen definiert, die die Liste durchlaufen.

Das Makro `va_start()` wird vor dem Zugriff auf die namenlosen Argumente ausgeführt und initialisiert *ap* zum nachfolgenden Bearbeiten durch `va_arg()` und `va_end()`. *argN* ist der Bezeichner des letzten Parameters in der variablen Parameterliste der Funktionsdefinition. Wenn dieser Parameter deklariert wird mit der Speicherklasse `register`, mit einem Funktions- oder Feldtyp oder gar mit einem Typ, der nicht mit dem Typ kompatibel ist, der nach der Anwendung der voreingestellten Argumentbehandlung resultiert, ist das Ergebnis undefiniert.

Das Makro `va_arg()` wird als ein Ausdruck bewertet, der den Typ und den Wert des nächsten Arguments des Aufrufs darstellt. *ap* sollte vorher durch `va_start()` initialisert werden. Jeder Aufruf von `va_arg()` ändert *ap* so, dass die Werte von nachfolgenden Argumenten der Reihe nach zurückgeliefert werden. Der Parameter *type* ist der Typname des nächsten zurückgelieferten Arguments. Der Typname muss so angegeben sein, dass der Typ eines Zeigers auf ein Objekt dieses Typs über Anhängen eines * an *type* erzeugt werden kann. Gibt es kein nächstes Argument oder ist *type* nicht kompatibel mit dem Typ des nächsten Arguments (entsprechend der voreingestellten Argumentbehandlung), ist das Verhalten undefiniert.

Das Makro `va_end()` wird zum Aufräumen verwendet. Es macht *ap* unbenutzbar, bis `va_start()` erneut aufgerufen wird.

Mehrfache Durchläufe, mit `va_start()` beginnend und `va_end()` endend, sind möglich.

Beispiel Im folgenden Beispiel werden die Zeiger auf Zeichenketten über eine variable Argumentenliste in ein Feld eingelesen. Dies geschieht über die Funktion `f1()`, die höchstens `MAXARGS`-Argumente akzeptiert. Danach wird das Feld als einzelnes Argument an die Funktion *f2* übergeben. Die Anzahl der Zeiger wird im ersten Argument von `f1()` angegeben.

```
#include <stdarg.h>
#define MAXARGS 31

int f2(int, char *);
void f1(int n_ptrs, ...)
{
   va_list ap;
   char *array[MAXARGS];
   int ptr_no = 0;

   if (n_ptrs > MAXARGS)
       n_ptrs = MAXARGS;
   va_start(ap, n_ptrs);
   while (ptr_no < n_ptrs)
       array[ptr_no++] = va_arg(ap, char*);
   va_end(ap);
   f2(n_ptrs, array);
}
```

Bei jedem Aufruf von `f1()` sollte die Definition der Funktion oder eine Deklaration wie `void f1(int, ...)` verfügbar sein.

Hinweis Es liegt bei der aufrufenden Routine, anzugeben, wie viele Argumente übergeben werden, da die Feststellung der Anzahl der Argumente aus dem Stack nicht immer möglich ist. Beispielsweise wird `execl()` ein Nullzeiger übergeben, um das Ende der Liste anzuzeigen. `printf()` erkennt anhand des Formats die Anzahl der Argumente. Die Angabe eines zweiten Arguments `char`, `short` oder `float` für `va_arg()` ist nicht portabel, da die von der aufgerufenen Funktion sichtbaren Argumente nicht `char`, `short` oder `float` sind. C konvertiert `char`- und `short`-Argumente in `int` und wandelt `float`-Argumente in `double` um, bevor sie an eine Funktion übergeben werden.

Siehe auch `vprintf()`.

stddef.h - Definitionen für Standard-Datentypen

Syntax #include <stddef.h>

Beschreibung

`stddef.h` definiert folgende symbolische Namen:

NULL — Nullzeiger

`offsetof(`*type*`,` *member-designator*`);`
Ganzzahliger konstanter Ausdruck vom Typ `size_t`, dessen Wert der Abstand einer Strukturkomponente (*member*) zum Strukturbeginn (*type*) in Bytes ist.

Folgende Datentypen sind durch `typedef` definiert:

`ptrdiff_t` — Ganzzahliger Datentyp mit Vorzeichen, der das Ergebnis einer Zeigersubtraktion enthält.

`wchar_t` — Ganzzahliger Datentyp, dessen Wertebereich unterschiedliche Langzeichen repräsentieren kann. Er ist für alle Elemente des größten Zeichensatzes der unterstützten Lokalitäten durch die Entwicklungsumgebung spezifiziert; das Nullzeichen hat den Wert 0 und jedes Element des portablen Zeichensatzes hat einen Zeichenwert, der dem Wert entspricht, der gilt, wenn er als Einzelzeichen in einer ganzzahligen Zeichen-Konstante benutzt wird.

`size_t` — Vorzeichenloser ganzzahliger Datentyp, der das Ergebnis des `sizeof`-Operators enthält.

Siehe auch `wchar.h`, `sys/types.h`.

stdio.h - Gepufferte Standard-Ein-/Ausgabe

Syntax #include <stdio.h>

Beschreibung

stdio.h definiert die folgenden Makronamen als positive ganzzahlige Konstanten:

BUFSIZ	Größe der Puffer für stdio.h.
FILENAME_MAX	Maximale Größe in Byte für Dateinamen-Zeichenketten, für die von der Implementierung garantiert wird, dass sie geöffnet werden können.
FOPEN_MAX	Maximalzahl von Dateien, die garantiert gleichzeitig geöffnet werden können.
_IOFBF	voll gepufferte Ein-/Ausgabe
_IOLBF	zeilenweise gepufferte Ein-/Ausgabe
_IONBF	ungepufferte Ein-/Ausgabe
L_ctermid	Maximalgröße des Zeichenvektors für ctermid-Ausgaben
L_cuserid	Maximalgröße des Zeichenvektors für cuserid-Ausgaben (Wird zukünftig vom X/Open-Standard nicht mehr unterstützt.)
L_tmpnam	Maximalgröße des Zeichenvektors für tmpnam-Ausgaben
SEEK_CUR	Suche relativ zur aktuellen Position.
SEEK_END	Suche relativ zum Dateiende.
SEEK_SET	Suche relativ zum Dateianfang.
TMP_MAX	Minimalzahl eindeutiger Dateinamen, die von tmpnam() generiert werden. Maximalzahl von zuverlässig möglichen tmpnam-Aufrufen für eine Anwendung. Der Wert ist größer oder gleich 10.000.

Folgende Makronamen sind definiert:

EOF	Returnwert für Dateiende, die als negative ganzzahlige Konstante definiert ist.
NULL	Nullzeiger
P_tmpdir	Voreingestelltes Dateiverzeichnis-Präfix für tempnam(), das als Zeichenketten-Konstante definiert ist.
stderr	Standardfehlerausgabestrom, der als Datentypzeiger für FILE definiert ist.

`stdin`	Standardeingabestrom, der als Datentypzeiger für `FILE` definiert ist.
`stdout`	Standardausgabestrom, der als Datentypzeiger für `FILE` definiert ist.

Folgende Datentypen sind durch `typedef` definiert:

FILE	Eine Struktur, die Informationen über eine Datei enthält
`fpos_t`	Datentyp, der alle Informationen enthält, die dazu nötig sind, jede Position innerhalb einer Datei eindeutig zu bezeichnen
`fpos64_t`	Datentyp, der alle Informationen enthält, die notwendig sind, um jede Position innerhalb einer Datei, in welcher der maximale Offset durch den Wertebereich eines off64_t-Objekts festgelegt ist, eindeutig zu spezifizieren.
`va_list`	siehe `stdarg.h`
`size_t`	siehe `stddef.h`

Folgende Namen sind als Funktionen deklariert:

`void clearerr(FILE *`*stream*`);` (und auch als Makro definiert)

`char *ctermid(char *`*s*`);`

`char *cuserid(char *`*s*`);` (wird zukünftig vom X/Open-Standard nicht mehr unterstützt)

`int fclose(FILE *`*stream*`);`

`FILE *fdopen(int` *fildes*`, const char *`*mode*`);`

`int feof(FILE *`*stream*`);` (und auch als Makro definiert)

`int ferror(FILE *`*stream*`);` (und auch als Makro definiert)

`void lockfile(FILE *`*stream*`);`

`int fflush(FILE *`*stream*`);`

`int fgetc(FILE *`*stream*`);`

`int fgetpos(FILE *`*stream*`, fpos_t *`*pos*`);`

`int fgetpos64(FILE *`*stream*`, fpos64_t *`*pos*`);`

`char *fgets(char *`*s*`, int` *n*`, FILE *`*stream*`);`

`int fileno(FILE *`*stream*`);` (und auch als Makro definiert)

`FILE *fopen(const char *`*filename*`, const char *`*mode*`);`

```
FILE *fopen64(const char *filename, const char *mode);
int fprintf(FILE *stream, const char *format, ...);
int fputc(int c, FILE *stream);
int fputs(char *s, FILE *stream);
size_t fread(void *ptr, size_t size, size_t nitems, FILE *stream);
FILE *freopen(const char *filename, const char *mode, FILE *stream);
FILE *freopen64(const char *filename, const char *mode, FILE *stream);
int fscanf(FILE *stream, const char *format, ...);
int fseek(FILE *stream, long int offset, int whence);
int fseek64(FILE *stream, long int offset, int whence);
int fsetpos(FILE *stream, const fpos_t *pos);
int fsetpos64(FILE *stream, const fpos64_t *pos);
long int ftell(FILE *stream);
off64_t ftello64(FILE *stream)
void ftrylockfile(FILE *stream);
void funlockfile(FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nitems, FILE *stream);
int getc(FILE *stream); (und auch als Makro definiert)
int getc_unlocked (FILE *stream); (und auch als Makro definiert)
int getchar(void); (und auch als Makro definiert)
int getchar_unlocked(void); (und auch als Makro definiert)
int getopt(int argc, char * const argv[], const char *optstring);
char *gets(char *s);
int getw(FILE *stream);
int pclose(FILE *stream);
void perror(const char *s);
FILE *popen(const char *command, const char *type);
int printf(const char *format, ...);
int putc(int c, FILE *stream); (und auch als Makro definiert)
```

```
int fputc_unlocked(int c, FILE *stream);
```
(und auch als Makro definiert)

```
int putchar(int c);
```
(und auch als Makro definiert)

```
int fputchar_unlocked(int c);
```
(und auch als Makro definiert)

```
int puts(const char *s);
int putw(int w, FILE *stream);
int remove(const char *path);
int rename(const char *old, const char *new);
void rewind(FILE *stream);
int scanf(const char *format, ...);
void setbuf(FILE *stream, char *buf);
int setvbuf(FILE *stream, char *buf, int type, size_t size);
int sprintf(char *s, const char *format, ...);
int sscanf(const char *s, const char *format, ...);
char *tempnam(const char *dir, const char *pfx);
FILE *tmpfile(void);
FILE *tmpfile64(void);
char *tmpnam(char *s);
int ungetc(int c, FILE *stream);
int vfprintf(FILE *stream, const char *format, va_list ap);
int vprintf(const char *format, va_list ap);
int vsprintf(char *s, const char *format, va_list ap);
```

Folgende externe Variablen sind definiert:

```
extern char *optarg;
```
(Wird zukünftig vom X/Open-Standard nicht mehr unterstützt.)

```
extern int opterr;
```
(Wird zukünftig vom X/Open-Standard nicht mehr unterstützt.)

```
extern int optind;
```
(Wird zukünftig vom X/Open-Standard nicht mehr unterstützt.)

```
extern int optopt;
```
(Wird zukünftig vom X/Open-Standard nicht mehr unterstützt.)

Hinweis Wenn `stdio.h` eingebunden wird, können die in `stddef.h` definierten Symbole ebenfalls sichtbar sein.

BS2000
Aus Kompatibilitätsgründen mit Vorgängerversionen des C-Laufzeitsystems werden gleichzeitig mit dem Einbinden von `stdio.h` folgende Include-Dateien eingebunden:

```
stdio.common.h
stdio.bs21.h
stdio.bs22.h
iobuf.h ❑
```

Siehe auch `clearerr()`, `ctermid()`, `cuserid()`, `fclose()`, `fdopen()`, `feof()`, `ferror()`, `fflush()`, `fgetc()`, `fgetpos()`, `fgets()`, `fileno()`, `fopen()`, `fputc()`, `fputs()`, `fread()`, `freopen()`, `fseek()`, `fsetpos()`, `ftell()`, `fwrite()`, `getc()`, `getc_unlocked()`, `getchar()`, `getchar_unlocked( )`, `getopt()`, `gets()`, `getw()`, `getwchar()`, `getws()`, `pclose()`, `perror()`, `popen()`, `printf()`, `putc()`, `putc_unlocked()`, `putchar_unlocked()`, `puts()`, `putw()`, `putwchar()`, `remove()`, `rename()`, `rewind()`, `scanf()`, `setbuf()`, `setvbuf()`, `sscanf()`, `stdin`, `system()`, `tempnam()`, `tmpfile()`, `tmpnam()`, `ungetc()`, `vprintf()`, `stdarg.h`, `stddef.h`, `sys/types.h`.

stdlib.h - Definitionen für die Standardbibliothek

Syntax #include <stdlib.h>

Beschreibung

Folgende Konstanten sind definiert:

EXIT_FAILURE	Fehlerhafte Beendigung für exit() ergibt einen Wert ≠ 0.
EXIT_SUCCESS	Erfolgreiche Beendigung für exit() ergibt den Wert 0.
NULL	Nullzeiger
RAND_MAX	Maximalwert für den Returnwert von rand(), mindestens 32 767.
MB_CUR_MAX	Maximale Byte-Anzahl in einem Zeichen, das durch die aktuelle Lokalität spezifiziert ist.

Folgende Datentypen sind durch typedef definiert:

div_t	Strukturtyp, der von der div-Funktion zurückgegeben wird.
ldiv_t	Strukturtyp, der von der ldiv-Funktion zurückgegeben wird.
size_t	siehe stddef.h
wchar_t	siehe stddef.h

Zusätzlich werden folgende symbolische Namen und Makros definiert wie in sys/wait.h, um den Returnwert von system() zu entschlüsseln:

```
WNOHANG
WUNTRACED
WEXITSTATUS()
WIFEXITED()
WIFSIGNALED()
WIFSTOPPED()
WSTOPSIG()
WTERMSIG()
```

Folgende Namen sind als Funktionen deklariert:

```
void abort(void);
int abs(int i);
int atexit(void (*func)(void));
double atof(const char *str);
int atoi(const char *str);
long int atol(const char *str);
void *bsearch(const void *key, const void *base, size_t nel,
size_t width, int (*compar)(const void *,
const void *));
void *calloc(size_t nelem, size_t elsize);
div_t div(int numer, int denom);
double drand48(void);
double erand48(unsigned short int xsubi[3]);
void exit(int status);
void free(void *ptr);
char *getenv(const char name);
long int jrand48(unsigned short int xsubi[3]);
long int labs(long int j);
void lcong48(unsigned short int param[7]);
ldiv_t ldiv(long int numer, long int denom);
long int lrand48(void);
void *malloc(size_t size);
int mblen(const char *s, size_t n);
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
int mbtowc(wchar_t *pwc, const char *s, size_t n);
long int mrand48(void);
long int nrand48(unsigned short int xsubi[3]);
int putenv(const char *str);
void qsort(void *base, size_t nel, size_t width,
int (*compar)(const void *, const void *));
int rand(void);
void *realloc(void *ptr, size_t size);
unsigned short int *seed48(unsigned short int seed16v[3]);
void setkey(const char *key);
void srand(unsigned int seed);
void srand48(long int seedval);
double strtod(const char *str, char **ptr);
long int strtol(const char *str, char **ptr, int base);
unsigned short int strtoul(const char *str, char **ptr, int base);
int system(const char *str);
int unlockpt(int n);
size_t wcstombs(const char *str, const wchar_t *pwcs, size_t n);
```

```
int wctomb(char *s, wchar_t wchar);
```

Hinweis Das Einbinden von `stdlib.h` kann auch alle Symbole aus `stddef.h`, `limits.h`, `math.h` und `sys/wait.h` sichtbar machen.

Siehe auch `abort()`, `abs()`, `atexit()`, `atof()`, `atoi()`, `atol()`, `bsearch()`, `calloc()`, `div()`, `drand48()`, `erand48()`, `exit()`, `free()`, `getenv()`, `jrand48()`, `labs()`, `lcong48()`, `ldiv()`, `lrand48()`, `malloc()`, `mblen()`, `mbstowcs()`, `mbtowc()`, `mrand48()`, `putenv()`, `qsort()`, `rand()`, `realloc()`, `srand()`, `srand48()`, `strtod()`, `strtol()`, `strtoul()`, `wcstombs()`, `wctomb()`, `stddef.h`, `limits.h`, `math.h`, `sys/wait.h`.

string.h - Zeichenketten-Operationen

Syntax #include <string.h>

Beschreibung

`string.h` definiert folgende symbolische Namen:

`NULL`	Nullzeiger
`size_t`	Siehe `stddef.h`.

Folgende Namen sind als Funktionen deklariert:

```
void *memccpy(void *s1, const void *s2, int c, size_t n);
void *memchr(const void *s, int c, size_t n);
int memcmp(const void *s1, const void *s2, int c, size_t n);
void *memcpy(void *s1, const void *s2, int c, size_t n);
void *memmove(void *s1, const void *s2, int c, size_t n);
void *memset(void *s, int c, size_t n);
char *strcat(char *s1, const char *s2);
char *strchr(const char *s, int c);
int strcmp(const char *s1, const char *s2);
int strcoll(const char *s1, const char *s2);
char *strcpy(char *s1, const char *s2);
size_t strcspn(const char *s1, const char *s2);
char *strerror(int errnum);
size_t strlen(const char *s);
char *strncat(char *s1, const char *s2, size_t n);
int strncmp(const char *s1, const char *s2, size_t n);
char *strncpy(char *s1, const char *s2, size_t n);
char *strpbrk(const char *s1, const char *s2);
char *strrchr(const char *s, int c);
size_t strspn(const char *s1, const char *s2);
char *strstr(const char *s1, const char *s2);
char *strtok(char *s1, const char *s2);
char *strtok_r(char *s1, const char *s2);
size_t strxfrm(char *s1, const char *s2, size_t n);
```

Hinweis Das Einbinden der Include-Datei `string.h` kann alle Symbole der Include-Datei `stddef.h` sichtbar machen.

Siehe auch `memccpy()`, `memchr()`, `memcmp()`, `memcpy()`, `memmove()`, `memset()`, `strcat()`, `strchr()`, `strcmp()`, `strcoll()`, `strcpy()`, `strcspn()`, `strerror()`, `strlen()`, `strncat()`, `strncmp()`, `strncpy()`, `strpbrk()`, `strrchr()`, `strspn()`, `strstr()`, `strtok()`, `strtok_r()`, `strxfrm()`, `sys/types.h`.

strings.h - Zeichenketten-Operationen

Syntax #include <strings.h>

Beschreibung

Die folgenden Funktionen sind deklariert:

```
int     bcmp(const void *s1, const void *s2, size_t n);
void    bcopy(const void *s1, void *s2, size_t n);
void    bzero(void *s, size_t n);
int     ffs(int i);
char   *index(const char *s, int c);
char   *rindex(const char *s, int c);
int     strcasecmp(const char *s1, const char *s2);
int     strncasecmp(const char *s1, const char *s2, size_t n);
```

Siehe auch `bcmp()`, `bcopy()`, `bzero()`, `ffs()`, `index()`, `rindex()`, `strcasecmp()`.

stropts.h - STREAMS-Schnittstelle

Syntax #include <stropts.h>

Beschreibung

Der Header `stropts.h` definiert die folgenden Strukturen:

Die Struktur `bandinfo` für die Prioritätsklassen I_FLUSHBAND der Funktion `ioctl()`:

```
struct bandinfo {
nsigned char bi_pri;
int bi_flag;
};
```

Die Struktur `strpeek` für das Stream-Format I_PEEK der Funktion `ioctl()`:

```
struct strpeek {
struct strbuf ctlbuf;
struct strbuf databuf;
long flags;
};
```

Die Struktur `strbuf` für `putmsg`- und `getmsg`-Systemaufrufe:

```
struct strbuf {
int maxlen;                          /* maximale Länge des Puffers */
int len;                             /* Datenlänge */
char *buf;                           /* Zeiger auf den Puffer */
};
```

Die Struktur `strfdinsert` für das Stream-Format I_FDINSERT der Funktion `ioctl()`:

```
struct strfdinsert {
sstruct strbuf ctlbuf;
struct strbuf databuf;
long flags;
int fildes;
int offset;
};
```

Die Struktur `strioctl` für Benutzerformate der Funktion `ioctl()`, die stromabwärts gehen (I_STR):

```
struct strioctl {
int ic_cmd;                            /* Kommando */
int ic_timeout;                        /* Timeout-Wert */
int ic_len;                            /* Datenlänge */
char *ic_dp;                           /* Zeiger auf die Daten */
};
```

Die Struktur strrecvfd für den Kompatibilitäts-Modus des Benutzers. Es wird EOVERFLOW zurückgegeben, wenn uid oder gid das ushort-Limit überschreitet:

```
struct strrecvfd {
int fd;
o_uid_t uid;


o_uid_t gid;
char fill[8];
};
```

Die Strukturen str_mlist und str_list für I_LIST der Funktion ioctl():

```
struct str_mlist {
char l_name[FMNAMESZ+1];
};

struct str_list {
int sl_nmods;
struct str_mlist *sl_modlist;
};
```

Als Argument *request* für ioctl() sind die folgenden Makros definiert:

I_PUSH	Schiebt das STREAMS-Modul ganz oben auf den aktuellen STREAM, gleich unterhalb des STREAM-Head.
I_POP	Entfernt das STREAMS-Modul gleich unterhalb des STREAM-Head.
I_LOOK	Ruft den Namen des Moduls gleich unterhalb des STREAM-Head ab und legt ihn in einer Zeichenkette ab. Als Argument *arg* ist mindestens das folgende Makro definiert: FMNAMESZ Mindestgröße des Puffers in Byte, der vom Argument *arg* angesprochen wird.
I_FLUSH	Löscht alle Input- bzw. Output-Warteschlangen (in Abhängigkeit vom Wert des Arguments *arg*). Als Argument *arg* sind mindestens die folgenden Makros definiert: FLUSHR Löscht die Lese-Warteschlangen. FLUSHW Löscht die Schreib-Warteschlangen. FLUSHRW Löscht die Lese- und Schreib-Warteschlangen.
I_FLUSHBAND	Löscht nur das angegebene Band.
I_SETSIG	Informiert den STREAM-Head, dass der Prozess das Signal SIGPOLL abgesetzt haben will (siehe signal() und sigset()), wenn im Rahmen des STREAM ein bestimmtes Ereignis eingetreten ist.

Die Include-Datei `stropts.h` definiert die folgenden Werte für *arg*, wenn `I_SETSIG` angegeben wird:

`S_RDNORM`	Normale Nachricht (Prioritätsklasse = 0) ist am Anfang einer Lese-Warteschlange des `STREAM`-Head angekommen.
`S_RDBAND`	Eine Nachricht (Prioritätsklasse ungleich 0) ist am Anfang der Lese-Warteschlange des `STREAM`-Head angekommen.
S_INPUT	Eine Nachricht ohne hohe Priorität ist am Anfang einer Lese-Warteschlange des `STREAM`-Head angekommen.
`S_HIPRI`	Eine Nachricht mit hoher Priorität liegt in einer Lese-Warteschlange des `STREAM`-Head vor.
`S_OUTPUT`	Die Schreib-Warteschlange für normale Daten (Prioritätsklasse 0) gleich unterhalb des `STREAM`-Head ist nicht mehr voll. Dies teilt dem Prozess mit, dass in der Warteschlange Platz für das Senden (oder Schreiben) normaler Daten vorhanden ist.
`S_WRNORM`	Wie `S_OUTPUT`.
`S_WRBAND`	Die Schreib-Warteschlange für eine Prioritätsklasse ungleich 0 gleich unterhalb des `STREAM`-Head ist nicht mehr voll.
`S_MSG`	Eine `STREAMS`-Signalnachricht mit dem Signal `SIGPOLL` erreicht den Anfang der Lese-Warteschlange des `STREAM`-Head.
`S_ERROR`	Eine Nachricht über eine Fehlerbedingung erreicht den `STREAM`-Head.
`S_HANGUP`	Die Nachricht über ein Hangup erreicht den `STREAM`-Head.
`S_BANDURG`	Bei Verwendung zusammen mit `S_RDBAND` wird `SIGURG` statt `SIGPOLL` generiert, wenn eine Nachricht mit Priorität den Anfang der Lese-Warteschlange des `STREAM`-Head erreicht.

`I_GETSIG` Gibt die Ereignisse zurück, für die der aufrufende Prozess derzeit ein Signal `SIGPOLL` erhalten soll.

`I_FIND` Vergleicht die Namen aller Module, die derzeit im `STREAM` vorliegen, mit dem Namen, auf den *arg* zeigt.

`I_PEEK` Ein Prozess kann die Daten in der ersten Nachricht der Lese-Warteschlange des `STREAM`-Head abrufen, ohne dass die Nachricht aus der Warteschlange genommen wird. Als Argument *arg* ist das folgende Makro definiert:

`RS_HIPRI`	Es wird nur nach Nachrichten hoher Priorität gesucht.

I_SRDOPT Der Lesemodus wird gesetzt. Als Argument *arg* sind die folgenden Makros definiert:

RNORM	Byte-STREAM-Modus (Voreinstellung).
RMSGD	Message-discard-Modus.
RMSGN	Message-nondiscard-Modus.
RPROTNORM	read() mit EBADMSG schlägt fehl, wenn sich eine Nachricht mit einem Steuerteil am Anfang der Lese-Warteschlange des STREAM-Head befindet.
RPROTDAT	Der Steuerteil einer Nachricht wird als Daten geliefert, wenn ein Prozess read() absetzt.
RPROTDIS	Der Steuerteil einer Nachricht wird verworfen, die Daten werden jedoch geliefert, wenn ein Prozess read() absetzt.

I_GRDOPT Gibt die aktuelle Einstellung des Lesemodus zurück.

I_NREAD Zählt die Anzahl der Datenbytes in Datenblöcken in der ersten Nachricht der Lese-Warteschlange des STREAM-Head.

I_FDINSERT Erstellt Nachrichten vom/von den angegebenen Puffer(n), fügt Informationen über einen anderen STREAM hinzu und sendet die Nachricht stromabwärts (downstream).

I_STR Erstellt eine interne STREAMS ioctl()-Nachricht und sendet sie stromabwärts (downstream).

I_SWROPT Der Schreibmodus wird gesetzt. Als Argument *arg* ist das folgende Makro definiert:

SNDZERO	Sendet eine Nachricht der Länge 0 downstream, wenn ein write() mit 0 Byte eintritt.

I_GWROPT Gibt die aktuelle Einstellung des Schreibmodus zurück.

I_SENDFD Fordert an, dass der STREAM, der mit *fd* verknüpft ist, eine Nachricht mit einem Dateizeiger an den STREAM-Head am anderen Ende einer STREAMS-Pipe sendet.

I_RECVFD Ruft den Dateideskriptor ab, der mit der Nachricht verknüpft ist, die von einem I_SENDFD ioctl() über eine STREAMS-Pipe gesendet wurde.

I_LIST Der Prozess kann alle Modulnamen des STREAM auflisten (bis einschließlich des obersten Treibernamens).

I_ATMARK Der Prozess kann sehen, ob die aktuelle Nachricht in der Lese-Warteschlange des STREAM-Head von einem Modul stromabwärts (downstream) "markiert" wurde. Als Argument *arg* sind die folgenden Makros definiert:

ANYMARK	Prüft, ob die Nachricht markiert ist.
LASTMARK	Prüft, ob die Nachricht die letzte markierte Nachricht in der Warteschlange ist.

I_CKBAND Prüft, ob eine Nachricht einer bestimmten Prioritätsklasse in der Lese-Warteschlange des STREAM-Head vorhanden ist.

I_GETBAND	Gibt die Prioritätsklasse der ersten Nachricht in der Lese-Warteschlange des STREAM-Head zurück.
I_CANPUT	Prüft, ob auf für eine bestimmte Prioritätsklasse geschrieben werden kann.
I_SETCLTIME	Der Prozess kann angeben, wie lange die Verzögerung für den STREAM-Head ist, wenn ein STREAM beendet wird und sich noch Daten in den Schreib-Warteschlangen befinden.
I_GETCLTIME	Gibt die Verzögerung für die Beendigung zurück.
I_LINK	Zwei STREAMs werden verbunden.
I_UNLINK	Die Verbindung der zwei STREAMs wird aufgehoben. Der Header definiert den folgenden Wert für *all*:
	MUXID_ALL Die Verknüpfung wird für alle STREAMs aufgehoben, die mit dem *fd* zugeordneten STREAM verknüpft sind.
I_PLINK	Verbindet zwei STREAMs mit einer beständigen Verknüpfung.
I_PUNLINK	Löst die Verbindung der zwei STREAMs, die mit einer persistenten Verknüpfung verbunden wurden.

Für getmsg(), getpmsg(), putmsg() und putpmsg() sind folgende Makros definiert:

MSG_ANY	Empfängt jede Nachricht.
MSG_BAND	Empfängt Nachrichten einer bestimmten Prioritätsklasse.
MSG_HIPRI	Sendet/empfängt Nachrichten hoher Priorität.
MORECTL	Es wird mehr an Steuerdaten in der Nachricht belassen.
MOREDATA	Es wird mehr an Daten in der Nachricht belassen.

stropts.h kann alle Symbole aus unistd.h sichtbar machen. Die folgenden Funktionen sind im Header stropts.h als Funktionen deklariert:

```
int  isastream(int fildes);

int  getmsg(int fd, struct strbuf *ctlptr, struct strbuf *dataptr,
            int *flagsp);

int  getpmsg(int fd, struct strbuf *ctlptr, struct strbuf *dataptr,
             int *bandp, int *flagsp);

int  ioctl(int fildes, int request, ... );

int  putmsg(int fd, const struct strbuf *ctlptr,
             const struct strbuf *dataptr, int flags);

int  putpmsg(int fd, const struct strbuf *ctlptr,
             const struct strbuf *dataptr, int band, int flags);

int  fattach(int fildes, const char *path);

int  fdetach(const char *path);
```

Siehe auch close(), fcntl(), getmsg(), ioctl(), open(), pipe(), read(), poll(), putmsg(), signal(), sigset(), write().

sys/ipc.h - Strukturen für Interprozesskommunikation

Syntax #include <sys/ipc.h>

Beschreibung

Die Include-Datei `sys/ipc.h` verwendet drei Mechanismen für die Interprozesskommunikation (IPC): Nachrichten (messages), Semaphore und gemeinsam genutzten Hauptspeicher (shared memory). Alle verwenden einen gemeinsamen Strukturtyp `ipc_perm`, um die Informationen zu übergeben, die bei der Festlegung der Berechtigung zur Ausführung von IPC-Operationen benutzt werden.

Die Struktur `ipc_perm` enthält die folgenden Komponenten:

`uid_t`	`uid`	Benutzernummer des Eigentümers
`gid_t`	`gid`	Gruppennummer des Eigentümers
`uid_t`	`cuid`	Benutzernummer des Erzeugers
`gid_t`	`cgid`	Gruppennummer des Erzeugers
`mode_t`	`mode`	Lese- und Schreibrechte

Die folgenden Konstanten werden definiert:

Zustandsbits:

`IPC_CREAT`	erzeuge Eintrag, wenn Schlüssel nicht existiert
`IPC_EXCL`	fehlschlagen, wenn Schlüssel existiert
`IPC_NOWAIT`	Fehler, wenn die Anforderung warten muss

Schlüssel:

`IPC_PRIVATE`	privater Schlüssel

Steuer-Kommandos:

`IPC_RMID`	Identifikator entfernen
`IPC_SET`	Optionen setzen
`IPC_STAT`	Optionen ermitteln

syslog.h - Definitionen zum Protokollieren von System-Fehlermeldungen

Syntax #include <syslog.h>

syslog.h definiert nachfolgende symbolische Konstanten. Durch ODER-Verknüpfungen dieser Konstanten in beliebiger Kombination wird die logopt Option von openlog() gebildet:

LOG_PID	Protokolliert die Prozess-ID mit jeder Meldung.
LOG_CONS	Gibt Fehlermeldungen auf der Systemkonsole aus.
LOG_NDELAY	Öffnet die Verbindung zu syslog sofort.
LOG_ODELAY	Verzögert das Öffnen der Verbindung so lange, bis syslog() aufgerufen wird.
LOG_NOWAIT	Wartet nicht auf Kindprozesse.

Die folgenden symbolischen Konstanten sind als Werte des Parameters *facility* von openlog() definiert:

LOG_KERN	Reserviert für Systemmeldungen.
LOG_USER	Von Benutzerprozessen erstellte Meldungen.
LOG_MAIL	Reserviert für Meldungen des Postsystems.
LOG_NEWS	Reserviert für Meldungen des USENET Netzwerknachrichtensystems.
LOG_UUCP	Reserviert für Meldungen des UUCP−Systems.
LOG_DAEMON	Reserviert für Meldungen der Systemdämone.
LOG_AUTH	Reserviert für Meldungen des Berechtigungssystems ("dämon").
LOG_CRON	Reserviert für Meldungen des "clock-Dämons".
LOG_LPR	Reserviert für Meldungen des Druckersystems.
LOG_LOCAL0	Reserviert für lokale Verwendung.
LOG_LOCAL1	Reserviert für lokale Verwendung.
LOG_LOCAL2	Reserviert für lokale Verwendung.
LOG_LOCAL3	Reserviert für lokale Verwendung.
LOG_LOCAL4	Reserviert für lokale Verwendung.
LOG_LOCAL5	Reserviert für lokale Verwendung.
LOG_LOCAL6	Reserviert für lokale Verwendung.
LOG_LOCAL7	Reserviert für lokale Verwendung.

Die folgenden Makros dienen dazu, den Parameter *maskpri* von setlogmask() darzustellen.

LOG_MASK(*pri*) Berechnet die Maske für eine bestimmte Priorität *pri*.

LOG_UPTO(*pri*) Gibt die Maske für alle Prioritäten bis *pri* an.

Die folgenden Makros sind als Werte des Parameters *priority* von `syslog()` definiert:

`LOG_EMERG`	"panic"-Bedingung, die an alle Benutzerprozesse gemeldet wird.
`LOG_ALERT`	Bedingung, die sofort korrigiert werden sollte (z.B. beschädigte Systemdatenbank).
`LOG_CRIT`	schwerwiegende Bedingung (z.B. Festplattenfehler).
`LOG_ERR`	Fehlermeldung.
`LOG_WARNING`	Warnmeldung.
`LOG_NOTICE`	Bedingung, die spezielle Schritte erfordert.
`LOG_INFO`	Informationsmeldung.
`LOG_DEBUG`	Meldung mit Information, die beim Testen eines Programmes verwendet wird.

Die folgenden Funktionen sind definiert:

```
void  closelog(void);
void  openlog(const char *id, int logopt, int facility);
int   setlogmask(int maskpri);
void  syslog(int priority, const char *format, ...);
```

Siehe auch `closelog()`.

sys/mman.h - Definitionen zur Speicherverwaltung

Syntax #include <sys/mman.h>

Beschreibung

Es gibt folgende Schutz-Optionen:

PROT_READ	Seite kann gelesen werden.
PROT_WRITE	Seite kann geschrieben werden.
PROT_EXEC	Seite kann ausgeführt werden.
PROT_NONE	auf Seite kann nicht zugegriffen werden.

Folgende flag Optionen sind definiert:

MAP_SHARED	Änderungen gemeinsam benutzbar machen.
MAP_PRIVATE	Änderungen privat halten.
MAP_FIXED	*addr* exakt interpretieren.

Für msync() sind folgende Flags (Bitmuster) definiert:

MS_ASYNC	asynchrone Schreibzugriffe durchführen.
MS_SYNC	synchrone Schreibzugriffe durchführen.
MS_INVALIDATE	Abbildungen (Verweise) ungültig machen.

Die Datentypen size_t und off_t sind in sys/types.h definiert und beschrieben.
Folgende Funktionen sind in sys/mman.h deklariert:

```
void *mmap(void *addr, size_t len, int prot, int flags, int fd, off_t off);
int   mprotect(void *addr, size_t len, int prot);
int   msync(void * addr, size_t len, int flags);
int   munmap(void *addr, size_t len);
```

Siehe auch mmap(), mprotect(), msync(), munmap().

sys/msg.h - Strukturen für Nachrichten-Warteschlangen

Syntax #include <sys/msg.h>

Beschreibung

Die Include-Datei `sys/msg.h` definiert die folgenden Konstanten und Komponenten der Struktur `msqid_ds`.

Kennzeichen für Nachrichtenbearbeitung:

`MSG_NOERROR`	kein Fehler, wenn große Nachricht

Die Struktur `msqid_ds` enthält die folgenden Komponenten:

`struct ipc_perm`	`msg_perm`	Struktur für Erlaubnis der Operation
`unsigned short`	`msg_qnum`	aktuelle Anzahl der Nachrichten in der Warteschlange
`unsigned short`	`msg_qbytes`	Maximalzahl der erlaubten Bytes in der Warteschlange
`pid_t`	`msg_lspid`	Prozessnummer des letzten `msgsnd()`
`pid_t`	`msg_lrpid`	Prozessnummer des letzten `msgrcv()`
`time_t`	`msg_stime`	Zeit des letzten `msgsnd()`
`time_t`	`msg_rtime`	Zeit des letzten `msgrcv()`
`time_t`	`msg_ctime`	Zeit der letzten Änderung

Die folgenden Namen sind als Funktionen deklariert:

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);

int msgget(key_t key, int msgflg);

int msgrcv(int msqid, void *msgq, size_t msgsz, long int msgtyp,
           int msgflg);

int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

Zusätzlich werden alle Symbole aus `sys/ipc.h` definiert, wenn `sys/msg.h` in den Quelltext einbezogen wird.

Siehe auch `msgctl()`, `msgget()`, `msgrcv()`, `msgsnd()`, `sys/types.h`.

sys/resource.h - Definitionen der Operationen für XSI Ressourcen

Syntax #include <sys/resource.h>

Beschreibung

sys/resource.h definiert die folgenden symbolischen Konstanten als mögliche Werte des Arguments *which* der Funktionen getpriority() und setpriority():

PRIO_PROCESS	identifiziert *who* als Prozess ID.
PRIO_PGRP	identifiziert *who* als Prozessgruppen ID.
PRIO_USER	identifiziert *who* als Benutzer ID.

Nachstehende symbolische Konstante ist folgendermaßen definiert:

RLIM_INFINITY	rlim_t besitzt einen unendlich großen Wert des Typs rlim_t.
RLIM64_INFINITY	rlim64_t besitzt einen unendlich großen Wert des Typs rlim64_t.
RLIM_SAVED_MAX	rlim_t besitzt einen unendlich großen Wert des Typs rlim_t.
RLIM64_SAVED_MAX	rlim64_t besitzt einen unendlich großen Wert des Typs rlim64_t.
RLIM_SAVED_CUR	Wert des Typs rlim_t.
RLIM64_SAVED_CUR	Wert des Typs rlim64_t.

Bei Implementierungen, in denen sich alle Ressourcen-Grenzen in einem Objekt des Typs rlim64_t darstellen lassen, müssen sich RLIM64_SAVED_MAX und RLIM64_SAVED_CUR nicht von RLIM64_INFINITY unterscheiden.

Die folgenden symbolischen Konstanten sind mögliche Werte des *who* Arguments der Funktion getrusage():

RUSAGE_SELF	Gibt Informationen über den aktuellen Prozess aus.
RUSAGE_CHILDREN	gibt Informationen über die Kindprozesse des aktuellen Prozesses aus.

rlim_t wird durch typedef als vorzeichenloser, ganzzahliger Datentyp definiert, der für Grenzwerte verwendet wird:

```
struct rlimit {
       rlim_t rlim_cur;                    /* aktuelles Limit */
       rlim_t rlim_max;                    /* Maximalwert für rlim_cur */
};
```

`rlim64_t` wird durch `typedef` als erweiterter, vorzeichenloser, ganzzahliger Datentyp definiert, der jede nicht-negative Zahl des Wertebereichs eines off64_t-Objekts darstellen kann und für Grenzwerte verwendet wird:

```
struct rlimit64 {
       rlim64_t rlim_cur;                  /* aktuelles (weiches) Limit */
        rlim64_t rlim_max;                      /* Maximalwert für rlim_cur*/
```

`rusage` wird folgendermaßen definiert:

```
struct rusage {
       struct timeval ru_utime;    /* verbrauchte Zeit im Benutzermodus */
       struct timeval ru_stime;    /* verbrauchte Zeit im Systemmodus */
       long ru_maxrss;
#define ru_first      ru_ixrss
        long ru_ixrss;            /* Größe gemeinsam benutzbarer Speicher */
        long ru_idrss;            /* Größe nicht gemeinsam benutzte Daten */
        long ru_isrss;            /* Größe nicht gemeinsam benutzter Stack */
        long ru_minflt;           /* Seitenfehler ohne Ein-/Ausgabe */
        long ru_mafflt;           /* Seitenfehler mit Ein-/Ausgabe */
        long ru_nswap;            /* swaps */
        long ru_inblock;          /* Block-Eingabeoperationen */
        long ru_oublock;          /* Block-Ausgabeoperationen */
        long ru_msgsnd;           /* gesendete Nachrichten */
        long ru_msgrcv;           /* empfangene Nachrichten */
        long ru_nsignals;         /* empfangene Signale*/
        long ru_nvcsw;            /* freiwillige Kontext-Wechsel */
        long ru_nivcsw;           /* unfreiwillige Kontext-Wechsel */
        long ru_totcsw;           /* Summe der Kontext-Wechsel */
/* Pyramid-Additions */
        long ru_zerofill;         /*  mit Nullen aufzufüllende Seiten */
        long ru_pffincr;          /* Anzahl Erhöhungen RSS */
        long ru_pffdecr;          /* Anzahl Verminderungen RSS */
        long ru_syscall;          /* Anzahl der Systemaufrufe */
        long ru_lread;            /* Lese-Aufrufe des Systems */
        long ru_lwrite;           /* Schreib-Aufrufe des Systems */
        long ru_phread;           /* Raw-Lesezugriffe */
        long ru_phwrite;          /* Raw-Schreibzugriffe */
#define ru_last        ru_write
        long ru_spare[5];         /* Aufrunden auf 32-Bitlong-Werte */
};
```

Die Struktur `timeval` ist in `sys/time.h` beschrieben.

Die folgenden symbolischen Konstanten sind mögliche Werte des *resource* Arguments der Funktionen `getrlimit()` und `setrlimit()`:

`RLIMIT_CORE`	Maximale Größe einer Speicherabzugsdatei.
`RLIMIT_CPU`	Maximale CPU-Zeit, die von einem Prozess verwendet werden darf.
`RLIMIT_DATA`	Maximale Größe des Heap-Speichers eines Prozesses.
`RLIMIT_FSIZE`	Maximale Länge einer Datei.
`RLIMIT_NOFILE`	Maximale Anzahl der geöffneten Dateideskriptoren, die ein Prozess besitzen kann.
`RLIMIT_STACK`	Maximale Größe des Prozessstapels.
`RLIMIT_AS`	Maximale Länge des Adressbereiches eines Prozesses.

Die folgenden Funktionen sind definiert:

```
int  getpriority(int which, id_t who);
int  getrlimit(int resource, struct rlimit *rlp);
int  getrlimit64(int resource, struct rlimit64 *rlp);
int  getrusage(int who, struct rusage *r_usage);
int  setpriority(int which, id_t who, int priority);
int  setrlimit(int resource, const struct rlimit *rlp);
int  setrlimit64(int resource, const struct rlimit64 *rlp);
```

Die Include-Datei `sys/resource.h` kann auch alle Inhalte von `sys/time.h` sichtbar machen.

Siehe auch `getpriority()`, `getrusage()`, `getrlimit()`.

sys/sem.h - Semaphor-Strukturen

Syntax #include <sys/sem.h>

Beschreibung

Die Include-Datei `sys/sem.h` definiert die folgenden Konstanten und Strukturen.

Kennzeichen für Semaphor-Operationen:

`SEM_UNDO`	automatische Freigabe von Semaphoren bei Prozessende

Kommandodefinitionen für die Funktion `semctl()`:

`GETNCNT`	`semcnt` ermitteln
`GETPID`	`sempid` ermitteln
`GETVAL`	`semval` ermitteln
`GETALL`	alle Fälle von `semval` ermitteln
`GETZCNT`	`semzcnt` ermitteln
`SETVAL`	`semval` setzen
`SETALL`	alle Fälle von `semval` setzen

Die Struktur `semid_ds` enthält die folgenden Komponenten:

`struct ipc_perm`	`sem_perm`	Berechtigungsstruktur
`unsigned short`	`sem_nsems`	Anzahl der Semaphore in der Menge
`time_t`	`sem_otime`	letzte Ausführung von `semop()`
`time_t`	`sem_ctime`	letzte Änderung durch `semctl()`

Die Datentypen `pid_t`, `time_t` und `size_t` sind definiert wie in `sys/types.h` beschrieben.

Die Anzahl der Semaphore in einer Menge ist `sem_nsems`, innerhalb der Semaphormenge wird von 0 bis `sem_nsems`-1 durchnummeriert. Die Nummer eines Semaphors heißt `sem_num`.

Ein Semaphor wird durch eine unbenannte Struktur repräsentiert, die die folgenden Komponenten enthält:

`unsigned short`	`semval`	Semaphorwert
`pid_t`	`sempid`	Prozessnummer der letzten Operation
`unsigned short`	`semncnt`	Anzahl der Prozesse, die darauf warten, dass *semval* größer wird als es zurzeit ist
`unsigned short`	`semzcnt`	Anzahl der Prozesse, die darauf warten, dass *semval* gleich 0 wird

Die Struktur *sembuf* enthält die folgenden Komponenten:

`unsigned short`	`sem_num`	Semaphornummer
`short`	`sem_op`	Semaphoroperation
`short`	`sem_flg`	Operationskennzeichen

Die folgenden Namen sind als Funktionen deklariert:

`int semctl(int` ***semid***`, int` ***semnum***`, int` ***cmd***`, ...);`

`int semget(key_t` ***key***`, int` ***nsems***`, int` ***semflg***`);`

`int semop(int` ***semid***`, struct sembuf *`***sops**`, size_t` ***nsops***`);`

Zusätzlich werden alle Symbole aus `sys/ipc.h` definiert, wenn `sys/sem.h` in den Quelltext einbezogen wird.

Siehe auch `semctl()`, `semget()`, `semop()`, `sys/types.h`.

sys/shm.h - Definitionen für Shared Memory

Syntax #include <sys/shm.h>

Beschreibung

Die Include-Datei `sys/shm.h` definiert die folgenden Konstanten und eine Struktur.

Schalter für Operationen:

`SHM_RDONLY`	nur zum Lesen anfügen (sonst: Lesen und Schreiben)
`SHMLBA`	Vielfaches der Adresse der Segmentuntergrenze
`SHM_RND`	Anhängadresse auf `SHMLBA` aufrunden

Der folgende Datentyp wird durch `typedef` definiert:

`shmatt_t`	Ganzzahl ohne Vorzeichen, benutzt für die Anzahl der aktuell angehängten Bereiche `shm_nattch`, in der Werte gespeichert werden müssen, die zumindest dem Wertebereich des Datentyps `unsigned integer` entsprechen.

Die Datentypen `pid_t`, `time_t` und `size_t` sind definiert wie in `sys/types.h` beschrieben.

Die Struktur `shmid_ds` enthält die folgenden Komponenten:

`struct ipc_perm`	`shm_perm`	Berechtigungsstruktur
`int`	`shm_segsz`	Größe des Segments in Bytes
`pid_t`	`shm_lpid`	Prozessnummer der letzten Operation
`pid_t`	`shm_cpid`	Prozessnummer des Erzeugers
`shmatt_t`	`shm_nattch`	Anzahl der aktuell angehängten Bereiche
`time_t`	`shm_atime`	Zeit des letzten `shmat()`
`time_t`	`shm_dtime`	Zeit des letzten `shmdt()`
`time_t`	`shm_ctime`	Zeit der letzten Änderung durch `shmctl()`

Die folgenden Namen sind als Funktionen deklariert:

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
int shmctl(int shmid, int cmd, struct shmid_s *buf);
int shmdt(const void *shmaddr);
int shmget(key_t key, size_t size, int shmflg);
```

Zusätzlich werden alle Symbole aus `sys/ipc.h` definiert, wenn `sys/shm.h` in den Quelltext einbezogen wird.

Siehe auch `shmat()`, `shmctl()`, `shmdt()`, `shmget()`, `sys/types.h`.

sys/stat.h - Daten für den Dateistatus

Syntax #include <sys/stat.h>

Beschreibung

Die Include-Datei `sys/stat.h` definiert die Struktur, die von den Funktionen `stat()` und `fstat()` zurückgeliefert wird.

Die Struktur `stat` enthält mindestens folgende Komponenten:

Datentyp	Strukturkomponente	Bedeutung
`dev_t`	`st_dev`	Nummer des Geräts, das die Datei enthält
`ino_t`	`st_ino`	Dateinummer
`ino64_t`	`st_ino`	Dateinummer
`mode_t`	`st_mode`	Dateityp (siehe auch unten)
`nlink_t`	`st_nlink`	Anzahl der Verweise
`uid_t`	`st_uid`	Benutzernummer des Eigentümers
`gid_t`	`st_gid`	Gruppennummer des Eigentümers
`dev_t`	`st_rdev`	Gerätenummer (falls Gerätedatei)
`off_t`	`st_size`	Dateigröße in Byte (nur normale Datei)
`off64_t`	`st_size`	Dateigröße in Byte
`time_t`	`st_atime`	letzte Zugriffszeit
`time_t`	`st_mtime`	letzte Modifikationszeit
`time_t`	`st_ctime`	Zeit der letzten Zustandsänderung
`blkcnt_t`	`stblocks`	Anzahl der für das Objekt allokierten Blöcke
`blkcnt64_t`	`st_blocks`	Anzahl der für das Objekt allokierten Blöcke

Die Dateinummer und die Nummer des Geräts, das die Datei enthält, identifizieren die Datei eindeutig innerhalb eines Systems. Die Datentypen `dev_t`, `ino_t`, `nlink_t`, `uid_t`, `gid_t`, `off_t` und `time_t` sind wie in `sys/types.h` beschrieben definiert. Zeiten werden in Sekunden seit dem Beginn der Epoche angegeben.

Folgende symbolische Namen für Werte von st_mode werden definiert:

Symbolischer Name	Bedeutung
S_IFMT	Dateityp
S_IFBLK	Datei für blockorientiertes Gerät
S_IFCHR	Datei für zeichenorientiertes Gerät
S_IFDIR	Dateiverzeichnis
S_IFIFO	FIFO-Gerätedatei
S_IFREG	normale Datei
S_IFSOCK	Socket
S_IRWXU	Zugriffsrechte für Eigentümer
S_IRUSR	Lesen für Eigentümer
S_IWUSR	Schreiben für Eigentümer
S_IXUSR	Ausführen/Durchsuchen für Eigentümer
S_IRWXG	Zugriffsrechte für Gruppe
S_IRGRP	Lesen für Gruppe
S_IWGRP	Schreiben für Gruppe
S_IXGRP	Ausführen/Durchsuchen für Gruppe
S_IRWXO	Zugriffsrechte für Andere
S_IROTH	Lesen für Andere
S_IWOTH	Schreiben für Andere
S_IXOTH	Ausführen/Durchsuchen für Andere
S_ISUID	Benutzernummer bei Ausführung setzen (s-Bit für Eigentümer)
S_ISGID	Gruppennummer bei Ausführung setzen (s-Bit für Gruppe)

Die Bits, die durch S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH, S_IWOTH, S_IXOTH, S_ISUID and S_ISGID definiert sind, sind eindeutig.

S_IRWXU ist ein bitweises Oder von S_IRUSR, S_IWUSR und S_IXUSR.

S_IRWXG ist ein bitweises Oder von S_IRGRP, S_IWGRP und S_IXGRP.

Folgende Dateityp-Testmakros ermitteln, ob die angesprochene Datei dem erwarteten Typ entspricht. *m* entspricht dem Wert der Strukturkomponente `st_mode` aus der Struktur `stat`. Bei Entsprechung ergeben die Makros einen Wert ungleich null.

`S_ISBLK(`*m*`)` Test auf Gerätedatei für blockorientierte Geräte

`S_ISCHR(`*m*`)` Test auf Gerätedatei für zeichenorientierte Geräte

`S_ISDIR(`*m*`)` Test auf Dateiverzeichnis

`S_ISFIFO(`*m*`)` Test auf FIFO-Gerätedatei

`S_ISREG(`*m*`)` Test auf normale Datei

Folgende Namen sind als Funktionen deklariert:

```
int chmod(const char *path, mode_t mode);
int fstat(int fildes, struct stat *buf);
int mkdir(const char *path, mode_t mode);
int mkfifo(const char *path, mode_t mode);
mode_t umask(mode_t cmask);

int fstat64(int fildes, struct stat64 *buf);
int lstat64(const char *path, struct stat64 *buf);
int stat64(const char *path, struct stat64 *buf);
```

Hinweis Um den Typ einer Datei zu ermitteln, wird die Verwendung der Makros empfohlen.

sys/statvfs.h - Struktur der VFS-Dateisysteminformation

Syntax #include <sys/statvfs.h>

Beschreibung

`sys/statvfs.h` definiert die Struktur `statvfs`:

```
typedef struct statvfs {
unsigned long f_bsize;    /* Blockgröße des Dateisystems */
unsigned long f_frsize;   /* Fragmentgröße */
unsigned long f_blocks;   /* # Blöcke auf Dateisystem mit Größe f_frsize */
unsigned long f_bfree;    /* # freie Blöcke mit Größe f_frsize */
unsigned long f_bavail;   /* # verfügbare freie Blöcke für Nicht-Superuser */
unsigned long f_files;    /* # Dateiknoten (inodes) */
unsigned long f_ffree;    /* # freie Dateiknoten (inodes) */
unsigned long f_favail;   /* # verfügbare freie Inodes für Nicht-Superuser */
unsigned long f_fsid;     /* Dateisystem-Id (Nummer) */
char f_basetype[FSTYPSZ]; /* Name des Ziel-Dateisystems, Null-terminiert */
unsigned long f_flag;     /* Bitmaske der Optionen von f_flag */
unsigned long f_namemax;  /* maximale Länge der Dateinamen */
char f_fstr[32];          /* dateisystemspezifische Zeichenkette */
unsigned long f_filler[16]; /* reserviert für zukünftige Versionen */
} statvfs_t;
```

Folgende Optionen sind für die Komponente `f_flag` definiert:

`ST_RDONLY`	nur lesbares Dateisystem
`ST_NOSUID`	`setgid`/`setuid`-Semantik wird nicht unterstützt

Folgende Funktionen sind in `sys/statvfs.h` definiert:

```
int statvfs(const char *path, struct statvfs *buf);
int fstatvfs(int fildes, struct statvfs *buf);
```

`sys/statvfs.h` definiert die Struktur `statvfs`:

```
typedef struct statvfs {
unsigned long f_bsize;    /* Blockgröße des Dateisystems */
unsigned long f_frsize;   /* Fragmentgröße */
fsblkcnt_t  f_blocks;     /* # Blöcke auf Dateisystem mit Größe f_frsize */
fsblkcnt_t  f_bfree;      /* # freie Blöcke mit Größe f_frsize */
fsblkcnt_t  f_bavail;     /* # verfügbare freie Blöcke für Nicht-Superuser */
fsfilcnt_t  f_files;      /* # Dateiknoten (inodes) */
fsfilcnt_t  f_ffree;      /* # freie Dateiknoten (inodes) */
fsfilcnt_t  f_favail;     /* # verfügbare freie Inodes für Nicht-Superuser */
unsigned long f_fsid;     /* Dateisystem-Id (Nummer) */
```

```
char f_basetype[FSTYPSZ]; /* Name des Ziel-Dateisystems, Null-terminiert */
unsigned long f_flag;     /* Bitmaske der Optionen von f_flag */
unsigned long f_namemax;  /* maximale Länge der Dateinamen */
char f_fstr[32];          /* dateisystemspezifische Zeichenkette */
unsigned long f_filler[16]; /* reserviert für zukünftige Versionen */
} statvfs_t;
```

`sys/statvfs.h` definiert die Struktur `statvfs64`:

```
typedef struct statvfs64 {
unsigned long f_bsize;    /* Blockgröße des Dateisystems */
unsigned long f_frsize;   /* Fragmentgröße */
fsblkcnt64_t  f_blocks;   /* # Blöcke auf Dateisystem mit Größe f_frsize */
fsblkcnt64_t  f_bfree;    /* # freie Blöcke mit Größe f_frsize */
fsblkcnt64_t  f_bavail;   /* # verfügbare freie Blöcke für Nicht-Superuser */
fsfilcnt64_t  f_files;    /* # Dateiknoten (inodes) */
fsfilcnt64_t  f_ffree;    /* # freie Dateiknoten (inodes) */
fsfilcnt64_t  f_favail;   /* # verfügbare freie Inodes für Nicht-Superuser */
unsigned long f_fsid;     /* Dateisystem-Id (Nummer) */
char f_basetype[FSTYPSZ]; /* Name des Ziel-Dateisystems, Null-terminiert */
unsigned long f_flag;     /* Bitmaske der Optionen von f_flag */
unsigned long f_namemax;  /* maximale Länge der Dateinamen */
char f_fstr[32];          /* dateisystemspezifische Zeichenkette */
unsigned long f_filler[16]; /* reserviert für zukünftige Versionen */
} statvfs_t;
```

Folgende Optionen sind für die Komponente `f_flag` definiert:

`ST_RDONLY`	nur lesbares Dateisystem
`ST_NOSUID`	`setgid`/`setuid`-Semantik wird nicht unterstützt

Folgende Funktionen sind in `sys/statvfs.h` definiert:

```
int statvfs(const char *path, struct statvfs64 *buf);
int statvfs64(const char *path, struct statvfs *buf);
int fstatvfs(int fildes, struct statvfs *buf);
int fstatvfs64(int fildes, struct statvfs64 *buf);
```

Siehe auch `fstatvfs()`, `statvfs()`.

sys/time.h - Zeittypen

Syntax #include <sys/time.h>

Beschreibung

sys/time.h definiert die Struktur timeval:

```
struct timeval {
long tv_sec;       /* Sekunden seit 1. Januar 1970 */
long tv_usec;      /* Mikrosekunden */
};
```

sys/time.h definiert auch die Struktur itimerval:

```
struct itimerval {
struct timeval it_interval;     /* Timer-Intervall */
struct timeval it_value;        /* aktueller Wert */
};
```

sys/time.h definiert weiter die Struktur fd_set, die eine Bitmaske mit Beschreibungen zum Öffnen von Dateien definiert:

```
typedef struct fd_set {
        fd_mask fds_bits[howmany(FD_SETSIZE,NFDBITS)];
} fd_set;
```

sys/time.h definiert folgende Werte für das *which* Argument von getitimer() und setitimer():

ITIMER_REAL	dekrementiert in Echtzeit.
ITIMER_VIRTUAL	dekrementiert in der virtuellen Prozessorzeit.
ITIMER_PROF	dekrementiert beide in virtueller Prozesszeit, wenn das System auf Grund des Prozesses läuft.

Folgende Makros werden definiert:

`void FD_CLR(int fd, fd_set *fdset)`
entfernt *fd* aus der Dateideskriptormenge *fdset*.

`int  FD_ISSET(int fd, fd_set *fdset)`
gibt einen Wert ungleich null zurück, wenn *fd* ein Element aus der Dateideskriptormenge *fdset* ist, ansonsten den Wert Null.

`void FD_SET(int fd, fd_set *fdset)`
fügt den Dateideskriptor *fd* in die Deskriptormenge *fdset* ein.

`void FD_ZERO(fd_set *fdset)`
initialisiert eine Deskriptormenge *fdset* mit der Nullmenge.

`FD_SETSIZE` Maximale Anzahl der Dateideskriptoren in einer `fd_set` Struktur.

Folgende Funktionen sind definiert:

```
int   getitimer(int which, struct itimerval *value);
int   setitimer(int which, const struct itimerval *value,
      struct itimerval *ovalue);
int   gettimeofday(struct timeval *tp, void *tzp);
int   select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *errorfds,
      struct timeval *timeout);
int   utimes(const char *path, const struct timeval times[2]);
```

Siehe auch `getitimer()`, `gettime()`, `select()`, `setitimer()`, `utimes()`.

sys/timeb.h - Zusätzliche Definitionen für Datum und Uhrzeit

Syntax #include <sys/timeb.h>

Beschreibung

`sys/timeb.h` definiert die Struktur `timeb`:

```
struct timeb {
time_t time;              /* aktuelle Zeit in Sek. seit 00:00:00 1.1.1970 */
unsigned short millitm;   /* time in Millisekunden */
short timezone;           /* lokale Zeitzone, in Minuten westlich        */
                          /*  von Greenwich */
short dstflag;            /* Sommerzeitvariable. >0 für Sommerzeit, 0 für */
                          /* Winterzeit, <0 wenn Information fehlt         */
};
```

Der Datentyp `time_t` ist in `sys/types.h` beschrieben.

Die folgende Funktion ist in `sys/timeb.h` definiert:

```
int ftime(struct timeb *tp);
```

Siehe auch `ftime()`, `time.h`.

sys/times.h - Struktur für Dateizeiten

Syntax #include <sys/times.h>

Beschreibung

Die Include-Datei `sys/times.h` definiert die Struktur `struct tms`, die von `times()` zurückgeliefert wird. Diese schließt die folgenden Komponenten ein:

`clock_t`	`tms_utime`	Benutzerzeit
`clock_t`	`tms_stime`	Systemzeit
`clock_t`	`tms_cutime`	Benutzerzeit beendeter Sohnprozesse
`clock_t`	`tms_cstime`	Systemzeit beendeter Sohnprozesse

Der Datentyp `clock_t` wird wie in `sys/types.h` beschrieben definiert.

Folgender Name ist als Funktion deklariert:
`clock_t times(struct tms *`*buffer*`);`

Siehe auch `times()`, `sys/types.h`.

sys/types.h - Datentypen

Syntax #include <sys/types.h>

Beschreibung

sys/types.h definiert Datentypen und schließt zumindest die folgenden Datentypen ein:

clock_t	für die Systemzeit in der Maßeinheit CLK_TCK oder CLOCKS_PER_SEC (siehe time.h)
blkcnt_t	für Blocknummern in Dateien
blkcnt64_t	für Blocknummern in Dateien
dev_t	für Gerätenummern
fsblkcnt_t	für Blocknummern in Dateiesystemen
fsblkcnt64_t	für Blocknummern in Dateiesystemen
fsfilcnt_t	für Dateinummern in Dateisystemen
fsfilcnt64_t	für Dateinummern in Dateisystemen
gid_t	für Gruppennummern
ino_t	für Dateinummern
ino64_t	für Dateinummern
key_t	für die Interprozesskommunikation
mode_t	für einige Dateiattribute
nlink_t	für den Verweiszähler
off_t	für Dateigrößen. Der Datentyp ist ganzzahlig mit Vorzeichen.
off64_t	für Dateigrößen. Der Datentyp ist ganzzahlig mit Vorzeichen.
pid_t	für Prozessnummern und Prozessgruppennummern
size_t	für die Größe von Objekten. Der Datentyp ist ganzzahlig und vorzeichenlos.
ssize_t	für die Byteanzahl oder Fehlernummern. Der Datentyp ist ganzzahlig mit Vorzeichen und kann Werte zwischen -1 und SSIZE_MAX (siehe limits.h) einschließlich enthalten.
time_t	für die Zeit in Sekunden
uid_t	für Benutzernummern
off_t	für Dateigrößen. Der Datentyp ist ganzzahlig mit Vorzeichen.

Die Datentypen `blkcnt64_t` und `off64_t` sind als erweiterte Datentypen (mit Vorzeichen) definiert.
Die Datentypen `fsblkcnt64_t`, `fsfilcnt64_t` und `ino64_t` sind als erweiterte vorzeichenlose Datentypen definiert.

Die restlichen Datentypen sind - außer `key_t` - als arithmetische Datentypen einer angemessenen Größe definiert.

Siehe auch `bsearch()`, `chmod()`, `chown()`, `closedir()`, `creat()`, `fcntl()`, `fstat()`, `getegid()`, `geteuid()`, `getgid()`, `getgroups()`, `getpgrp()`, `getpid()`, `getppid()`, `getuid()`, `kill()`, `lseek()`, `mkdir()`, `mkfifo()`, `msgctl()`, `msgget()`, `msgrcv()`, `msgsnd()`, `open()`, `opendir()`, `readdir()`, `rewinddir()`, `semctl()`, `semget()`, `semop()`, `setgid()`, `setpgid()`, `setsid()`, `setuid()`, `shmat()`, `shmctl()`, `shmdt()`, `shmget()`, `stat()`, `tcgetpgrp()`, `tcsetpgrp()`, `umask()`, `utime()`, `limits.h`, `time.h`.

sys/uio.h - Definitionen für Vektor-Ein-/Ausgabe-Operationen

Syntax #include <sys/uio.h>

Beschreibung

`sys/uio.h` definiert die Struktur `iovec`:

```
typedef struct iovec {
        caddr_t iov_base;  /* Basisadresse des E/A-Speicherbereichs */
        int iov_len;       /* Größe des iov_base-Speicherbereichs */
} iovec_t;
```

Die nachfolgenden Funktionen sind definiert:

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

Siehe auch `read()`, `write()`.

sys/utsname.h - Struktur für Systemnamen

Syntax #include <sys/utsname.h>

Beschreibung

`sys/utsname.h` definiert die Struktur `utsname`, die zumindest folgende Komponenten enthält:

`char sysname[]`	Name des auf dem lokalen Rechner implementierten Betriebssystems
`char nodename[]`	Name des Knotens für den lokalen Rechner in einem implementierungs-spezifischen Kommunikationsnetz
`char release[]`	Freigabe-Nummer der System-Implementierung
`char version[]`	Datum der Freigabe
`char machine[]`	Name des Hardwaretyps, auf dem das System läuft

Die Größe der Zeichen-Vektoren ist nicht festgelegt, aber die dort gespeicherten Daten werden durch das Nullbyte abgeschlossen.

Folgender Name ist als Funktion deklariert:

```
int uname(struct utsname *name);
```

Siehe auch `uname()`.

sys/wait.h - Deklarationen für das Warten von Prozessen

Syntax #include <sys/wait.h>

Beschreibung

Die Include-Datei `sys/wait.h` definiert die folgenden symbolischen Konstanten für die Verwendung in `waitpid()`:

`WNOHANG` nicht warten, wenn kein Status verfügbar, sofort zurückkehren

`WUNTRACED` Status eines angehaltenen Sohnprozesses liefern

und die folgenden Makros für die Analyse der Prozess-Statuswerte:

`WEXITSTATUS()`
liefert Endestatus

`WIFEXITED()`
wahr, wenn Sohnprozess normal beendet wurde

`WIFSIGNALED()`
wahr, wenn Sohnprozess durch nicht abgefangenes Signal beendet wurde

`WIFSTOPPED()`
wahr, wenn Sohnprozess derzeit angehalten ist

`WSTOPSIG()` liefert die Signalnummer, die den Prozess angehalten hat

`WTERMSIG()` liefert die Signalnummer, die den Prozess beendet hat

Erweiterung
`WIFCONTINUED()`
liefert einen Wert ungleich null zurück, wenn der Status eines wieder aufgenommenen Sohnprozesses zurückgeliefert wurde.

`WCOREDUMP()` Ist der Wert von `WIFSIGNALED()` ungleich null, liefert das Makro einen Wert ungleich null, wenn ein Speicherabzug für den beendeten Sohnprozess erzeugt wurde. ❑

Die folgenden Namen sind als Funktionen deklariert:

```
pid_t wait(int *stat_loc);

pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

Siehe auch `exit()`, `wait()`, `waitpid()`, `sys/types.h`.

tar.h - Erweiterte tar-Definitionen

Syntax #include <tar.h>

Beschreibung

Definitionen für den Vorspannblock sind:

Allgemeine Definitionen:

Name	Wert	Beschreibung
TMAGIC	"ustar"	ustar plus Nullbyte
TMAGLEN	6	Länge von TMAGIC (ustar)
TVERSION	"00"	00 ohne ein Nullbyte
TVERSLEN	2	Länge von TVERSION (00)

Typeflag-Feld-Definitionen:

Name	Wert	Beschreibung
REGTYPE	'0'	reguläre Datei
AREGTYPE	'\0'	reguläre Datei
LNKTYPE	'1'	Link
SYMTYPE	'2'	reserviert
CHRTYPE	'3'	zeichenorientierte Gerätedatei
BLKTYPE	'4'	blockorientierte Gerätedatei
DIRTYPE	'5'	Dateiverzeichnis
FIFOTYPE	'6'	FIFO-Datei
CONTTYPE	'7'	reserviert

Mode-Feld-Bit-Definitionen (oktal):

Name	Wert	Beschreibung
TSUID	04000	UID während der Ausführung setzen
TSGID	02000	GID während der Ausführung setzen
TSVTX	01000	reserviert
TUREAD	00400	Leserecht für den Benutzer
TUWRITE	00200	Schreibrecht für den Benutzer
TUEXEC	00100	Ausführ-/Suchrecht für den Benutzer
TGREAD	00040	Leserecht für die Gruppe

Name	Wert	Beschreibung
TGWRITE	00020	Schreibrecht für die Gruppe
TGEXEC	00010	Ausführ-/Suchrecht für die Gruppe
TOREAD	00004	Leserecht für andere
TOWRITE	00002	Schreibrecht für andere
TOEXEC	00001	Ausführ-/Suchrecht für andere

termios.h - Werte für termios definieren

Syntax #include <termios.h>

Beschreibung

Die Include-Datei `termios.h` enthält die Definitionen, die von der `termios`-Schnittstelle verwendet werden.

Es existieren verschiedene `unsigned`-Typdefinitionen für:

`cc_t` `speed_t` `tcflag_t`

Die Struktur `termios` enthält die folgenden Komponenten:

`tcflag_t`	`c_iflag`	Eingabeverarbeitung
`tcflag_t`	`c_oflag`	Ausgabeverarbeitung
`tcflag_t`	`c_cflag`	Hardware-Eigenschaften
`tcflag_t`	`c_lflag`	lokale Verarbeitung
`cc_t`	`c_cc[NCCS]`	Steuerzeichen

Es wird eine Definition gegeben für:

`NCCS` Größe des Vektors `c_cc` für Kontroll-Zeichen

Die besonderen Steuerzeichen sind durch den Vektor `c_cc` definiert:

Ersatzzeichen:

Standard-Eingabeverarbeitung	**rohe**	**Beschreibung**
`VEOF`		`EOF`-Zeichen
`VEOL`		`EOL`-Zeichen
`VERASE`		`ERASE`-Zeichen
`VINTR`	`INTR`	`INTR`-Zeichen
`VKILL`		`KILL`-Zeichen
	`VMIN`	`MIN`-Wert
`VQUIT`	`VQUIT`	`QUIT`-Zeichen
`VSTART`	`VSTART`	`START`-Zeichen
`VSTOP`	`VSTOP`	`STOP`-Zeichen
`VSUSP`	`VSUSP`	`SUSP`-Zeichen
	`VTIME`	`TIME`-Zeichen

Die Ersatzwerte sind eindeutig, außer dass VMIN und VTIME dieselben Werte wie VEOF und VEOL haben können.

Eingabeverarbeitung

Die Komponente c_iflag beschreibt die grundlegende Eingabesteuerung für Terminals:

BRKINT	Signal-Unterbrechung bei "break"
ICRNL	Bei der Eingabe CR auf NL abbilden
IGNBRK	"break"-Bedingung ignorieren
IGNCR	CR ignorieren
IGNPAR	Zeichen mit Paritätsfehlern ignorieren
INLCR	Bei der Eingabe NL auf CR abbilden
INPCK	Eingabe-Paritätsprüfung einschalten
ISTRIPa	Löschzeichen
IUCLC	Bei der Eingabe Groß- auf Kleinbuchstaben abbilden
IXANY	Irgendein Zeichen als Startzeichen für Ausgabewiederholung freigeben
IXOFF	Start/Stop-Eingabesteuerung ermöglichen
IXON	Start/Stop-Ausgabesteuerung ermöglichen
PARMRK	Paritätsfehler markieren

Ausgabeverarbeitung

Die Komponente c_oflag bestimmt, wie das System Ausgaben behandelt:

OPOST	Ausgabe nachbearbeiten
OLCUC	Bei Ausgabe Klein- in Großbuchstaben umwandeln
ONLCR	Bei Ausgabe NL in CR-NL umwandeln
OCRNL	Bei Ausgabe CR in NL umwandeln
ONOCR	Keine Ausgabe von CR bei Spalte 0
ONLRET	NL führt CR-Funktion aus
OFILL	Füllzeichen für Wartezeiten benutzen
OFDEL	Füllzeichen ist DEL, sonst NUL

NLDLY	Wartezeiten für NL wählen:
NL0	Zeilenendezeichen Typ 0
NL1	Zeilenendezeichen Typ 1
CRDLY	Wartezeiten für CR wählen:
CR0	CR-Wartezeit Typ 0
CR1	CR-Wartezeit Typ 1
CR2	CR-Wartezeit Typ 2
CR3	CR-Wartezeit Typ 3
TABDLY	Wartezeiten für Horizontal-Tabulatoren auswählen:
TAB0	Horizontal-Tabulator Wartezeit Typ 0
TAB1	Horizontal-Tabulator Wartezeit Typ 1
TAB2	Horizontal-Tabulator Wartezeit Typ 2
TAB3	Tabulatoren in Leerzeichen umwandeln
BSDLY	Wartezeiten für Rückschritt auswählen:
BS0	Rückschritt-Wartezeit Typ 0
BS1	Rückschritt-Wartezeit Typ 1
VTDLY	Wartezeiten für Vertikal-Tabulatoren auswählen:
VT0	Vertikal-Tabulator Wartezeit Typ 0
VT1	Vertikal-Tabulator Wartezeit Typ 1
FFDLY	Wartezeit für Seitenvorschub auswählen:
FF0	Seitenvorschub Wartezeit Typ 0
FF1	Seitenvorschub Wartezeit Typ 1

Auswahl der Baudrate

Die Eingabe- und Ausgabe-Baudraten sind in der Struktur `termios` abgespeichert. Diese sind gültige Werte für Objekte des Typs `speed_t`. Die folgenden Werte sind definiert, aber nicht alle Baudraten müssen von der zu Grunde liegenden Hardware unterstützt werden:

B0	Verbindungsabbruch (Hang Up)
B50	50 Baud
B75	75 Baud

B110	110 Baud
B134	134.5 Baud
B150	150 Baud
B200	200 Baud
B300	300 Baud
B600	600 Baud
B1200	1200 Baud
B1800	1800 Baud
B2400	2400 Baud
B4800	4800 Baud
B9600	9600 Baud
B19200	19200 Baud
B38400	38400 Baud

Hardware-Eigenschaften

Die Komponente c_cflag beschreibt die Hardware-Steuerung der Datensichtstation; nicht alle der beschriebenen Werte müssen von der zu Grunde liegenden Hardware unterstützt werden:

CSIZE	Zeichengröße
CS5	5 Bit
CS6	6 Bit
CS7	7 Bit
CS8	8 Bit
CSTOPB	Zwei Stopbits senden, sonst eins
CREAD	Empfänger einschalten
PARENB	Paritätsprüfung einschalten
PARODD	Ungerade Parität, sonst gerade
HUPCL	Verbindungsabbruch bei letztem Schließen
CLOCAL	lokale Verbindung, sonst Wählverbindung

Lokale Verarbeitung

Die Komponente `c_lflag` der Argument-Struktur wird verwendet, um verschiedene Funktionen des Terminals zu steuern:

`ECHO`	Echo einschalten
`ECHOE`	Rückschritt durch Löschzeichen darstellen
`ECHOK`	KILL ausgeben
`ECHONL`	NL ausgeben
`ICANON`	Standard-Eingabeverarbeitung
`IEXTEN`	Besondere Eingabezeichen-Verarbeitung
`ISIG`	Signalverarbeitung einschalten
`NOFLSH`	Verwerfen nach Unterbrechung oder Ende ausschalten
`TOSTOP`	Signal SIGTTOU für Hintergrund-Ausgabe senden
`XCASE`	Kanonische Groß-/Kleinbuchstaben-Darstellung

Auswahl der Eigenschaften

Die folgenden symbolischen Konstanten für die Verwendung in `tcsetattr()` sind definiert:

`TCSANOW`	Eigenschaften sofort ändern
`TCSADRAIN`	Eigenschaften erst nach Abwarten der Ausgabe ändern
`TCSAFLUSH`	Eigenschaften erst nach Abwarten der Ausgabe ändern; zusätzlich anstehende Eingaben verwerfen

Leitungssteuerung

Die folgenden symbolischen Konstanten für die Verwendung in `tcflush()` sind definiert:

`TCIFLUSH`	anstehende Eingaben verwerfen
`TCOFLUSH`	nicht übertragene Ausgaben verwerfen
`TCIOFLUSH`	anstehende Eingaben und nicht übertragene Ausgaben verwerfen

Die folgenden symbolischen Konstanten für die Verwendung in `tcflow()` sind definiert:

`TCIOFF`	ein Stop-Zeichen übertragen, um die Eingabe zu unterbinden
`TCION`	ein Start-Zeichen übertragen, um die Eingabe erneut zu starten
`TCOOFF`	Ausgabe unterbinden
`TCOON`	Ausgabe erneut starten

Die folgenden Namen sind als Funktionen deklariert:

`cfgetispeed()`	`tcdrain()`	`tcgetattr()`
`cfgetospeed()`	`tcflow()`	`tcsendbreak()`
`cfsetispeed()`	`tcflush()`	`tcsetattr()`
`cfsetospeed()`		

Siehe auch `cfgetispeed()`, `cfgetospeed()`, `cfsetispeed()`, `cfsetospeed()`, `tcdrain()`, `tcflow()`, `tcflush()`, `tcgetattr()`, `tcsendbreak()`, `tcsetattr()`.

time.h - Datentypen für die Zeit

Syntax #include <time.h>

Beschreibung

time.h deklariert die Struktur tm, die zumindest folgende Komponenten enthält:

int tm_sec	Sekunden [0, 61]. Der Bereich [0,61] lässt die Schaltsekunde zu.
int tm_min	Minuten [0, 59]
int tm_hour	Stunde [0, 23]
int tm_mday	Tag des Monats [1, 31]
int tm_mon	Monat des Jahres [0, 11]
int tm_year	Jahre seit 1900
int tm_wday	Wochentag [0, 6] (Sonntag = 0)
int tm_yday	Tag im Jahr [0, 365]
int tm_isdst	Sommerzeitvariable; der Wert ist positiv für Sommerzeit, 0 für Winterzeit und negativ, wenn die Information fehlt.

Folgende symbolische Namen sind definiert:

NULL	Nullzeiger
CLK_TCK	Zeittakt pro Sekunde, die von time() zurückgegen wird. Der Wert ist derzeit identisch mit dem Returnwert von sysconf(_SC_CLK_TCK); neue Anwendungen sollten jedoch sysconf() aufrufen (Wird zukünftig vom X/Open-Standard nicht mehr unterstützt.).
CLOCKS_PER_SEC	Zahl, die benutzt wird, um den Returnwert von clock() in Sekunden umzuwandeln. Obwohl der Wert auf X/Open-konformen Systemen 1 000 000 betragen soll, kann er auf anderen Systemen variabel sein. Auf jeden Fall sollte nicht davon ausgegangen werden, dass CLOCKS_PER_SEC eine Übersetzungszeit-Konstante ist.

Die Datentypen clock_t, size_t und time_t sind definiert (siehe sys/types.h).

Folgende Namen sind als Funktionen deklariert:

```
char *asctime(const struct tm *timeptr);
clock_t clock(void);
char *ctime(const time_t *clock);
double difftime(time_t *time1, time_t *time0)
```

```
struct tm *gmtime(const time_t *timer);
struct tm *localtime(const time_t *timer);
time_t mktime(struct tm *timeptr);
size_t strftime(char *s, size_t maxsize, const char *format);
char *strptime(const char *buf, const char *format);
time_t time(time_t *tloc);
void tzset(void);
```

Folgende externe Variablen sind deklariert:

```
extern in daylight;
extern long int timezone;
extern char *tzname[]
```

Siehe auch `asctime()`, `clock()`, `ctime()`, `daylight`, `difftime()`, `gmtime()`, `localtime()`, `mktime()`, `strftime()`, `strptime()`, `sysconf()`, `time()`, `timezone`, `tzname()`, `tzset()`, `utime()`.

ucontext.h - Benutzerkontext

Syntax #include <ucontext.h>

Beschreibung

Es wird der Typ `mcontext_t` definiert:

```
typedef struct {
        gregset_t    _gpegs;
        fpregset_t   _fpregs;
        /* BS2000 */
        int          _uc_cc;
        int          _uc_pc;
        char         _uc_fpvalid;
        char         _uc_type;
        short        _uc_level;
        int          _uc_priv1;
        int          _uc_priv2;
        int          _uc_priv3[256];
        /* BS2000 */
} mcontext;
```

Ferner die Struktur `ucontext`, die den Kontrollkontext innerhalb eines ablaufenden Prozesses definiert:

```
typedef struct ucontext {
        ulong_t   uc_flags;
        struct ucontext  *uc_link;
        sigset_t    uc_sigmask;
        stack_t     uc_stack;
        mcontext_t  uc_mcontext;
} ucontext_t;
```

`uc_link` ist ein Zeiger auf den Kontext, der wieder aufgenommen werden soll, wenn der aktuelle Kontext beendet ist. Wenn `uc_link` gleich Null ist, dann ist der aktuelle Kontext der Hauptkontext; der Prozess wird beendet, wenn dieser Kontext beendet wird.

`uc_sigmask` definiert die Signalmengen, welche blockiert werden, wenn dieser Kontext aktiv ist (siehe `sigprocmask`(2)).

`uc_stack` definiert den Stapel für diesen Kontext (siehe `sigaltstack`(2)).

`uc_mcontext` enthält die gesicherten Maschinenregister und implementierungsspezifische Kontextdaten. Portable Anwendungen sollten `uc_mcontext` nicht modifizieren.

Die folgenden Funktionen sind definiert:

```
int getcontext(ucontext_t *ucp);
int setcontext(const ucontext_t *ucp);
void makecontext(ucontext_t *ucp, (void *func)(), int argc, ...);
int swapcontext(ucontext_t *oucp, const ucontext_t *ucp);
```

Siehe auch `getcontext`(2), `sigaction`(2), `sigprocmask`(2), `sigaltstack`(2), `makecontext`(3C).

ulimit.h - Kommandos für ulimit

Syntax #include <ulimit.h>

Beschreibung

Die Include-Datei `ulimit.h` definiert die symbolischen Konstanten, die für die Funktion `ulimit()` verwendet werden können.

Symbolische Konstanten:

`UL_GETFSIZE`	maximale Dateigröße ermitteln
`UL_SETFSIZE`	maximale Dateigröße setzen

Der folgende Name wird entweder als Funktion oder Makro definiert:

```
long int ulimit (int cmd, ...);
```

Siehe auch `ulimit()`.

unistd.h - Symbolische Standardkonstanten und -strukturen

Syntax #include <unistd.h>

Beschreibung

unistd.h definiert verschiedene symbolische Konstanten und Datentypen und deklariert verschiedene Funktionen.

Makros für die Versionsabfrage

Folgende symbolische Konstanten sind mit festen Werten definiert:

_POSIX_VERSION ganze Zahl, die die Version des Standards ISO POSIX-1 angibt (C-Sprachbindung). Der Wert verändert sich mit jeder neuen Version des Standards.

_POSIX2_VERSION ganze Zahl, die die Version des Standards ISO POSIX-2 DIS angibt (Kommandos). Der Wert verändert sich mit jeder neuen Version des Standards.

_POSIX2_C_VERSION ganze Zahl, die die Version des Standards ISO POSIX-2 DIS (C-Sprachbindung) angibt und ob die X/Open POSIX2 C-Sprachbindungsfunktionen unterstützt werden. Der Wert verändert sich mit jeder neuen Version des Standards. Wenn die Option für die C-Sprachbindung des ISO POSIX-2 Standard und damit die X/Open POSIX2 C-Sprachbindungsfunktionen nicht unterstützt werden, wird der Wert auf -1 gesetzt.

_XOPEN_VERSION ganze Zahl, die die Version des XPG angibt, zu der das System kompatibel ist. Sie hat in dieser Implementierung den Wert 4.

Obligatorische symbolische Konstanten

Folgende symbolische Konstanten sind entweder undefiniert oder mit einem Wert ≠ -1 definiert. Wenn eine Konstante undefiniert ist, sollte die Anwendung sysconf(), pathconf() oder fpathconf() benutzen, um festzustellen, welche Gegebenheiten auf dem System aktuell sind.

_POSIX_CHOWN_RESTRICTED
Die Verwendung von chown() ist auf Prozesse mit besonderen Rechten beschränkt. Der Wert ist ≠ -1.

_POSIX_NO_TRUNC Pfadnamen-Komponenten, die länger als {NAME_MAX} sind, erzeugen einen Fehler. Der Wert ist ≠ -1.

_POSIX_VDISABLE Sonderzeichen für die Terminalausgabe (siehe termios.h) können mit diesem Zeichen abgeschaltet werden. Der Wert ist ≠ -1.

_POSIX_SAVED_IDS Sichert exec, die effektive Benutzer- und Gruppennummer.

_POSIX_JOB_CONTROL
Die Implementierung unterstützt eine Auftragssteuerung.

Konstanten für Optionen und Merkmalsgruppen

Folgende symbolische Konstanten sind mit -1 definiert, wenn die Implementierung das entsprechende Merkmal nicht unterstützt. Sie sind mit einem Wert ≠ -1 definiert, wenn die Implementierung dieses Merkmal unterstützt. Wenn die Konstanten nicht definiert sind, kann die sysconf-Funktion verwendet werden, um festzustellen, ob das Merkmal für einen speziellen Anwendungsfall unterstützt wird.

_POSIX2_C_DEV Die Implementierung unterstützt die C-Entwicklungswerkzeug-Option.

_POSIX2_CHAR_TERM Die Implementierung unterstützt mindestens einen Terminaltyp.

_POSIX2_FORT_DEV Die Implementierung unterstützt die Entwicklungswerkzeugs-Option für FORTRAN.

_POSIX2_FORT_RUN Die Implementierung unterstützt die Laufzeitwerkzeugs-Option für FORTRAN.

_POSIX2_LOCALEDEF Die Implementierung unterstützt die Erzeugung von Lokalitäten mit dem localedef-Kommando.

_POSIX2_SW_DEV Die Implementierung unterstützt die Option für die Software-Entwicklungs-Kommandos.

_POSIX2_UP Die Implementierung unterstützt die Option für die Benutzer-Portabilitäts-Kommandos.

_XOPEN_CRYPT Die Implementierung unterstützt die X/Open-Merkmalgruppe für Verschlüsselung.

_XOPEN_SHM Die Implementierung unterstützt die X/Open Merkmalsgruppe für gemeinsam nutzbaren Speicherplatz.

Die folgenden Makros informiere über die eingesetzte Version bzw. Implementierung:

_LFS_LARGEFILE

_LFS64_LARGEFILE

_LFS_ASYNCHRONOUS-IO

_LFS64_ASYNCHRONOUS-IO

_LFS_STDIO

_LFS64_STDIO

Konstanten für Funktionen

Folgende symbolische Konstante ist unabhängig von einer Funktion definiert:

`NULL`	Nullzeiger

Folgende symbolischen Konstanten werden für die Funktion `access()` definiert:

`R_OK`	Test auf Leserecht
`W_OK`	Test auf Schreibrecht
`X_OK`	Test auf Ausführungs-/Durchsuchrecht
`F_OK`	Test auf Existenz der Datei

Die Konstanten `F_OK`, `R_OK`, `W_OK` und `X_OK` und die Ausdrücke `R_OK|W_OK`, `R_OK|X_OK` und `R_OK|W_OK|X_OK` besitzen verschiedene Werte.

Folgende symbolische Konstanten sind für die Funktion `confstr()` definiert:

`_CS_PATH`	Wert für die Umgebungsvariable PATH, durch die alle Standard-Kommandos gefunden werden.

`_CS_LFS_CFLAGS`
`_CS_LFS_LDFLAGS`
`_LFS_LIBS`
`_CS_LFS_LINTFLAGS`

`_CS_LFS64_CFLAGS`
`_CS_LFS64_LDFLAGS`
`_LFS_LIBS64`
`_CS_LFS64_LINTFLAGS`

Folgende symbolische Konstanten werden für die Funktionen `lseek()` und `fcntl()` definiert (sie haben eindeutige Werte):

`SEEK_SET`	Dateiposition gleich *offset* setzen.
`SEEK_CUR`	Dateiposition gleich aktuelle Position plus *offset* setzen.
`SEEK_END`	Dateiposition gleich `EOF` plus *offset* setzen.

Folgende symbolische Konstanten werden für `sysconf()` definiert:

_SC_2_C_BIND
_SC_2_C_DEV
_SC_2_C_VERSION
_SC_2_FORT_DEV
_SC_2_FORT_RUN
_SC_2_LOCALEDEV
_SC_2_SW_DEV
_SC_2_UPE
_SC_2_VERSION
_SC_ARG_MAX
_SC_BC_BASE_MAX
_SC_BC_DIM_MAX
_SC_BC_SCALE_MAX
_SC_BC_STRING_MAX
_SC_CHILD_MAX
_SC_CLK_TCK
_SC_COLL_WEIGHTS_MAX
_SC_EXPR_NEST_MAX
_SC_JOB_CONTROL
_SC_LINE_MAX
_SC_NGROUPS_MAX
_SC_OPEN_MAX
_SC_PASS_MAX (Wird zukünftig vom X/Open-Standard nicht mehr unterstützt.)
_SC_RE_DUP_MAX
_SC_SAVED_IDS
_SC_STREAM_MAX
_SC_TZNAME_MAX
_SC_VERSION
_SC_XOPEN_CRYPT
_SC_XOPEN_ENH_I18N
_SC_XOPEN_SHM
_SC_XOPEN_VERSION

Folgende symbolische Konstanten werden für `pathconf()` definiert:

```
_PC_CHOWN_RESTRICTED
_PC_LINK_MAX
_PC_MAX_CANON
_PC_MAX_INPUT
_PC_NAME_MAX
_PC_NO_TRUNC
_PC_PATH_MAX
_PC_PIPE_BUF
_PC_VDISABLE
```

Folgende symbolische Konstanten werden für Datenströme definiert:

`STDIN_FILENO`	Dateinummer von stdin. Der Wert ist 0.
`STDOUT_FILENO`	Dateinummer von stout. Der Wert ist 1.
`STDERR_FILENO`	Dateinummer von stderr. Der Wert ist 2.

Datentypdefinitionen

Die Datentypen `size_t`, `ssize_t`, `uid_t`, `gid_t`, `off_t`, und `pid_t` sind definiert (siehe `sys/types.h`).

Deklarationen

Folgende Namen sind als Funktionen deklariert:

```
int access(const char *path, int amode);
unsigned int alarm(unsigned int seconds);
int chdir(const char *path);
int chown(const char *path, uid_t owner, gid_t group);
int chroot(const char *path);
```
(Wird zukünftig vom X/Open-Standard nicht mehr unterstützt.)
```
int close(int fildes);
size_t confstr(int name, char *buf, size_t len);
char *crypt(const char *key, const char *salt);
char *ctermid(char *s);
char *cuserid(char *s);
```
(Wird zukünftig vom X/Open-Standard nicht mehr unterstützt.)
```
int dup(int fildes);
int dup2(int fildes, int fildes2);
void encrypt(char block[64], int edflag);
int execl(const char *path, const char *arg0, ...);
int execle(const char *file, const char *arg0, ...);
```

```
int execlp(const char *file, const char *arg0, ...);
int execv(const char *path, char * const argv[]);
int execve(const char *path, char * const argv[], char *const envp[]);
int execvp(const char *file, char * const argv[]);
void _exit(int status);
pid_t fork(void);
long int fpathconf(int fildes, int name);
int fsync(int fildes);
int ftruncate64 (int fildes, off64_t length);
char *getcwd(char *buf, size_t size);
gid_t getegid(void);
uid_t geteuid(void);
gid_t getgid(void);
int getgroups(int gidsetsize, gid_t grouplist[]);
char *getlogin(void);
int getlogin_r(char *name, size_t namesize);
int getopt(int argc, char * const argv[], const char *optstring);
char *getpass(const char *prompt); (Wird zukünftig vom X/Open-Standard nicht
mehr unterstützt.)
pid_t getpgrp(void);
pid_t getpid(void);
pid_t getppid(void);
uid_t getuid(void);
int isatty(int fildes);
int link(const char *path1, const char *path2);
int lockf64(int fildes, int function, off64_t size);
off_t lseek(int fildes, off_t offset, int whence);
off64_t lseek64 (int fildes, off64_t offset, int whence);
int nice(int incr);
long int pathconf(const char *path, int name);
int pause(void);
int pipe(int fildes[2]);
ssize_t read(int fildes, void *buf, size_t nbyte);
int rmdir(const char *path);
int setgid(gid_t gid);
int setpgid(pid_t pid, pid_t pgid);
pid_t setsid(void);
int setuid(uid_t uid);
unsigned int sleep(unsigned int seconds);
void swab(const void *src, void *dest, ssize_t nbytes);
long int sysconf(int name);
pid_t tcgetpgrp(int fildes);
int tcsetpgrp(int fildes, pid_t pgrp_id);
int truncate64 (const char *path, off64_t length);
```

```
char *ttyname(int fildes);
int unlink(const char *path);
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

Folgende externe Variablen werden deklariert:

```
extern char *optarg;
extern int optind, opterr, optopt;
```

Siehe auch access(), alarm(), chdir(), chown(), chroot(), close(), crypt(), ctermid(), cuserid(), dup(), encrypt(), environ, exec, exit(), fcntl(), fork(), fpathconf(), fsync(), getcwd(), getegid(), geteuid(), getgid(), getgroups(), getlogin(), getlogin_r(), getpass(), getpgrp(), getpid(), getppid(), getuid(), isatty(), link(), lseek(), nice(), pathconf(), pause(), pipe(), read(), rmdir(), setgid(), setpgid(), setsid(), setuid(), sleep(), swab(), sysconf(), tcgetpgrp(), tcsetpgrp(), ttyname(), unlink(), write(), limits.h, sys/types.h, termios.h.

utime.h - Zeitstrukturen manipulieren

Syntax #include <utime.h>

Beschreibung

Die Include-Datei `utime.h` deklariert die Struktur `utimbuf`, die die folgenden Komponenten besitzt:

`time_t`	`actime`	Zugriffszeit
`time_t`	`modtime`	Modifikationszeit

Die Zeiten werden in Sekunden seit dem Epochenwert gemessen.
Der Typ `time_t` ist in `sys/types.h` deklariert.

Der folgende Name ist als Funktion deklariert:

```
int utime(const char *path, const struct utimbuf *times);
```

Siehe auch `utime()`, `sys/types.h`.

utmpx.h - Einsprungformat

Syntax #include <utmpx.h>

Beschreibung

Es wird die Struktur `utmpx` definiert:

```
struct utmpx {
       char  ut_user[32];           /* Benutzerkennung */
       char  ut_id[4];              /* ID der Inittab */
       char  ut_line[32];           /* Gerätename (Konsole, lnxx) */
       pid_t ut_pid;                /* Prozess-Id */
       short ut_type;               /* Typ des Eintrags */
       struct exit_status ut_exit;  /* Exit-Status */
       struct timeval ut_tv;        /* Zeitstempel des Eintrags */
       long  ut_session;            /* Session-Id bei Fenstertechnik */
       long  pad[5];                /* reserviert für künftige Versionen */
       short ut_syslen;             /* signifik. Länge ut_host (incl. \0) */
       char  ut_host[257];          /* Name des entfernten Host */
};
```

Die Datei `/var/adm/utmpx` enthält Benutzer- und Abrechnungsinformationen für Kommandos wie `who()`, `write()` und `login()`.

Die folgenden symbolischen Konstanten sind als mögliche Werte des Datenmembers `ut_type` der `utmpx`-Struktur definiert:

`EMPTY`	keine gültige Abrechnungsinformation für den Benutzer vorhanden.
`BOOT_TIME`	Zeitpunkt, zu dem das Sytem gebootet wurde.
`OLD_TIME`	Zeitpunkt, zu dem die Systemzeit geändert wurde.
`NEW_TIME`	Identifiziert die Zeit nach der Änderung der Systemzeit.
`USER_PROCESS`	Identifiziert einen Prozess.
`INIT_PROCESS`	Identifiziert einen Prozess, der vom INIT_Prozess erzeugt wurde.
`LOGIN_PROCESS`	Identifiziert den Session-Leader eines angemeldeten Benutzers.
`DEAD_PROCESS`	Identifiziert einen Session-Leader, der sich beendet hat.

Die folgenden Funktionen sind definiert:

```
void endutxent(void);
struct utmpx *getutxent(void);
struct utmpx *getutxid(const struct utmpx *id);
struct utmpx *getutxline(const struct utmpx *line);
struct utmpx *pututxline(const struct utmpx *utmpx);
void setutxent(void);
```

Siehe auch `endutxent()`.

varargs.h - variable Argumentenliste behandeln

Syntax #include <varargs.h>

va_alist

va_dcl

void va_start(pvar)
va_list pvar;

type va_arg(pvar, *type*)
va_list pvar;

void va_end(pvar)
va_list pvar;

Beschreibung

Die Include-Datei `varargs.h` enthält einige Makros, die es erlauben, portable Prozeduren mit variablen Argumentenlisten zu schreiben. Routinen, die variable Argumentenlisten haben (wie z.B. `printf()`) , die aber nicht `varargs.h` benutzen, sind nicht portabel, da verschiedene Rechner verschiedene Konventionen für die Argumentübergabe verwenden.

`va_alist`	wird als Parameterliste in einem Funktionskopf verwendet.
`va_dcl`	ist eine Deklaration für `va_alist`. Auf `va_dcl` sollte kein Semikolon folgen.
`va_list`	ist ein Datentyp, der für die Variable definiert ist, welche die Liste abarbeitet.
`va_start()`	wird aufgerufen, um `pvar` auf den Anfang der Liste zu initialisieren.
`va_arg()`	liefert das nächste Argument aus der Liste, auf die `pvar` zeigt. *type* ist der Typ, der für das Argument angenommen wird. Verschiedene Typen können gemischt werden, aber die Routine muss wissen, welchen Typ das nächste Argument hat, da dies zur Laufzeit nicht festgestellt werden kann.
`va_end()`	wird zum "Aufräumen" verwendet.

Mehrere Abarbeitungen sind möglich; jede wird durch `va_start ... va_end` eingeschlossen.

Hinweis Die aufrufende Routine muss angeben, wie viele Argumente es gibt, da es nicht immer möglich ist, dies aus dem Inhalt des Stacks zu bestimmen. Für `execl()` zum Beispiel wird als Kennzeichen für das Listenende ein Nullzeiger übergeben. Die Funktion `printf()` ermittelt die Zahl der Argumente aus dem ersten Argument (Format).

Es ist nicht portabel, ein zweites Argument des Typs `char`, `short` oder `float` für `va_arg` anzugeben, da es für die gerufene Funktion keine Argumente des Typs `char`, `short` oder `float` gibt. Die Sprache C konvertiert Argumente vom Typ `char` und `short` zu `int` und Argumente des Typs `float` zu `double`, bevor sie an eine Funktion übergeben werden.

Siehe auch `iconvexec()`, `printf()`.

wchar.h - Datentypen für Langzeichen-Werte

Syntax #include <wchar.h>

Beschreibung

Folgende Datentypen sind durch `typedef` definiert:

`wchar_t`	Siehe `stddef.h`.
`wint_t`	Ein ganzzahliger Datentyp, der beliebige gültige Werte von `wchar_t` oder `WEOF` speichern kann.
`wctype_t`	Ein skalarer Datentyp eines Datenobjekts, der Werte aufnehmen kann, die eine für die Lokalität spezifische Zeichenklassifikation darstellen.
`FILE`	Siehe `stdio.h`.
`size_t`	Siehe `stddef.h`.

Folgende Namen sind als Funktionen deklariert:

```
wint_t fgetwc(FILE *stream);
wchar_t *fgetws(wchar_t *s, int n; FILE  *stream);
wint_t fputwc(wint_t c, FILE *stream);
int  fputws(const wchar_t *s, FILE *stream);
wint_t getwc(FILE *stream); (und auch als Makro definiert)
wint_t getwchar(void); (und auch als Makro definiert)
wchar_t *getws(wchar_t *s);
int iswalnum(wint_t wc); (und auch als Makro definiert)
int iswalpha(wint_t wc); (und auch als Makro definiert)
int iswcntrl(wint_t wc); (und auch als Makro definiert)
int iswdigit(wint_t wc); (und auch als Makro definiert)
int iswgraph(wint_t wc); (und auch als Makro definiert)
int iswlower(wint_t wc); (und auch als Makro definiert)
int iswprint(wint_t wc); (und auch als Makro definiert)
int iswpunct(wint_t wc); (und auch als Makro definiert)
int iswspace(wint_t wc); (und auch als Makro definiert)
int iswupper(wint_t wc); (und auch als Makro definiert)
int iswxdigit(wint_t wc); (und auch als Makro definiert)
int iswctype(wint_t wc, wctype_t prop); (und auch als Makro definiert)
wint_t putwc(wint_t c, FILE *stream); (und auch als Makro definiert)
wint_t putwchar(wint_t c); (und auch als Makro definiert)
int putws(const wchar_t *s);
wint_t towlower(wint_t wc); (und auch als Makro definiert)
wint_t towupper(wint_t wc); (und auch als Makro definiert)
wint_t ungetwc(wint_t c, FILE *stream);
```

```
wctype_t wctype(const char *property);
wchar_t *wcscat(wchar_t *ws1, const wchar_t *ws2);
wchar_t *wcschr(const wchar_t *ws, wchar_t wc);
int wcscmp(const wchar_t *ws1, const wchar_t *ws2);
int wcscoll(const wchar_t *ws1, const wchar_t *ws2);
wchar_t *wcscpy(wchar_t *ws1, const wchar_t *ws2);
size_t wcscspn(const wchar_t *ws1, const wchar_t *ws2);
size_t wcsftime(wchar_t *wcs, size_t maxsize, const char *fmt,
const struct tm *timptr);
size_t  wcslen(const wchar_t *ws);
wchar_t *wcsncat(wchar_t *ws1, const wchar_t *ws2, size_t n);
int wcsncmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);
wchar_t *wcsncpy(wchar_t *ws1, const wchar_t *ws2, size_t n);
wchar_t *wcspbrk(const wchar_t *ws1, const wchar_t *ws2);
wchar_t *wcsrchr(const wchar_t *ws, wchar_t *wc)
size_t wcsspn(const wchar_t *ws1, const wchar_t *ws2);
double wcstod(const wchar_t *nptr, wchar_t **endptr);
wchart *wcstok(wchar_t *ws1, const wchar_t *ws2);
long int wcstol(const wchar_t *nptr, wchar_t **endptr, int base);
unsigned long int wcstoul (const wchar_t *nptr, wchar_t **endptr,
                           int base);
wchar_t *wcswcs(const wchar_t *ws1, const wchar_t *ws2);
int wcswidth(const wchar_t *pwcs, size_t n);
size_t wcsxfrm(wchar_t *ws1, const wchar_t *ws2, size_t n);
int wcwidth(const wchar_t wc);
```

Folgende Makronamen sind definiert:

`WEOF`	Konstante vom Typ `wint_t`, der von mehreren Multibyte-Funktionen zurückgegeben wird, um das Dateiende anzuzeigen.
`NULL`	Siehe `stddef.h`.

Hinweis Das Einbinden der Include-Datei `wchar.h` kann alle Symbole aus den Include-Dateien `ctype.h`, `stdio.h`, `stdarg.h`, `stdlib.h`, `string.h`, `stddef.h` und `time.h` sichtbar machen.

Siehe auch `iswalnum()`, `iswalpha`, `iswcntrl()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswprint()`, `iswpunct()`, `iswspace()`, `iswupper()`, `iswxdigit()`, `mblen()`, `mbstowcs()`, `mbtowc()`, `setlocale()`, `towlower()`, `towupper()`, `wcstombs()`, `wctomb()`, `locale.h`, `stddef.h`.

wctype.h – wide character classification and mapping utilities

Syntax #include <wctype.h>

Beschreibung

Folgende Datentypen sind durch `typedef` definiert:

`wint_t`	Siehe `wchar.h`.
`wctrans_t`	Ein skalarer Datentyp eines Datenobjekts, der Werte aufnehmen kann, die eine für die Lokalität spezifische Zeichenklassifikation darstellen.
`wctype_t`	Siehe `wchar.h`.

Folgende Namen sind als Funktionen deklariert:

```
int iswalnum(wint_t);
int iswalpha(wint_t);
int iswcntrl(wint_t);
int iswdigit(wint_t);
int iswgraph(wint_t);
int iswlower(wint_t);
int iswprint(wint_t);
int iswpunct(wint_t);
int iswspace(wint_t);
int iswupper(wint_t);
int iswxdigit(wint_t);
int iswctype(wint_t, wctype_t);
wint_t towctrans(wint_t, wctrans_t);
wint_t towlower(wint_t);
wint_t towupper(wint_t);
wctrans_t wctrans(const char *);
wctype_t wctype(const char *);
```

Folgende Makronamen sind definiert:

`WEOF`	Konstante vom Typ `wint_t`, der von mehreren Multibyte-Funktionen zurückgegeben wird, um das Dateiende anzuzeigen.

Für alle in dieser Headerdatei beschriebenen Funktionen, die ein Argument des Typs `wint_t` akzeptieren, ist der Wert repräsentativ als `wchar_t` oder ist gleich dem Wert von `WEOF`. Bei allen anderen Werten dieses Arguments ist das Verhalten undefiniert.

Das Verhalten dieser Funktionen ist beeinflusst durch die Kategorie `LC_CTYPE` der aktuellen Lokalität.

Hinweis Das Einbinden der Include-Datei `wctype.h` kann alle Symbole aus den Include-Dateien `ctype.h`, `stdio.h`, `stdarg.h`, `stdlib.h`, `string.h`, `stddef.h`, `time.h` und `wchar.h` sichtbar machen.

Siehe auch `iswalnum()`, `iswalpha()`, `iswcntrl()`, `iswctype()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswprint()`, `iswpunct()`, `iswspace()`, `iswupper()`, `iswxdigit()`, `setlocale ()`, `towctrans()`, `towlower()`, `towupper()`, `wcttrans()`, `wctype()`, `locale.h`, `wchar.h`.

Anhang: KR- und ANSI-Funktionalität

Die Ausführungen dieses Abschnitts beziehen sich auf die Funktionen, die in der Tabelle auf Seite 18 (Umfang der unterstützten C-Bibliothek) mit xx gekennzeichnet sind.

Die C-Bibliotheksfunktionen wurden erstmals mit C V1.0 zur Verfügung gestellt. Zu diesem Zeitpunkt gab es keinen ANSI-definierten C-Bibliotheksumfang. Die Implementierung orientierte sich an der „vorläufigen“ Definition durch Kernighan/Ritchie („KR“) bzw. an den marktüblichen UNIX/SINIX-Implementierungen.

Die Anpassung der C-Bibliotheksfunktionen an den ANSI-Standard (C V2.0) führte bei der Ausführung einiger Ein-/Ausgabefunktionen zu Abweichungen gegenüber der Vorgängerversion. Um einerseits den ANSI-Standard voll zu erfüllen, andererseits das gewohnte Ablaufverhalten von „Alt“-Programmen zu gewährleisten, wurden die von den Abweichungen betroffenen Ein-/Ausgabefunktionen bei C/C++ VersionenV2.xx in zwei Varianten angeboten:
Mit der neuen „ANSI“-Funktionalität und mit der zu C V1.0 kompatiblen „KR“-Funktionalität.

Die gewünschte Funktionalität wird zum Übersetzungszeitpunkt mit folgender Compileroption ausgewählt:

```
SOURCE-PROPERTIES=PAR(LIBRARY-SEMANTICS=STD|V1-COMPATIBLE)
```

Die Auswahl der KR-Funktionalität (V1-COMPATIBLE) ist nur in den Übersetzungsmodi KR und ANSI möglich. In den Übersetzungsmodi STRICT-ANSI und CPLUSPLUS wird die Angabe V1-COMPATIBLE ignoriert und automatisch STD angenommen.
Die KR- bzw. ANSI-Funktionalität gilt für die Aufrufe aller Bibliotheksfunktionen einer Übersetzungseinheit.

Achtung

Wird in mehreren, getrennt übersetzten Quellprogrammen dieselbe Datei verarbeitet, müssen diese Quellprogramme mit dem gleichen LIBRARY-SEMANTICS-Parameter übersetzt werden!

Bei Programmentwicklungen in der POSIX-Shell kann die KR-Funktionalität nicht eingeschaltet werden, d.h. die Ein-/Ausgabefunktionen werden generell mit ANSI-Funktionalität ausgeführt.

Ab C/C++ V3.0 steht die KR-Funktionalität nicht mehr zur Verfügung.

Die Unterschiede zwischen KR- und ANSI-Funktionalität sind im Folgenden aufgeführt.

KR-Funktionalität

1. Standardattribute von Textdateien
   Wird eine nicht vorhandene Textdatei neu angelegt, wird standardmäßig eine SAM-Datei mit variabler Satzlänge erstellt.

2. Position des Lese-/Schreibzeigers im Anfügemodus
   Wenn der Lese-/Schreibzeiger in einer Datei, die im Anfügemodus geöffnet wurde, explizit vom Dateiende wegpositioniert wurde (`rewind()`, `fsetpos()`, `fseek()`, `lseek()`), wird er nur beim Schreiben mit der Elementarfunktion `write()` automatisch ans Ende der Datei positioniert.

   Wenn eine Datei im Anfügemodus und zum Lesen geöffnet wurde, ist sie nach dem Öffnen auf das Dateiende positioniert. Der alte Inhalt bereits vorhandener Dateien bleibt erhalten.

3. ISAM-Dateien (Pufferleerung)
   Wenn die Daten einer ISAM-Datei im Puffer nicht mit einem Neue-Zeile-Zeichen enden, bewirkt das Schreiben in die externe Datei einen Satzwechsel. Nachfolgende Daten werden in einen neuen Satz geschrieben.

4. `ungetc()`
   Beim Schreiben des Pufferinhalts in die externe Datei werden die Originaldaten verändert, wenn an Stelle des zuvor eingelesenen Zeichens ein anderes Zeichen in den Puffer zurückgestellt wurde.

5. Auswertung des Tabulatorzeichens (`\t`)
   Bei der Ausgabe in Textdateien von FCBTYP SAM und ISAM wird das Tabulatorzeichen standardmäßig in die entsprechende Anzahl Leerzeichen umgesetzt.

6. `fprintf()`, `printf()`, `sprintf()`, `fscanf()`, `scanf()`, `sscanf()`
   Die ANSI-Erweiterungen der Formatierungs- und Umwandlungszeichen stehen nicht zur Verfügung. Es gilt die Syntax und Semantik der Vorgängerversion.

7. `vfprintf()`, `vprintf()`, `vsprintf()`
   Das Umwandlungszeichen `L` kann nicht verwendet werden, da im KR-Modus der Typ `long double` nicht unterstützt wird.

ANSI-Funktionalität

1. Standardattribute von Textdateien
   Wird eine nicht vorhandene Textdatei neu angelegt, wird standardmäßig eine ISAM-Datei mit variabler Satzlänge erstellt.

2. Position des Lese-/Schreibzeigers im Anfügemodus
   Wenn eine Datei, die im Anfügemodus geöffnet wurde, explizit vom Dateiende wegpositioniert wurde (`rewind()`, `fsetpos()`, `fseek()`, `lseek()`), wird der aktuelle Lese-/Schreibzeiger bei allen Schreibfunktionen ignoriert und automatisch an das Ende der Datei positioniert.

   Wenn eine Datei im Anfügemodus und zum Lesen geöffnet wurde, ist sie nach dem Öffnen auf Dateianfang positioniert. Der alte Inhalt bereits vorhandener Dateien bleibt erhalten.

3. ISAM-Dateien (Pufferleerung)
   Wenn die Daten einer ISAM-Datei im Puffer nicht mit einem Neue-Zeile-Zeichen enden, bewirkt das Schreiben in die externe Datei keinen Satzwechsel. Nachfolgende Daten verlängern den Satz in der Datei. Beim Lesen einer ISAM-Datei werden daher nur Neue-Zeile-Zeichen eingelesen, die vom Programm explizit geschrieben wurden.

   Wenn das Lesen aus einer beliebigen Textdatei eine Datenübertragung von der externen Datei in den C-internen Puffer notwendig macht, werden die noch in Puffern zwischengespeicherten Daten aller ISAM-Dateien automatisch in die Dateien hinausgeschrieben.

4. `ungetc()`
   Beim Schreiben des Pufferinhalts in die externe Datei werden die Originaldaten nicht verändert, wenn an Stelle des zuvor eingelesenen Zeichens ein anderes Zeichen in den Puffer zurückgestellt wurde. Es werden stets die Originaldaten vor dem `ungetc`-Aufruf in die externe Datei geschrieben.

5. Auswertung des Tabulatorzeichens (`\t`)
   Bei der Ausgabe in Textdateien von FCBTYP SAM und ISAM wird das Tabulatorzeichen standardmäßig nicht in die entsprechende Anzahl Leerzeichen umgesetzt, sondern als Textzeichen (EBCDIC-Wert) in die Datei geschrieben.

Fachwörter

In diesem Verzeichnis sind die wichtigsten Begriffe dieses Handbuchs in alphabetischer Reihenfolge aufgeführt und erklärt.

8-Bit-Transparenz
8-bit-transparency
: Die Fähigkeit einer Software-Komponente, 8-Bit-Zeichen zu verarbeiten, ohne sie zu verändern, oder einen Teil des Zeichens so zu benutzen, dass dies inkompatibel mit dem aktuellen Zeichensatz ist.

Abrechnungsnummer
account number
: *BS2000:*
Bezeichnet ein Abrechnungskonto für die zugehörige Benutzerkennung. Mehreren Benutzerkennungen kann dieselbe Abrechnungsnummer zugewiesen werden. Eine Benutzerkennung kann über maximal 60 Abrechnungsnummern verfügen. Die Abrechnungsnummer wird bei LOGON und ENTER-JOB ausgewertet.

absoluter Pfadname
absolute pathname
: Der Pfadname, der beim Root-Verzeichnis des POSIX-Dateisystems beginnt und zu einer bestimmten Datei oder einem bestimmten Dateiverzeichnis führt. Jede Datei und jedes Dateiverzeichnis hat einen eindeutigen absoluten Pfadnamen (siehe `Pfadnamen-Auflösung`).

Adresse
address
: Im Allgemeinen eine Zahl zur Angabe eines Speicherplatzes.

Adressraum
address space
: Der Speicherbereich, auf den ein Prozess zugreifen kann.

aktuelles Dateiverzeichnis
current directory

Ein einem Prozess zugeordnetes Dateiverzeichnis. Dieses Dateiverzeichnis wird bei der Pfadnamen-Auflösung für solche Pfadnamen verwendet, die nicht mit einem Schrägstrich (`/`) beginnen.

Alias-Name
alias name

Ein Wort, das nur Unterstriche (_), Ziffern und alphabetische Zeichen des portablen Zeichensatzes sowie die `!`, `%`, `@` enthalten darf. Andere Implementierungen können auch andere Zeichen innerhalb eines Alias-Namen als Erweiterung zulassen.

anzeigen (auf dem Bildschirm)
display

Eine Ausgabe auf die Terminal-Gerätedatei. Die Ausgabe erscheint auf dem Bildschirm des Monitors. Wird die Ausgabe nicht auf ein Terminal gelenkt, ist das Ergebnis undefiniert.
Gemäß XPG4 werden die Benennungen „anzeigen“ und „schreiben“ genau unterschieden. Unter „anzeigen“ wird eine Ausgabemethode auf das Terminal verstanden, die nicht spezifiziert ist. Häufig werden dazu `termcap` oder `terminfo` benutzt; dies ist jedoch nicht erforderlich. Von „schreiben“ wird gesprochen, wenn ein Dateideskriptor benutzt wird und die Ausgabe umlenkbar ist. Wird jedoch direkt auf ein Terminal geschrieben, ohne dass umgelenkt wurde, kann der Benutzer oder eine Testsuite nicht abfragen, ob ein Dateideskriptor benutzt wurde oder nicht. Deshalb ist der Gebrauch eines Dateideskriptors nur erforderlich, wenn die Ausgabe umgelenkt wird.

Äquivalenzklasse
equivalence class

Eine Menge von Zeicheneinheiten mit derselben primären Sortierpriorität. Beispielsweise haben alle nachfolgenden Buchstaben denselben Grundbuchstaben, unterscheiden sich jedoch durch ihren Akzent: á, à, â, ä, ã, å. Die Vergleichsfolge der Zeicheneinheiten in einer Äquivalenzklasse wird durch die Sortierpriorität bestimmt, die jeder der Stufen zugewiesen wird, die der primären Sortierpriorität nachfolgen.

Argument
argument

In der Shell ist ein Argument ein Parameter, der an ein Kommando übergeben wird. Dieser Parameter ist gleichbedeutend mit einer einzelnen Zeichenkette im Vektor `argv`, die durch eine der `exec`-Funktionen erzeugt wurde. Ein Argument kann eine Option, ein Optionsargument oder ein Operand sein, die dem Kommandonamen folgen.
In der Sprache C ist ein Argument eine Zeichenkette, die Daten an eine Funktion übergibt. Die Argumente einer Funktion werden in runden Klammern angegeben, die auf den Funktionsnamen folgen. Die Anzahl der Argumente kann Null sein. Zwei oder mehr Argumente werden durch Kommas getrennt. Die Definition einer Funktion beschreibt die Anzahl und die Datentypen der Argumente.

Auftragssteuerung
job control

Die Möglichkeit, die Ausführung einzelner Prozesse zu stoppen (oder auszusetzen) und zu einem späteren Zeitpunkt fortzusetzen. Der Benutzer verwendet diese Fähigkeit typischerweise über die interaktive Schnittstelle, die vom Ein-/Ausgabetreiber des Terminals und einem Kommando-Interpreter gemeinsam angeboten wird.

Auftragssteuerungsnummer
job control ID

Eine Zugriffsmöglichkeit auf einen Auftrag. Die Auftragssteuerungsnummer kann eine der folgenden Formen haben:

Auftragssteuernummer	**Bedeutung**
%%	aktueller Auftrag
%+	aktueller Auftrag
%-	vorhergehender Auftrag
%*n*	Auftragsnummer *n*
%*string*	Auftrag, dessen Kommando mit *string* beginnt.
%?*string*	Auftrag, dessen Kommando *string* enthält

Ausdruck
expression

Ein mathematisches oder logisches Symbol oder eine sinnvolle Kombination dieser Symbole.

ausführbare Datei
executable file

Eine normale Datei, die als neues Prozessabbild von den Funktionen der `exec`-Familie akzeptiert wird, das Ausführrecht hat und damit wie ein Kommando aufgerufen werden kann. Die als Standard-Kommandos beschriebenen Compiler können ausführbare Dateien erzeugen. Andere, hier nicht beschriebene Methoden, ausführbare Dateien zu erzeugen, können ebenso versorgt werden. Das interne Format einer ausführbaren Datei ist nicht spezifiziert, aber eine konforme Anwendung erkennt, dass eine ausführbare Datei keine Textdatei ist.

Authentisierung
authentication

Überprüfung der Angaben eines Benutzers beim Systemzugang. Die Benutzerattribute „Benutzerkennung“ und „Kennwort“ werden gegen die Einträge im Benutzerkennungs-Katalog geprüft.

Benutzer
user

Ein Repräsentant einer Benutzerkennung. Der Begriff Benutzer ist ein Synonym für Personen, Anwendungen, Verfahren etc., die über eine Benutzerkennung Zugang zum Betriebssystem erhalten können.

Benutzerattribute
user attributes

Alle Merkmale einer Benutzerkennung, die im Benutzerkennungskatalog hinterlegt sind.

Benutzerdatenbank
user database

Eine Systemdatenbank, die ein herstellerabhängiges Format hat und die mindestens die nachfolgenden Informationen für jede Benutzerkennung enthält: Benutzername, numerische Benutzkennung, numerische Anfangsbenutzerkennung, Anfangsarbeitsverzeichnis, Anfangsbenutzerprogramm.
Die numerische Anfangsbenutzerkennung wird von dem Dienstprogramm `newgrp` benutzt. Alle anderen Umstände, unter denen Anfangswerte wirksam sind, sind herstellerabhängig.

Benutzergruppe
user group

Eine Zusammenfassung einzelner Benutzer unter einem Namen (Benutzergruppenkennung).

Benutzerkatalog
user catalog

siehe Benutzerkennungskatalog.

Benutzerkennung
login name

BS2000:
Maximal acht Zeichen langer Name, der im Benutzerkennungskatalog eingetragen wird. Anhand der Benutzerkennung wird der Benutzer beim Systemzugang identifiziert. Alle Dateien und Jobvariablen werden unter einer Benutzerkennung eingerichtet. Die Namen der Dateien und Jobvariablen werden mit der Benutzerkennung im Dateikatalog hinterlegt.

Benutzerkennungskatalog
join file

Eine Datei, die die Benutzerattribute aller Benutzerkennungen eines Pubsets bzw. eines Rechners enthält.

Benutzerklasse Andere
file other class

Die Eigenschaft einer Datei, die das Zugriffsrecht für einen Prozess anzeigt, der mit der Benutzernummer und Gruppennummer eines Prozesses verbunden ist. Ein Prozess gehört zur Benutzerklasse Andere einer Datei, wenn der Prozess nicht der Benutzerklasse Eigentümer oder der Benutzerklasse Gruppe angehört.

Benutzerklasse Eigentümer
file owner class

Die Eigenschaft einer Datei, die das Zugriffsrecht für einen Prozess anzeigt, der mit der Benutzernummer eines Prozesses verbunden ist.
Ein Prozess gehört zur Benutzerklasse Eigentümer einer Datei, wenn die effektive Benutzernummer des Prozesses zur Benutzernummer der Datei passt. Von kompatiblen Implementierungen können andere Mitglieder dieser Klasse definiert werden.

Benutzerklasse Gruppe
file group class

Ein Prozess gehört zur Benutzerklasse Gruppe einer Datei, wenn der Prozess nicht der Benutzerklasse Eigentümer angehört und die effektive Gruppennummer oder eine der zusätzlichen Gruppennummern des Prozesses zur Gruppennummer der Datei passt. Von kompatiblen Implementierungen können andere Mitglieder dieser Klasse definiert werden.

Benutzername
user name

Eine Zeichenkette, mit der der Benutzer identifiziert wird, wie in der Benutzerdatenbank beschrieben. Um portabel zu XSI-konformen Systemen zu sein, muss der Wert aus Zeichen des portablen Zeichensatzes für Dateinamen zusammengesetzt sein. Der Bindestrich darf nicht als erstes Zeichen eines portablen Benutzernamens verwendet werden.

Benutzernummer
user ID

Eine nichtnegative ganze Zahl, durch die ein Systembenutzer identifiziert wird. Wenn die Identität eines Benutzers einem Prozess zugeordnet wird, dann wird auf eine Benutzernummer als reale, effektive oder gesicherte Benutzernummer zugegriffen.

Benutzerrechte
user privileges

BS2000:
Alle an eine Benutzerkennung vergebenen und im Benutzerkennungskatalog hinterlegten Attribute, die die Rechte des Benutzers festlegen.

Benutzerverwaltung
user administration

siehe `Systemglobale Benutzerverwaltung`.

Bibliothek
library

Eine Sammlung von statisch gebundenen Objektdateien oder von Quelldateien, die dynamisch gebunden werden können (gemeinsam nutzbare Bibliothek). Die einzelnen Dateien einer Bibliothek enthalten jeweils den Programmtext für eine oder mehrere zusammenhängende Funktionen. Wird im Quellcode eine entsprechende Funktion aufgerufen, muss die jeweilige Objektdatei eingebunden werden (siehe `Include-Datei`). Beim Binden muss die Bibliothek angegeben werden. Die Datei, die die verwendete Bibliotheksfunktion enthält, wird dann in den Quellcode der Anwendung kopiert.

Bildschirmanzeige
display

siehe `anzeigen`.

Binärdatei
binary file

Eine geordnete Folge von Bytes. Die mit den C-Ausgabefunktionen geschriebenen Daten werden 1:1 in die Datei übernommen. Im Unterschied zu Textdateien werden Steuerzeichen für Zeilenvorschub und Tabulatoren nicht umgesetzt (siehe `Textdatei`), sondern als entsprechende EBCDIC-Werte abgebildet. Daten, die aus einer Binärdatei eingelesen werden, entsprechen daher genau den Daten, die ursprünglich in die Datei geschrieben wurden.
Binärdateien mit stromorientierter Ein-/Ausgabe sind: katalogisierte PAM-Dateien, temporäre PAM-Dateien (INCORE), katalogisierte SAM-Dateien, die mit `fopen()` bzw. `freopen()` im Binärmodus eröffnet wurden.
Binärdateien mit Satz-Ein-/Ausgabe sind: katalogisierte ISAM-Dateien, katalogisierte SAM-Dateien, katalogisierte PAM-Dateien, die mit den Funktionen `fopen()` bzw. `freopen()` im Binärmodus und mit dem Zusatz "`type=record`" geöffnet wurden.
Der Binärmodus kann nur mit den Funktionen `fopen()` bzw. `freopen()` angegeben werden. Mit den elementaren Funktionen `open()` und `creat()` werden SAM- und ISAM-Dateien stets als Textdateien geöffnet.

blockorientierte Gerätedatei
block special file

Eine Gerätedatei für blockorientierte Ein-/Ausgabegeräte. Sie unterscheidet sich von einer Gerätedatei für zeichenorientierte Geräte dadurch, dass sie den Zugriff auf das Gerät in einer Art und Weise bietet, die die Hardware-Eigenschaften des Gerätes verbirgt.

Blockterminal
block-mode terminal

Ein Terminal, das keine zeichenweisen Ein- und Ausgabe-Operationen unterstützt.

Dämonprozess
daemon

Ein Hintergrundprozess, der, einmal gestartet, seine Aktivitäten für den Benutzer unbemerkt verrichtet. Er wird erst beim Ausschalten des Rechners beendet. Bekanntestes UNIX-Beispiel ist der Drucker-Dämonprozeß, der dafür sorgt, dass eine Datei ausgedruckt wird, während der Benutzer bereits wieder arbeitet.

Datei
file

Ein Objekt, auf das geschrieben und/oder von dem gelesen werden kann. Eine Datei wird bei UNIX über einen Indexeintrag identifiziert und besitzt bestimmte Attribute, einschließlich der Zugriffsrechte und des Dateityps. Dateitypen schließen normale Dateien, Gerätedateien für zeichen- und blockorientierte Geräte, FIFO-Gerätedateien und Dateiverzeichnisse ein. Eine normale Datei enthält Text, Daten, Programme oder sonstige Informationen. Eine Gerätedatei bezeichnet ein Gerät oder einen Teil eines Gerätes, wie zum Beispiel ein Laufwerk oder eine Festplattenpartition. Ein Dateiverzeichnis enthält andere Dateien.

BS2000:
Sätze, die zueinander in Beziehung stehen, werden in einer benannten Einheit, der Datei, zusammengefasst. Dateien sind beispielsweise: konventionelle Ein-/Ausgabedaten von Programmen, Lademodule, Textinformation, die mit einem Editor erstellt und verarbeitet wird.

Dateihierarchie
file hierarchy

Die hierarchische Struktur, in der Dateien im System organisiert sind.
Alle Knoten, die keine Blätter sind, sind Dateiverzeichnisse (nichtterminale Knoten). Alle Knoten, die Blätter sind, sind Dateien beliebigen Dateityps (terminale Knoten). Es können sich mehrere Dateiverzeichniseinträge auf dieselbe Datei beziehen.

Dateibeschreibung
file description

Ein Objekt, das Daten darüber enthält, wie ein Prozess oder eine Gruppe von Prozessen auf eine Datei zugreifen. Jeder Dateideskriptor verweist auf genau eine Dateibeschreibung. Auf eine Dateibeschreibung aber kann mehr als ein Dateideskriptor verweisen. Die Dateiposition, der Dateimodus und die Zugriffsarten auf diese Datei sind Attribute einer Dateibeschreibung.

Dateideskriptor
file descriptor

Je Prozess genau eine positive ganze Zahl, die dazu benutzt wird, eine eindeutige Beziehung zwischen einem Prozess und einer offenen Datei für den Zugriff herzustellen. Der Wert eines Dateideskriptors liegt im Bereich zwischen 0 und `{OPEN_MAX}`. Ein Prozess kann nicht mehr als `{OPEN_MAX}` Dateideskriptoren gleichzeitig offen haben. Dateideskriptoren können auch dafür genutzt werden, einen Meldungskatalog-Deskriptor und Dateiverzeichnisströme zu implementieren. Siehe auch `Dateibeschreibung` und `{OPEN_MAX}` in der Include-Datei `limits.h`.

Dateimodus
file mode

Eine Ansammlung von Attributen, die den Dateityp und die Zugriffsrechte der Datei angeben (siehe Include-Datei sys/stat.h).

Dateiname
file name

Ein Name, der aus 1 bis {NAME_MAX} Bytes besteht und eine Datei benennt. Die Zeichen, die einen Namen bilden, können aus dem gesamten Zeichensatz gewählt werden, mit Ausnahme der Zeichen Nullbyte (\0) und Schrägstrich (/). Die Dateinamen . und .. haben eine besondere Bedeutung (siehe Pfadnamen-Auflösung). Dateinamen werden aus dem Zeichensatz für portable Dateinamen zusammengesetzt, da die Verwendung anderer Zeichen in bestimmten Zusammenhängen mehrdeutig sein kann. Beispielsweise kann die Verwendung eines Doppelpunktes (:) in einem Pfadnamen mehrdeutig sein, wenn dieser Pfadname in einer PATH-Definition enthalten ist (siehe Zeichensatz für portable Dateinamen).

Dateinummer
file serial number

Ein in einem Dateisystem eindeutiger Bezeichner für eine Datei.

Dateiposition
file offset

Die Dateiposition gibt an, wieviele Bytes vom Dateianfang entfernt die nächste Ein- oder Ausgabeoperation beginnt (erstes Byte = 1). Jede Dateibeschreibung, die zu einer normalen Datei, einer Gerätedatei für blockorientierte Geräte oder einem Dateiverzeichnis gehört, hat eine Dateiposition. Eine Gerätedatei für ein zeichenorientiertes Gerät, das kein Terminal ist, kann eine Dateiposition haben. Es gibt keine Position in Pipes und FIFOs.

Dateistatus
file status

Der aktuelle Zustand einer Datei.

Dateisystem
file system

Eine Ansammlung von Dateien und bestimmter Attribute von Dateien. Ein UNIX-Dateisystem ist hierarchisch aufgebaut (siehe Dateihierarchie). Es bietet den Namensbereich für Dateinummern, die sich auf diese Dateien beziehen.

Dateiverzeichnis
directory

Eine Datei, die Dateiverzeichniseinträge mit eindeutigen Namen enthält (siehe Dateinamen). Sie wird verwendet, um Dateien oder Dateiverzeichnisse zu gruppieren und zu organisieren.

Dateiverzeichniseintrag
directory entry

Ein Objekt, das einer Datei einen Namen zuordnet. Mehrere Dateiverzeichniseinträge können Namen derselben Datei zuordnen.

Dateiverzeichnisstrom
directory stream

Ein für jeden Prozess eindeutiger Wert, der benutzt wird, um auf ein offenes Dateiverzeichnis zu verweisen.

Dateizeiger
data set pointer

Ein Dateizeiger ist ein Zeiger auf eine Struktur vom Typ FILE. Er dient dazu, eine Datei mit den Standard-Zugriffsfunktionen (siehe stdio.h) zu verarbeiten. Beim Öffnen mit fopen(), fdopen(), freopen() wird einer Datei ein Dateizeiger zugewiesen. Bei weiteren Zugriffen mit fprintf(), fscanf(), fclose(), etc. wird der Dateizeiger als Dateiargument benutzt. Beim Programmstart sind die Standard-Ein-/Ausgabedateien automatisch mit folgenden Dateizeigern geöffnet: stdin (Standard-Eingabe), stdout (Standard-Ausgabe), stderr (Standard-Fehlerausgabe).

Dateizeiten-Änderung
file times update

Jeder Datei sind drei Zeitwerte zugeordnet, die geändert werden, wenn auf die Daten in dieser Datei zugegriffen wurde oder die Daten bzw. der Dateizustand verändert wurden. Diese Werte werden in der Struktur stat zurückgegeben (siehe sys/stat.h).

Für jede Funktion in diesem Handbuch, die Daten einer Datei liest oder schreibt oder den Zustand einer Datei ändert, werden die entsprechenden, zeitbezogenen Felder als "zum Ändern markiert" bezeichnet. Zu einem Änderungszeitpunkt werden alle markierten Felder mit der aktuellen Zeit besetzt und die Änderungsmarken gelöscht. Zwei solche Änderungszeitpunkte sind, wenn eine Datei nicht länger von irgendeinem Prozess geöffnet ist und wenn stat() oder fstat() für diese Datei ausgeführt werden. Sonstige Änderungszeitpunkte sind nicht festgelegt. Für Dateien in Dateisystemen, die nur zum Lesen eingehängt sind, werden keine Änderungen durchgeführt.

Dateizugriffsrechte
file access permissions

Bestandteil der Dateibeschreibung. Der Dateizugriff wird durch Bits gesteuert. Diese Bits werden bei der Erzeugung einer Datei durch Funktionen wie `open()`, `creat()`, `mkdir()` und `mkfifo()` gesetzt und durch `chmod()` geändert. Diese Bits werden von `stat()` oder `fstat()` gelesen.
Anwendungen können zusätzliche und/oder alternative Dateizugriff-Steuerungsmechanismen zur Verfügung stellen. Ein alternativer Dateizugriff-Steuerungsmechanismus verhält sich wie folgt:

- Er legt die Datei-Schutzbits für die Benutzerklassen Eigentümer, Gruppe und Andere fest.
- Er kann nur durch eine explizite Benutzeraktion auf eine Datei durch den Eigentümer der Datei oder einen Benutzer mit Sonderrechten aktiviert werden.
- Er kann für eine Datei deaktiviert werden, nachdem die Schutzbits für diese Datei durch `chmod()` geändert wurden. Die Deaktivierung des alternativen Mechanismus muss auch keine zusätzlichen, von der Implementierung definierten Mechanismen deaktivieren.

Sobald ein Prozess für eine Datei die Zugriffsrechte zum Lesen, Schreiben oder Ausführen/Durchsuchen anfordert, wird der Zugriff, sofern keine zusätzlichen Mechanismen den Zugriff verweigern, wie folgt entschieden:

Wenn ein Prozess Sonderrechte hat, gilt Folgendes:

- Wenn das Lese-, Schreib- oder Durchsuchrecht gefordert wird, dann wird der Zugriff gestattet.
- Wenn das Ausführungsrecht gefordert wird, so wird der Zugriff dann erlaubt, wenn das Ausführungsrecht zumindest einem Benutzer durch die Schutzbits oder einen anderen Zugriffssteuerungsmechanismus gewährt wird; andernfalls wird der Zugriff verweigert.

Wenn ein Prozess keine Sonderrechte hat, gilt Folgendes:

- Die Schutzbits einer Datei enthalten das Lese-, Schreib- und Ausführungs- bzw. Durchsuchrecht für die Benutzerklassen Eigentümer, Gruppe und andere Benutzer.
- Der Zugriff wird gestattet, wenn ein alternativer Zugriff-Steuerungsmechanismus nicht aktiviert ist und das Schutzbit für die geforderten Zugriffsrechte in der Benutzerklasse gesetzt ist, zu der der Prozess gehört, oder wenn ein alternativer Zugriff-Steuerungsmechanismus aktiviert ist und dieser den geforderten Zugriff erlaubt; andernfalls wird der Zugriff verweigert.

Datenstrom
stream

Ein Dateizugriffsobjekt, das Zugriff auf eine angeforderte Zeichenfolge erlaubt. Solche Objekte können mit den Funktionen `fdopen()`, `fopen()` oder `popen()` erzeugt werden und sind mit einem Dateideskriptor verbunden. Ein Datenstrom stellt einen zusätzlichen Service mit vom Benutzer auszuwählender Pufferung und formatierter Ein-/Ausgabe zur Verfügung.

Dezimalzeichen
radix character

Das Zeichen zwischen dem ganzzahligen und dem gebrochenen Teil einer Zahl.

effektive Benutzernummer
effective user ID

Ein Prozessattribut, das verwendet wird, um verschiedene Rechte zu bestimmen, einschließlich der Dateizugriffsrechte (siehe `Benutzernummer`). Dieser Wert kann sich während der Lebensdauer eines Prozesses ändern, so wie dies unter `setuid()` und `exec` beschrieben wird.

effektive Gruppennummer
effective group ID

Ein Prozessattribut, das verwendet wird, um verschiedene Rechte zu bestimmen, einschließlich der Dateizugriffsrechte (siehe `Gruppennummer`). Dieser Wert kann sich während der Lebensdauer eines Prozesses ändern, und zwar so, wie dies unter `setgid()` und `exec` beschrieben wird.

Einhängepunkt
mount point

Entweder das Rootverzeichnis des Systems oder ein Dateiverzeichnis, bei dem das Feld `st_dev` der Struktur `stat` (siehe `sys/stat.h`) von seinem übergeordneten Dateiverzeichnis abweicht.

elementare Funktionen
elemantary functions

BS2000:
Als „elementar“ werden alle Funktionen bezeichnet, die eine Datei auf der Basis von Dateideskriptoren verarbeiten. Im Unterschied dazu gibt es die Standard-Ein-/Ausgabefunktionen, die alle auf der Basis von Dateizeigern arbeiten.

Außerdem lassen sich mit den elementaren Funktionen SAM-Dateien nur als Textdateien und nicht, wie mit den Standardfunktionen, auch als Binärdateien verarbeiten.
In UNIX/POSIX sind die elementaren Funktionen als Systemaufrufe realisiert und unterscheiden sich von den Standardfunktionen durch größere Systemnähe und bessere Performance. Diesen Unterschied zwischen Systemaufruf und Funktion gibt es im BS2000 nicht.

Epochenwert
epoch

Die Zeit 0 Uhr, 0 Minuten und 0 Sekunden am 1. Januar 1970 (Coordinated Universal Time).

BS2000:
Die Zeit 0 Uhr, 0 Minuten und 0 Sekunden am 1. Januar 1950.

erweiterte Sicherheitssteuerungen
extended security controls

Die Zugriffssteuerung (siehe `Dateizugriffsrechte`) und die Rechte (siehe `Sonderrechte`) wurden definiert, um herstellerabhängige, erweiterte Sicherheitssteuerungen zuzulassen. Diese erlauben einer Implementierung, Sicherheitsmechanismen anzubieten, die von den im Standard definierten verschieden sind. Diese Sicherheitsmechanismen ändern oder ersetzen die definierte Semantik der in diesem Handbuch beschriebenen Funktionen nicht.

ferner Rechner
remote machine

In einem lokalen Netz werden ferne und lokale Rechner unterschieden. Alle Rechner im Netz, an denen ein Benutzer nicht direkt arbeitet, sind für diesen Benutzer ferne Rechner. Er kann mit allen fernen Rechnern im Netz kommunizieren.

FIFO-Gerätedatei
FIFO special file

Eine Dateiart, bei der Daten auf first-in/first-out-Basis gelesen werden. Andere Eigenschaften von FIFO-Gerätedateien werden unter `lseek()`, `open()`, `read()`, `write()` und `lseek()` beschrieben.

FILE-Struktur
file structure

Einer Datei, die mit `fopen()`, `fdopen()` oder `freopen()` geöffnet wird, ist automatisch ab diesem Zeitpunkt eine bestimmte Struktur vom Typ `FILE` zugeordnet. Diese Struktur ist in `stdio.h` definiert. Sie enthält u.a. folgende Informationen über die Datei: Zeiger auf den Ein-/Ausgabepuffer, Puffergröße, Position des Lese-/Schreibzeigers, Größe der Datei.

Filter
filter

Ein Kommando, mit dem Daten von der Standard-Eingabe oder aus einer Liste von Eingabedateien gelesen und auf die Standard-Ausgabe geschrieben werden. Mit dieser Funktion werden einige Umwandlungen am Datenstrom ausgeführt.

Gegenschrägstrich
backslash

Das Zeichen \, das auch als inverser Schrägstrich bekannt ist.

Gerät
device

Ein Peripheriegerät oder ein Objekt, das von einer Anwendung wie ein Peripheriegerät behandelt wird.

Gerätedatei
special file

Eine auch als Gerätetreiber bezeichnete Datei, die als Schnittstelle zu einem Ein/Ausgabegerät (z.B. Terminal, Plattenlaufwerk, Zeilendrucker) benutzt wird.

Gerätenummer
device ID

Eine nichtnegative ganze Zahl, die verwendet wird, um ein Gerät zu identifizieren.

gesicherte Benutzernummer
saved set-user-ID

Ein Prozessattribut, das mehr Flexibilität bei der Zuweisung des Attributs effektive Benutzernummer erlaubt, so wie dies unter `setuid()` und `exec` beschrieben wird.

gesicherte Gruppennummer
saved set-group-ID

Ein Prozessattribut, das mehr Flexibilität bei der Zuweisung des Attributs effektive Gruppennummer erlaubt, so wie dies unter `setgid()` und `exec` beschrieben wird.

Gruppendatenbank
group database

Eine Systemdatenbank mit herstellerabhängigem Format, die mindestens folgende Informationen für jede Gruppennummer enthält: Gruppenname, numerische Gruppennummer und eine Liste der in der Gruppe erlaubten Benutzer. Die Liste der in der Gruppe erlaubten Benutzer wird vom Dienstprogramm `newgrp` verwendet.

Gruppenname
group name

Eine Zeichenkette zum Identifizieren einer Gruppe, wie unter Datenbankgruppe beschrieben. Um portabel zu XSI-konformen Systemen zu sein, muss der Wert aus Zeichen des portablen Zeichensatzes für Dateinamen zusammengesetzt sein. Der Bindestrich darf nicht als erstes Zeichen eines portablen Benutzernamens verwendet werden.

Gruppennummer
group ID

Eine nichtnegative ganze Zahl zum Identifizieren einer Gruppe von Systembenutzern. Jeder Systembenutzer ist Mitglied zumindest einer Gruppe. Wenn die Identität einer Gruppe einem Prozess zugeordnet wird, dann wird der Wert einer Gruppennummer als reale, effektive, zusätzliche oder gesicherte Gruppennummer angesprochen.

Hintergrund
background

Eine Methode zur Ausführung eines Programms, bei der während des Programmlaufs kein Dialog zwischen Benutzer und Rechner stattfindet. Die Shell gibt während des Programmlaufs ihre Eingabeaufforderung aus, so dass am Terminal weitere Kommandos aufgerufen werden können (siehe `Vordergrund`).

Hintergrundprozess
background process

Ein Prozess, der Mitglied einer Hintergrund-Prozessgruppe ist, und der die Ressourcen des Rechners nicht vollständig ausschöpft, sondern die gleichzeitige Durchführung von weiteren (in der Regel wichtigeren) Prozessen ermöglicht. Ein Hintergrundprozess nutzt normalerweise die Zeitnischen aus, in denen der Prozessor sonst unbeschäftigt wäre.

Hintergrund-Prozessgruppe
background process group

Jede Prozessgruppe, die Mitglied einer Sitzung ist, die eine Verbindung zu einem steuernden Terminal hergestellt hat und die keine Vordergrund-Prozessgruppe ist.

Home-Verzeichnis
home directory

Ein Dateiverzeichnis, in das der Benutzer automatisch gelangt, wenn er mit POSIX verbunden wird.

Hostrechner
host

Der Zentralrechner eines Rechnernetzes. Auf dem Hostrechner werden Programme ausgeführt, Dateien gespeichert sowie Ein- und Ausgaben gesteuert. In vielen Fällen verfügt ein leistungsfähiges Rechnernetz über mehrere Hostrechner.

Include-Datei
header file

Die Datei, die die Datendefinitionen enthält, die vom Compiler in die Quelldateien kopiert werden (siehe `Bibliothek`). Include-Dateinamen enden mit dem Suffix `.h`. Sie werden durch die `#include`-Anweisung in die Quelldateien eingebunden.

Internationalisierung
internationalization

Die Möglichkeit der Anpassung eines Computerprogramms an die verschiedenen Landessprachen, länderspezifischen Eigenheiten und verschlüsselten Zeichensätze.

Jobvariable
job variable

BS2000:
Jobvariablen sind Speicherbereiche zum Austausch von Informationen zwischen Aufträgen (Jobs) untereinander sowie zwischen Betriebssystem und Aufträgen. Sie haben einen Namen und einen Inhalt (Wert). Der Inhalt kann zur Steuerung von Aufträgen und Programmen genutzt werden. Der Benutzer kann Jobvariablen erzeugen, verändern, abfragen und löschen. Außerdem kann er das Betriebssystem anweisen, eine überwachende Jobvariable entsprechend zu setzen, wenn sich der Zustand eines Auftrags oder eines Programms ändert.

Kennwort
password

Eine Folge von Zeichen, die der Benutzer eingeben muss, um den Zugriff zu einer Benutzerkennung, einer Datei, einer Jobvariablen, einem Netzknoten oder einer Anwendung zu erhalten.

Kommando
command

Eine Anweisung an die Shell, eine bestimmte Aufgabe auszuführen (siehe Handbuch „POSIX Kommandos“).

Kommando-Interpreter
command interpreter

Eine Schnittstelle, die Eingabetextfolgen als Kommandos interpretiert. Diese Schnittstelle arbeitet an Eingabe-Datenströmen und kann interaktiv Kommandos vom Terminal anfordern oder lesen. Für Anwendungen ist es möglich, Dienstprogramme über eine Reihe von Schnittstellen aufzurufen, für die man annimmt, dass sie sich wie Kommando-Interpreter verhalten. Die am häufigsten benutzten Schnittstellen sind `sh` und `system()`, obwohl auch `popen()` und die verschiedenen Formen von `exec` ebenfalls als Interpreter verstanden werden können.

länderspezifisch
locale

Die Definition einer Untermenge in einer Benutzerumgebung, die von der Landessprache und den kulturellen Konventionen abhängt.

Langzeichen
wide-character code

Ein ganzzahliger Wert, der einem graphischen Zeichen oder einem Steuerzeichen entspricht. Alle Langzeichen eines Prozesses bestehen aus der gleichen Anzahl von Bits. Ein Langzeichen, dessen Bits alle auf Null gesetzt sind, heißt Null-Langzeichen.

Langzeichenkette
wide-character string
: Eine Folge von aneinandergrenzenden Langzeichen, einschließlich des Null-Langzeichens, mit dem die Zeichenkette abgeschlossen wird.

leere Langzeichenkette
empty wide-character string
: Eine Langzeichenkette, deren erstes Zeichen ein Null-Langzeichen ist.

leere Zeichenkette
empty string
: Eine Zeichenkette, deren erstes Byte ein Nullbyte ist.

leeres Dateiverzeichnis
empty directory
: Ein Dateiverzeichnis, das höchstens die Dateiverzeichniseinträge . und .. enthält (siehe `Punkt` und `Punkt-Punkt`).

Lese-/Schreibzeiger
file position indicator
: Der Lese-/Schreibzeiger enthält Informationen über die aktuelle Position einer Datei. Daten werden jeweils ab dieser aktuellen Position gelesen bzw.geschrieben. Die Information im Lese-/Schreibzeiger ist je nach Dateiart unterschiedlich aufgebaut.
Bei Textdateien enthält der Lese-/Schreibzeiger Informationen über den aktuellen Satz und die Position innerhalb des Satzes.

BS2000:
Bei Binärdateien mit Strom-Ein-/Ausgabe enthält der Lese-/Schreibzeiger Informationen über die Anzahl Bytes vom Dateianfang gerechnet. Der Aufbau ist für SAM- und ISAM-Dateien unterschiedlich. Die Information wird vom Laufzeitsystem intern verwendet.
Bei Binärdateien mit Satz-Ein-/Ausgabe enthält der Lese-/Schreibzeiger Informationen über die Position hinter dem zuletzt gelesenen, geschriebenen oder gelöschten Satz bzw. der Position, die durch ein unmittelbar vorangegangenes Positionieren erreicht wurde.
Bei ISAM-Dateien mit Schlüsselverdoppelung enthält der Lese-/Schreibzeiger die Position hinter dem letzten Satz einer Gruppe mit gleichen Schlüsseln, wenn einer dieser Sätze zuvor gelesen, geschrieben oder gelöscht wurde.

lokaler Rechner
local machine
: Für einen Benutzer ist immer derjenige Rechner lokal, an dem er arbeitet. Alle anderen Rechner im Rechnernetz sind dann für ihn ferne Rechner.

Lokalisierung
localization

Der Prozess, spezifische Informationen für die verschiedenen Landessprachen, länderspezifischen Eigenheiten und verschlüsselten Zeichensätze in einem Computersystem einzurichten.

Lokalität
locale

Die Konventionen eines geographischen Bereiches oder Gebietes für Datum, Zeit und Währungsformate.

Makro für den Test von Eigenschaften
feature test macro

Ein Makro, das verwendet wird, um zu entscheiden, ob eine bestimmte Menge von Eigenschaften aus einer Include-Datei eingebunden wird.

mathematischer Wertebereich
mathematical range

Die Notation [n, m] und [n, m) bezeichnet einen mathematischen Bereich. Die eckigen Klammern [und] schließen die Grenzen jeweils mit ein, die runden Klammern (und) schließen diese aus. Das heißt, wenn x aus dem Bereich [0,1] ist, dann kann dies von 0 bis einschließlich 1 sein. Wenn aber x aus dem Bereich [0,1] ist, dann kann dies von 0 bis ausschließlich 1 sein.

Multibyte-Zeichen
multi-byte character

Zeichen, das aus mehreren Bytes besteht, unabhängig davon, ob es sich um einen einfachen oder einen Langzeichensatz handelt.

Meldungskatalog
message catalog

Eine Datei oder ein Speicherbereich, der Programmmeldungen, Eingabeaufforderungen und Antworten darauf für eine bestimmte Landessprache, ein bestimmtes Gebiet und einen bestimmten Zeichensatz enthält.

Meldungskatalog-Deskriptor
message catalog descriptor

Ein je Prozess eindeutiger Wert, der verwendet wird, um einen offenen Meldungskatalog zu identifizieren.

Modus
mode

Eine Zusammenfassung von Attributen, die einen Dateityp und seine Zugriffsrechte beschreibt (siehe `Dateizugriffsrechte`).

normale Datei
regular file

Eine Datei, die eine wahlfrei zugreifbare Folge von Bytes ohne jede weitere vom System festgelegte Struktur ist.

Nullbyte
null byte

Ein Byte, in dem alle Bits auf 0 gesetzt sind.

Nullzeiger
null pointer

Dies ist der Wert, den man erhält, wenn man die Zahl 0 in einen Zeiger umwandelt, z.B. `(void *) 0`. Die Programmiersprache C garantiert, dass dieser Wert keinem gültigen Zeiger entspricht, daher wird er von vielen Funktionen verwendet, die Zeiger zurückgeben, um einen Fehler anzuzeigen.

Objektdatei
object file

Eine Datei, die den Quellcode eines Programms in Binärdarstellung enthält. Eine relokierbare Objektdatei enthält Referenzen auf Symbole, die noch nicht mit zugehörigen Definitionen verbunden sind. Eine ausführbare Objektdatei ist ein gebundenes Programm.

offene Datei
open file

Eine Datei, die aktuell einem Dateideskriptor zugeordnet ist.

Option
option

Ein Argument eines Kommandos, das den Ablauf dieses Kommandos beeinflusst. Eine Option ist ein Argumenttyp, der auf den Kommandonamen folgt und im Normalfall den übrigen Argumenten in der Kommandozeile vorangestellt ist. Eine Option beginnt üblicherweise mit einem Minuszeichen. Anzahl und Art der zulässigen Argumente sind von Kommando zu Kommando unterschiedlich. Wenn Optionen Argumente haben, werden sie durch Leerzeichen getrennt.

Optionsargument
option-argument

Ein Parameter, der nach verschiedenen Optionen angegeben ist. In manchen Fällen befindet sich ein Optionsargument in derselben Argumentzeichenkette wie die Option. In den meisten Fällen ist es ein Textargument.

Parser
parser

Ein Parser führt eine syntaktische und lexikalische Analyse eines Textes durch.

Pfadname
pathname

Eine Zeichenkette, die eine Datei identifiziert. Sie besteht aus höchstens {PATH_MAX} Bytes, einschließlich des abschließenden Nullbytes. Sie besteht aus einem optionalen führenden Schrägstrich, gefolgt von einem oder mehreren Dateinamen, die wiederum durch Schrägstriche getrennt sind, bzw. aus einem führenden Schrägstrich ohne Dateinamen. Wenn der Pfadname sich auf ein Dateiverzeichnis bezieht, dann kann er auch einen oder mehrere abschließende Schrägstriche enthalten. Mehrere Aufeinander folgende Schrägstriche werden als ein Schrägstrich behandelt. Ein Pfadname, der mit zwei Schrägstrichen beginnt, kann von einigen kompatiblen Implementierungen in besonderer Weise interpretiert werden, obwohl mehr als zwei führende Schrägstriche als ein einziger Schrägstrich behandelt werden (siehe Pfadnamen–Auflösung).

BS2000:
Jede im BS2000 katalogisierte Datei ist ebenfalls durch einen Pfadnamen eindeutig identifizierbar. Der Pfadname setzt sich zusammen aus der Katalogkennung (catid), der Benutzerkennung (userid) und einem vom Benutzer vergebenen vollqualifizierten Dateinamen (z. B.: *catid*:$*userid.dateiname*).

Pfadnamen-Auflösung
pathname resolution

Die Auflösung eines Pfadnamens wird für einen Prozess durchgeführt, um in einer Dateihierarchie zu einer bestimmten Datei zu führen. Zu einer Datei können mehrere Pfadnamen führen.
Jeder Dateiname in einem Pfadnamen befindet sich in dem Dateiverzeichnis, das durch das voranstehende Dateiverzeichnis beschrieben ist (z.B. befindet sich in dem Pfadnamen a/b die Datei b in dem Dateiverzeichnis a). Die Auflösung des Pfadnamens schlägt fehl, wenn dies nicht so ist.
Wenn der Pfadname mit einem Schrägstrich beginnt, dann wird der Vorgänger des ersten Dateinamens im Pfadnamen als das Root-Dateiverzeichnis des Prozesses angenommen. Solche Pfadnamen werden auch als absolute Pfadnamen bezeichnet.
Wenn der Pfadname nicht mit einem Schrägstrich beginnt, dann wird als Vorgänger des ersten Dateinamens im Pfadnamen das aktuelle Dateiverzeichnis des Prozesses angenommen. Solche Pfadnamen werden auch als relative Pfadnamen bezeichnet.
Die Interpretation einer Pfadnamen-Komponente hängt von den Werten {NAME_MAX} und {_POSIX_NO_TRUNC} ab, die dem Pfadnamen-Präfix dieser Komponente zugeordnet sind. Wenn eine Pfadnamen-Komponente länger als

{NAME_MAX} ist, und {_POSIX_NO_TRUNC} für den Pfadnamen-Präfix dieser Komponente aktiv ist (siehe pathconf()), dann gilt dies als Fehler. Andernfalls werden nur die ersten {NAME_MAX} Bytes der Pfadnamen-Komponente berücksichtigt. Der besondere Dateiname . verweist auf das Dateiverzeichnis, das durch seinen Vorgänger angegeben wird. Der besondere Dateiname .. verweist auf das übergeordnete Dateiverzeichnis seines Vorgängers. Als Sonderfall kann .. im Root-Dateiverzeichnis auf das Root-Dateiverzeichnis selbst verweisen.
Ein Pfadname, der nur aus einem einzelnen Schrägstrich besteht, benennt das Root-Dateiverzeichnis des Prozesses. Ein leerer Pfadname ist ungültig.

Pfadnamen-Präfix
pathname prefix

Ein Pfadname, der optional mit einem Schrägstrich endet, und der auf ein Dateiverzeichnis verweist.

Pipe
pipe

Ein Objekt auf das über einen der beiden Dateideskriptoren zugegriffen wird, die durch die Funktion pipe() erzeugt worden sind. Einmal erzeugt, können diese Dateideskriptoren das Objekt manipulieren, und es verhält sich genauso wie eine FIFO-Gerätedatei, wenn in dieser Weise darauf zugegriffen wird. Es hat im Dateibaum keinen Namen.

Portabilität
portability

Die Fähigkeit eines Programms, unverändert auf unterschiedlichen Betriebssystemen ablaufen zu können. Sie wird durch die Verwendung standardisierter, offener Programmschnittstellen erreicht, die auf einer Vielzahl von Plattformen angeboten werden.

portabler Pfadname
portable pathname

Damit ein Pfadname unter kompatiblen Systemen portabel ist, sollte er aus höchstens {PATH_MAX} Bytes bestehen, einschließlich des abschließenden Nullbytes. Es sollte ein Pfadname sein, der aus einem optionalen, führenden Schrägstrich sowie keinem oder mehr portablen Dateinamen besteht, die durch Schrägstriche voneinander getrennt sind.

portabler Zeichensatz
portable character set

Die Erfassung von Zeichen, die in allen Lokalitäten, die durch XSI-konforme Systeme unterstützt werden, vorhanden sein müssen:
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 ! # % ^ & * ( ) _ + - = { } [ ]
: „ ~ ; ´ , `, < > ? , . | \ / @ $

POSIX-Dateisystem
POSIX file system

Ein Dateisystem im BS2000 mit der Struktur eines UNIX-Dateisystems (UFS). Es stellt eine hierarchisch gegliederte Menge von Dateiverzeichnissen und Dateien (POSIX-Dateien) dar, die in einer Baumstruktur angeordnet sind. Die Wurzel dieser Baumstruktur ist das Root-Verzeichnis (`/`). Alle anderen Dateiverzeichnisse sind Zweige, die vom Root-Verzeichnis ausgehen. Jede Datei eines Dateisystems ist über genau einen absoluten Pfad des Dateisystems erreichbar; relative Pfade sind beliebig viele denkbar.
Der Unterschied zwischen einem POSIX- und einem UNIX-Dateisystem besteht im Ablageort: Bei einem UNIX-Dateisystem ist der Ablageort ein physikalisches Gerät, bei einem POSIX-Dateisystem eine PAM-Behälterdatei.

POSIX-Shell
POSIX shell

Ein portiertes SINIX-Systemprogramm, das die Kommunikation zwischen dem Benutzer und dem System übernimmt. Die POSIX-Shell ist ein Kommando-Interpreter. Sie übersetzt die eingegebenen POSIX-Kommandos in eine Sprache, die das System verarbeiten kann.
Wenn beim Benutzerattribut „Programm“ die POSIX-Shell eingetragen ist, wird die POSIX-Shell gestartet, sobald sich der Benutzer an einem fernen Rechner an POSIX angeschlossen hat (`rlogin`).

Protokoll
protocol

Regeln für den Datenaustausch zwischen zwei Rechnern, die die Art der elektrischen Verbindung, das Datenformat sowie die Abfolge der Daten bestimmen.

Prozess
process

Ein Adressraum und einzelner Programmcode, der in diesem Adressraum ausgeführt wird, sowie die dafür benötigten Betriebsmittel des Systems. Ein Prozess wird von einem anderen Prozess durch den Aufruf der Funktion `fork()` erzeugt. Der Prozess, der `fork()` aufruft, heißt Vaterprozess und der neue, durch `fork()` erzeugte Prozess, heißt Sohnprozess.

Prozess, Lebensdauer
process lifetime

Der Zeitraum, der mit der Erzeugung des Prozesses beginnt, und der endet, wenn die Prozessnummer an das System zurückgegeben wird.
Nachdem ein Prozess mit der Funktion `fork()` erzeugt wurde, gilt er als aktiv. Sein Steuerbereich und Adressraum existieren, bis er sich beendet. Dann gelangt er in einen inaktiven Zustand, in dem bestimmte Betriebsmittel an das System zurückgegeben werden können, obwohl einige Betriebsmittel, wie z.B. die Prozessnummer, noch immer verwendet werden. Wenn ein anderer Prozess eine der Funktionen `wait()` oder `waitpid()` für einen inaktiven Prozess ausführt, dann werden die übrigen Betriebsmittel an das System zurückgegeben. Das letzte an das System zurückgegebene Betriebsmittel ist die Prozessnummer. Zu diesem Zeitpunkt endet die Lebensdauer des Prozesses.

Prozessgruppe
process group

Eine Gruppe von Prozessen, die es erlauben, verwandten Prozessen Signale zu senden. Jeder Prozess im System ist Mitglied einer Prozessgruppe, die durch eine Prozessgruppennummer identifiziert wird. Diese Gruppierung von Prozessen erlaubt es, verwandten Gruppen von Prozessen Signale zu senden. Ein neu erzeugter Prozess gehört der Prozessgruppe seines Erzeugers an.

Prozessgruppe, Lebensdauer
process group lifetime

Ein Zeitraum, der dann beginnt, wenn eine Prozessgruppe erzeugt wird, und der dann endet, wenn der letzte Prozess dieser Prozessgruppe sie verlässt. Das Verlassen einer Prozessgruppe erfolgt entweder durch die Beendigung des Prozesses oder durch den Aufruf einer der Funktionen `setsid()` oder `setpgid()`.

Prozessgruppennummer
process group ID

Ein eindeutiger Bezeichner während der Lebensdauer eines Prozesses. Eine Prozessgruppennummer ist eine positive ganze Zahl und kann vom System erst wieder verwendet werden, wenn die Lebensdauer der Prozessgruppe endet.

Prozessgruppenleiter
process group leader

Ein Prozess, dessen Prozessnummer und Prozessgruppenummer identisch sind.

Prozessnummer
process ID

Ein eindeutiger Bezeichner eines Prozesses. Eine Prozessnummer ist eine positive ganze Zahl und kann vom System erst wieder verwendet werden, wenn die Prozesslebensdauer endet. Wenn eine Prozessgruppe existiert, deren Prozessgruppennummer dieselbe ist wie die Prozessnummer, kann die Prozessnummer erst wieder verwendet werden, wenn die Lebensdauer einer Prozessgruppe endet. Nur ein Systemprozess hat die Gruppennummer 1.

Pthread
pthread

Ein Thread ist ein Programmteil, der parallel zu anderen Teilen abläuft. Innerhalb eines Prozesses können mehrere Threads parallel ablaufen; ein Prozess besteht jedoch mindestens aus einem Thread. Im Unterschied zu Prozessen teilen sich alle Threads eines Programms einen gemeinsamen Adressraum. Bei den Pthreads im BS2000 können die Threads eines Prozesses, anders als bei z. B. DCE-Threads, auf mehrere Tasks verteilt werden.

Puffer
buffer

Ein Speicherbereich, in dem Daten zeitweise gespeichert werden.

Pufferung
buffering

Bei allen Ausgabefunktionen, die Daten in Textdateien und Binärdateien mit stromorientierter Ein-/Ausgabe schreiben (`printf()`, `putc()`, `fwrite()` etc.), werden die Daten in einem Puffer zwischengespeichert und erst in die externe Datei geschrieben, wenn ein bestimmtes Ereignis eintritt. Dieses unterscheidet sich bei Text- und Binärdateien.

Punkt
dot

Ein Dateiname, der einen einzelnen Punkt (.) enthält, steht für das aktuelle Dateiverzeichnis (siehe `Pfadnamen-Auflösung`).

Punkt-Punkt
dot-dot

Ein Dateiname, der nur zwei Punkte (..) enthält, steht für das übergeordnete Dateiverzeichnis (siehe `Pfadnamen-Auflösung`).

reale Benutzernummer
real user ID

Das Prozessattribut, das zum Zeitpunkt der Erzeugung eines Prozesses denjenigen Benutzer identifiziert, der diesen Prozess erzeugt hat (siehe `Benutzernummer`). Dieser Wert kann während der Lebensdauer des Prozesses verändert werden, wie dies unter `setuid()` beschrieben ist.

reale Gruppennummer
real group ID

Das Prozessattribut, das zum Zeitpunkt der Erzeugung eines Prozesses die Gruppe des Benutzers identifiziert, der diesen Prozess erzeugt hat (siehe `Gruppennummer`). Dieser Wert kann sich während der Lebensdauer des Prozesses ändern, so wie dies unter `setgid()` beschrieben ist.

regulärer Ausdruck
regular expression

Ein Suchmuster, das nach bestimmten Regeln zusammengesetzt ist (siehe Abschnitt „Reguläre Ausdrücke“ im Handbuch „POSIX Kommandos“).

relativer Pfadname
relative pathname

Ein Zugriffspfad für eine Datei oder ein Dateiverzeichnis, der von der Position des aktuellen Dateiverzeichnisses innerhalb des Dateisystems ausgeht. Relative Pfadnamen beginnen nicht mit einem Schrägstrich (`/`) (siehe `Pfadnamen-Auflösung`).

Root-Verzeichnis
root directory

Ein Dateiverzeichnis, das einem Prozess zugeordnet ist, und das bei der Pfadnamen-Auflösung für Pfadnamen verwendet wird, die mit einem Schrägstrich beginnen.

Rücksetzzeichen
backspace character

Ein Zeichen, das bewirkt, dass im Ausgabestrom das Drucken oder Anzeigen in einer Spaltenposition erfolgt, die sich vor der Spaltenposition befindet, in der gedruckt oder angezeigt werden sollte. Ist die Spaltenposition, in der gedruckt oder angezeigt werden sollte, die erste Spalte der Zeile, so ist das Verhalten undefiniert. Das Rücksetzzeichen wird in C mit `\b` angegeben. Es ist undefiniert, ob das Zeichen die exakte Folge ist, die durch das System an ein Ausgabegerät übertragen wird, um den Rückschritt abzusetzen.

Seitenvorschubzeichen
form-feed character

Ein Zeichen, das anzeigt, das im Ausgabestrom das Drucken auf ein Ausgabegerät auf der nächsten Seite beginnen soll. Das Seitenvorschubzeichen wird in C mit `\f` angegeben. Ist das Seitenvorschubzeichen nicht das erste Zeichen in einer Ausgabezeile, so ist das Ergebnis undefiniert. Es ist ebenfalls undefiniert, ob das Zeichen die exakte Folge ist, die durch das System an ein Ausgabegerät übertragen wird, um den Seitenvorschub abzusetzen.

Satzorientierte Ein-/Ausgabe
record oriented I/O

BS2000:
Satzorientierte Ein-/Ausgabe bedeutet, dass sich der Lese-/Schreibzeiger der Datei jeweils nur auf den Beginn eines Satzes bzw. Blockes positionieren lässt. Satzorientierte Ein-/Ausgabe ermöglicht eine der BS2000-Struktur angepasste performante Dateiverarbeitung. Die Einheit für einen Ein-/Ausgabe-Funktionsaufruf ist stets ein Satz bzw. Block. Satzorientiert können katalogisierte SAM-, ISAM- und PAM-Dateien verarbeitet werden. Es stehen zusätzliche Funktionen zur Verfügung wie Löschen und Einfügen von Sätzen, Zugriff auf Schlüssel in ISAM-Dateien.

Schrägstrich
slash

Ein Begriff der dazu verwendet wird, das einzelne Zeichen (`/`) darzustellen. Dieses Zeichen ist im Amerikanischen auch unter dem Namen `solidus` bekannt.

schreibgeschütztes Dateisystem
read-only file system

Ein Dateisystem, das eine von der Implementierung definierte, charakteristische einschränkende Änderung besitzt.

Schutzattribute
security attributes

BS2000:
Die sicherheitsrelevanten Eigenschaften eines Objekts (Datei, Jobvariable etc.), die die Art und Möglichkeit des Zugriffs auf dieses Objekt festlegen. Für Dateien gibt es beispielsweise folgende Schutzattribute: ACCESS/USER-ACCESS, SERVICE-Bit, AUDIT-Attribut, RDPASS, WRPASS, EXPASS, RETPD, BACL, ACL und GUARD.

Schutzbits einer Datei
file permission bits

Eine Information über eine Datei, die zusammen mit anderen Daten benutzt wird, um zu entscheiden, ob ein Prozess Lese-, Schreib- oder Ausführungsrecht/Durchsuchrecht für eine Datei besitzt. Die Bits sind in drei Abschnitte eingeteilt: Eigentümer, Gruppe und andere Benutzer. Jeder Abschnitt wird in Verbindung mit der entsprechenden Benutzerklasse der Prozesse verwendet. Diese Bits sind im Dateimodus enthalten, wie unter `sys/stat.h` beschrieben. Der Gebrauch der Schutzbits für eine Datei in Zugriffsentscheidungen wird detailliert unter Dateizugriffsrechte beschrieben.

Shell
shell

Ein Systemprogramm in UNIX, das die Kommunikation zwischen dem Benutzer und dem System übernimmt. Die Shell ist ein Kommando-Interpreter. Sie übersetzt die eingegebenen Kommandos in eine Sprache, die vom System verarbeitet werden kann. Eine Shell wird für jeden Benutzer gestartet, sobald er sich am System anmeldet.

Signal
signal

Ein Mechanismus, durch den ein Prozess von einem im System auftretenden Ereignis benachrichtigt oder beeinflusst werden kann. Beispiele für solche Ereignisse schließen Hardware-Ausnahmen und besondere Aktionen von Prozessen ein. Der Begriff Signal wird auch für die Ereignisse selbst verwendet.

Signalmaske
signal mask

Für einen Prozess definierte Signale, die aktuell - vor der Zustellung an diesen Prozess - blockiert werden. Die Signalmaske eines Prozesses wird von dessen Vaterprozess initialisiert. `sigaction()`, `sigfprocmask()` und `sigsuspend()` steuern die Manipulation dieser Signalmaske.

Sitzung
session

Eine Gruppierung von Prozessen für die Auftragssteuerung. Jede Prozessgruppe ist Mitglied einer Sitzung. Für einen Prozess wird angenommen, dass er ein Mitglied derjenigen Sitzung ist, in der seine Prozessgruppe Mitglied ist. Ein neu erzeugter Prozess gehört der Sitzung seines Erzeugers an. Ein Prozess kann die Mitgliedschaft in einer Sitzung ändern (siehe `setsid()`). Implementierungen, die `setpgid()` unterstützen, können mehrere Prozessgruppen in derselben Sitzung haben.

Sitzung, Lebensdauer
session lifetime
: Der Zeitraum zwischen der Erzeugung einer Sitzung und dem Ende der Lebensdauer aller Prozessgruppen, die Mitglieder dieser Sitzung bleiben.

Sitzungsleiter
session leader
: Ein Prozess, der eine Sitzung erzeugt hat (siehe `setsid()`).

Sohnprozeß
child process
: Siehe `Prozess`.

Sonderrechte
appropriate privileges
: Spezielle Rechte, die einige der in diesem Handbuch definierten Funktionen und Funktions-Optionen zu ihrem Aufruf verlangen. Dieser Begriff ersetzt gemäß POSIX-Standard den Begriff der Systemverwalter-Rechte.

Sonderzeichen
special character
: Zeichen, denen bei der Ein-/Ausgabe bestimmte Sonderfunktionen zugeordnet sind (siehe Abschnitt “Allgemeine Terminalschnittstelle” auf Seite 96).

Sortierreihenfolge
collating sequence
: Die relative Reihenfolge von Sortierelementen, wie sie beim Setzen der Kategorie `LC_COLLATE` in der aktuellen Lokalität festgelegt wird.
Die Reihenfolge von Zeichen, wie sie für die Kategorie `LC_COLLATE` in der aktuellen Lokalität festgelegt ist, definiert die relative Reihenfolge aller Sortierelemente, so dass jedes Element eine eindeutige Position in der Reihenfolge belegt. Dies ist die Reihenfolge, die in Bereichen von Zeichen und Sortierelementen in regulären Ausdrücken und Mustervergleichen benutzt wird. Außerdem werden bei der Definition der Sortierpriorität von Zeichen und Sortierelementen die Sortierelemente dazu benutzt, ihre jeweilige Position innerhalb der Sortierfolge zu repräsentieren.
Eine mehrstufige Sortierung wird ausgeführt, wenn Sortierelementen mehr als eine Sortierpriorität zugewiesen wird. Die Obergrenze ist durch `{COLL_WEIGHTS_MAX}` definiert (siehe Include-Datei `limits.h`).
Auf jeder Stufe kann den Elementen dieselbe Priorität zugewiesen werden (für dieselbe Priorität auf der Primärstufe, die Äquivalenzklasse genannt wird, siehe auch `Äquivalenzklasse`) oder sie kann in der Reihenfolge weggelassen wer-

den. Zeichenketten, die in der ersten Prioritätsstufe dieselbe Sortierposition haben (Primärreihenfolge), werden mit der nächsten Prioritätsstufe verglichen (Sekundärreihenfolge) usw.

Spaltenposition
column position

Die Entfernung eines Zeichens vom Anfang der Zeile. Es wird angenommen, dass jedes Zeichen in einem Zeichensatz eine interne Standard-Spaltenbreite hat, die abhängig vom Ausgabegerät ist. Jedes druckbare Zeichen im portablen Zeichensatz hat eine Spaltenbreite von eins. Wenn die XPG4-Kommandos so verwendet werden, wie in dieser Handbuchreihe beschrieben, erwarten sie, dass alle Zeichen eine interne Standard-Spaltenbreite haben. Die Spaltenbreite muss nicht notwendigerweise mit der internen Darstellung der Zeichen (Anzahl der Bits oder Bytes) verbunden sein.
Die Spaltenposition eines Zeichens in einer Zeile wird definiert als Eins, zuzüglich der Summe der Spaltenbreite der vorhergehenden Zeichen in der Zeile.

Speicherabzug
core dump

Eine Kopie des Speicherbereiches, den ein bestimmter Prozess belegt. Wenn dieser Prozess abnormal beendet wurde, wird der Speicherabzug in die Datei `core` geschrieben.

Speicherbereich
memory area

Ein abgegrenzter Raum des Arbeitsspeichers. Er kann bestimmten Programmen zugewiesen und entsprechend den Programmerfordernissen - beliebig unterteilt werden.

Standard-Ausgabe
standard output

Ein Datenstrom, der mit einem primären Ausgabegerät verbunden ist.

Standard-Eingabe
standard input

Ein Datenstrom, der mit einem primären Eingabegerät verbunden ist.

Standard-Fehlerausgabe
standard error

Ein Ausgabestrom, der für Diagnosemeldungen verwendet wird.

Standard-Kommandos
standard utilities

Die Kommandos, die im Handbuch „POSIX Kommandos (BS2000/OSD)“ beschrieben sind.

steuernder Prozess
controlling process

Der Sitzungsleiter, der die Verbindung zum steuernden Terminal hergestellt hat. Wenn das Terminal aufhört, ein steuerndes Terminal für diese Sitzung zu sein, dann hört auch der Sitzungsleiter auf, der steuernde Prozess zu sein.

steuerndes Terminal
controlling terminal

Ein Terminal, das einer Sitzung zugewiesen ist. Jede Sitzung kann höchstens ein zugewiesenes steuerndes Terminal besitzen. Ein steuerndes Terminal ist genau einer Sitzung zugewiesen. Bestimmte Eingabesequenzen vom steuernden Terminal bewirken, dass Signale an alle Prozesse gesendet werden, die sich in der Prozessgruppe befinden, die diesem steuernden Terminal zugewiesen ist.

Steuerzeichen
control character

Ein nicht darstellbares Zeichen, das die Aufzeichnung, Verarbeitung, Übertragung oder Interpretation von Text beeinflusst.

stromorientierte Ein-/Ausgabe
stream oriented I/O

Stromorientierte Ein-/Ausgabe bedeutet, dass sich der Lese-/Schreibzeiger auf jedes einzelne Byte in der Datei positionieren lässt. Stromorientierte Ein-/Ausgabe ist der herkömmliche Verarbeitungsmodus und standardmäßig eingestellt, d.h. ohne besondere Zusatzangaben bei den Eröffungsfunktionen. Textdateien können ausschließlich in diesem Ein-/Ausgabe-Modus verarbeitet werden. Im Gegensatz zur satzorientierten Ein-/Ausgabe werden bei der Ausgabe in Dateien mit stromorientierter Ein-/Ausgabe die Daten zunächst in einem Puffer zwischengespeichert und erst später in die externe Datei geschrieben (siehe `Pufferung`).

Suchmuster
pattern

Eine Zeichenfolge, die entweder mit der Notation für reguläre Ausdrücke oder für Pfadnamen-Erweiterung in dem Sinne verwendet wird, dass verschiedene Zeichenketten bzw. Pfadnamen ausgewählt werden. Die Syntax von zwei Such-

mustern ist gleich, aber nicht identisch. Diese Handbuchreihe zeigt immer den Suchmustertyp an, auf den sich im unmittelbaren Kontext zur Verwendung des Ausdrucks bezogen wird.

System
system

Der Begriff System wird in diesem Handbuch verwendet, um eine Implementierung der Systemschnittstellen zu bezeichnen.

Systemaufruf
system call

Anforderung eines Dienstes, der vom Betriebssystem-Kernel ausgeführt wird, aus einem Programm.

systemglobale Benutzerverwaltung
user administration

BS2000:
Alle Rechte, die mit dem Kommando /SET-PRIVILEGE vergeben werden können, sowie das Recht des Sicherheitsbeauftragten und der Systemkennung TSOS.

Systemkern
kernel

Der Code des POSIX-/UNIX-Betriebssystems.

Systempriorität
system scheduling priority

Eine Zahl, mit der das System die Systempriorität ändern kann. Die Erhöhung des Wertes bewirkt eine höhere Priorität beim Ablauf des Prozesses. Die Verminderung des Wertes bewirkt eine niedrigere Priorität.

Systemprozeß
system process

Ein herstellerabhängiges Objekt, das anders als ein Prozess, der eine Anwendung ausführt, vom System definiert ist. Ein Systemprozeß besitzt eine Prozeßnummer.

Terminal
terminal

Eine Gerätedatei für ein zeichenorientiertes Gerät, die den Vorgaben der allgemeinen Terminalschnittstelle genügt (siehe Abschnitt “Allgemeine Terminalschnittstelle” auf Seite 96).

Textdatei
text file

BS2000:
Textdateien gibt es nur für Strom-Ein-/Ausgabe. Folgende Dateiarten werden als Textdateien behandelt:
– katalogisierte SAM-Dateien (kein Binärmodus beim Öffnen)
– katalogisierte ISAM-Dateien
– Systemdateien (SYSDTA, SYSOUT, SYSLST, SYSTERM)

Eine Textdatei ist eine geordnete Folge von Bytes, die zu Zeilen (bzw. Sätzen) zusammengefasst ist. Im Unterschied zu Binärdateien werden die Steuerzeichen für Zwischenraum je nach Art der Textdatei in ihre entsprechende Wirkung umgesetzt (siehe `Zwischenraum`). Daten, die aus einer Textdatei eingelesen werden, entsprechen daher nicht genau den Daten, die ursprünglich in die Datei geschrieben wurden. Für einen geschriebenen Tabulator (`\t`) wird eine entsprechende Anzahl von Leerzeichen gelesen. Zusätzlich gibt es bei Textdateien noch Folgendes zu beachten:
– Es können Neue-Zeile-Zeichen eingelesen werden, die ursprünglich nicht in die Datei geschrieben wurden (siehe `fflush()`, `fseek()`, `fsetpos()`, `lseek()`, `rewind()`).
– Ausgabe auf SYSOUT und SYSTERM (zum Schreiben)
  Jede Zeile wird mit einem Leerzeichen als Drucksteuerzeichen begonnen. Dies bewirkt einen Zeilenvorschub.
– Ausgabe auf SYSLST
  Nur wenn keines der Steuerzeichen `\f`, `\v`, `\r` oder `\b` in einer Zeile angegeben wird, beginnt die Zeile mit einem Leerzeichen als Drucksteuerzeichen.

übergeordnetes Dateiverzeichnis
parent directory

Das Dateiverzeichnis, das einen Dateiverzeichniseintrag für die betreffende Datei enthält. Dieses Konzept findet für . und .. keine Anwendung.

Umwandlung in Großbuchstaben
upshifting

Die Umwandlung von Kleinbuchstaben in ihre entsprechenden Großbuchstaben.

Umwandlung in Kleinbuchstaben
downshifting

Die Umwandlung von Großbuchstaben in ihre entsprechenden Kleinbuchstaben.

UNIX-System

Ein im Dialogbetrieb arbeitendes Betriebssystem, das 1969 von Bell Laboratories entwickelt wurde. Da nur ein zentraler Systemkern von UNIX hardwareabhängig ist, wird UNIX auf vielen unterschiedlichen Systemen verschiedener Computerhersteller eingesetzt. Die Siemens Nixdorf Version von UNIX heißt SINIX bzw. Reliant UNIX. UNIX-Anwendungen sind in hohem Maße portierbar.

Unterbrechung

interrupt

Eine Unterbrechung der normalen Bearbeitung eines Programms. Unterbrechungen werden durch Signale verursacht, die durch einen Hardwarezustand oder ein Peripheriegerät ausgelöst werden, um einen besonderen Zustand anzuzeigen. Wird die Unterbrechung von der Hardware erkannt, wird eine Interrupt-Service-Routine ausgeführt. Ein Unterbrechungszeichen ist üblicherweise ein ASCII-Zeichen, das eine Unterbrechung auslöst, wenn es über die Tastatur eingegeben wird.

unterbrochener Auftrag

suspended job

Ein Hintergrundauftrag, der ein Signal `SIGSTOP`, `SIGTSTP`, `SIGTTIN` oder `SIGTTOU` erhalten hat.

Unterverzeichnis

child directory

Ein Dateiverzeichnis, auf das ein Dateiverzeichnis aus der nächsthöheren Ebene des Dateisystems verweist.

Variable

variable

Objekt, dessen Wert sich während der Ausführung eines Programms ändern kann.

Vaterprozeß

parent process

Siehe `Prozess`.

Vaterprozeßnummer

parent process ID

Ein neuer Prozess wird von einem aktiven Prozess erzeugt. Die Vaterprozeßnummer eines Prozesses ist die Prozeßnummer seines Erzeugers, solange dieser lebt. Endet diese Prozeßlebensdauer, ist die Vaterprozeßnummer die Prozeßnummer des `init`–Prozesses.

Vergleichsfolge
collation order
Die logische Reihenfolge von Zeichenketten und Langzeichenketten nach vorher festgelegten Präzedenzregeln. Diese Präzedenzregeln bestimmen die Vergleichsreihenfolge zwischen Vergleichseinheiten und solchen zusätzlichen Regeln, die dazu benutzt werden, Zeichenketten, die zusammengesetzte Vergleichseinheiten enthalten, in die richtige Reihenfolge zu bringen.

verwaiste Prozessgruppe
orphaned process group
Eine Prozessgruppe, in der der Vaterprozeß jedes Mitglieds selbst Mitglied der Gruppe oder nicht Mitglied der Sitzung dieser Gruppe ist.

Verweis
link
Siehe Dateiverzeichniseintrag.

Verweiszähler
link count
Der Verweiszähler einer Datei ist die Anzahl der Dateiverzeichniseinträge, die sich auf diese Datei beziehen.

Vordergrund
foreground
Normale Methode der Kommandoausführung in einer Shell. Wenn ein Kommando im Vordergrund ausgeführt wird, wartet die Shell auf die Beendigung dieses Kommandos, bevor der Benutzer zu einer weiteren Eingabe aufgefordert wird.

Vordergrundprozeß
foreground process
Ein Prozess, der Mitglied einer Vordergrund-Prozessgruppe ist.

Vordergrund-Prozessgruppe
foreground process group
Eine Prozessgruppe, deren Mitglieder bestimmte Privilegien haben, die Hintergrundprozessen verweigert werden, wenn sie auf ihr steuerndes Terminal zugreifen. Jede Sitzung, die eine Verbindung zu einem steuernden Terminal aufgebaut hat, besitzt exakt eine Prozessgruppe dieser Sitzung als Vordergrund-Prozessgruppe dieses steuernden Terminals.

Vordergrund-Prozeßgruppennummer
foreground process group ID
Die Prozeßgruppenummer einer Vordergrund-Prozessgruppe.

Voreinstellung
default

Art und Weise, in der ein Programm ausgeführt wird, wenn keine weiteren Angaben gemacht werden.

Wagenrücklauf-Zeichen
carriage-return character

Ein Zeichen, das im Ausgabestrom anzeigt, dass der Druck am Anfang derselben physikalischen Zeile beginnen soll, in der sich auch das Wagenrücklauf-Zeichen befindet. Das Wagenrücklauf-Zeichen wird in C mit `\r` angegeben. Es ist undefiniert, ob das Zeichen die exakte Folge ist, die durch das System an ein Ausgabegerät übertragen wird, um das Warnzeichen abzusetzen.

Warnsignal
alert

Ein akustisches oder visuelles Signal des Benutzerterminals, das einen Fehler oder ein anderes Ereignis anzeigt. Wenn die Standard-Ausgabe auf das Terminal gelenkt wird, ist die Warnmethode nicht festgelegt. Wenn die Standard-Ausgabe nicht auf das Terminal gelenkt wird, wird die Warnung durch einen Warnbuchstaben auf die Standardausgabe geschrieben.

Warnzeichen
alert character

Ein Zeichen, das im Ausgabestrom bewirkt, dass das Terminal den Benutzer durch ein visuelles oder akustisches Signal warnt. Das Warnzeichen wird in C mit `\a` angegeben. Es ist undefiniert, ob das Zeichen die exakte Folge ist, die durch das System an ein Ausgabegerät übertragen wird, um das Warnzeichen abzusetzen.

Zeichen
character

Eine Folge von einem oder mehreren Bytes, die ein einzelnes grafisches Symbol oder ein Steuerzeichen repräsentiert. Dies gilt auch für Multibyte-Zeichen und Einzelbyte-Zeichen, wobei Einzelbyte-Zeichen ein spezieller Fall von Multibyte-Zeichen sind.

Zeicheneinheit
collating element

Die kleinste Einheit zur Bestimmung der Reihenfolge von Zeichen- oder Langzeichenketten (siehe `Sortierreihenfolge`). Eine Zeicheneinheit besteht entweder aus einem Einzelzeichen oder aus zwei und mehr Zeichen, die eine Einheit bilden. Der Wert der Kategorie `LC_COLLATE` in der aktuellen Lokalität bestimmt die aktuelle Menge der Zeicheneinheiten.

Zeichenklasse
character class

Eine benannte Zeichenmenge, die sich ein Attribut teilt, das auf den Namen der Klasse weist. Die Klassen und Zeichen, die in dieser Menge enthalten sind, sind vom Wert der Kategorie `LC_CTYPE` in der aktuellen Lokalität abhängig.

Zeichenkette
character string

Eine zusammenhängende Folge von Zeichen, die als letztes Element das Nullbyte enthält.

zeichenorientierte Gerätedatei
character special file

Eine Gerätdatei für zeichenorientierte Ein-/Ausgabegeräte. Ein Beispiel für eine solche Datei ist die Gerätedatei für ein Terminal.

Zeichensatz
character set

In der internationalen Umgebung für C werden die Zeichen gemäß den Regeln des 7-Bit US-ASCII-Zeichensatzes codiert. Dabei kann für jedes Zeichen des Zeichensatzes angegeben werden, welches Symbol es repräsentiert, ob eine Umwandlung in den entsprechenden Groß- oder Kleinbuchstaben erfolgen soll, welcher Zeichenklasse das Zeichen angehört und welche Position es innerhalb der Sortierreihenfolge einnimmt. In internationalisierten Programmen sind hier beliebige landessprachliche Zeichensätze denkbar (siehe Abschnitt “Zeichensätze” auf Seite 47).

Zeichensatz für portable Dateinamen
portable filename character set

Damit ein Dateiname portabel ist für alle Implementierungen, die konform zum ISO POSIX-1 Standard sind, darf er nur aus folgenden Zeichen bestehen:
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 . _ -
Die letzten drei Zeichen sind der Punkt, der Unterstrich und der Bindestrich. Der Bindestrich darf nicht als erstes Zeichen des portablen Dateinamens verwendet werden. Groß- und Kleinbuchstaben werden von allen konformen Implementierungen unterschieden.
Im Falle eines portablen Pfadnamens ist auch ein Schrägstrich zugelassen.

Zeilenendezeichen
newline character

Ein Zeichen, das anzeigt, das im Ausgabestrom das Drucken auf der nächsten Zeile beginnen soll. Das Zeilenendezeichen wird in C mit `\n` angegeben. Ist das Zeilenendezeichen nicht das erste Zeichen in einer Ausgabezeile, so ist das Ergebnis undefiniert. Es ist ebenfalls undefiniert, ob das Zeilenendezeichen die exakte Folge ist, die durch das System an ein Ausgabegerät übertragen wird, um den Zeilenvorschub abzusetzen.

Zeittakt
clock tick

Die (maschinenspezifische) Anzahl der Intervalle pro Sekunde wird durch {`CLK_TCK`} definiert. Sie wird verwendet, um den Wert im Typ `clock_t` auszudrücken, der von `time.h` geliefert wird.

Zombieprozeß
zombie process

Ein inaktiver Prozess, der zu einem späteren Zeitpunkt gelöscht werden wird, wenn sein Vaterprozeß eine der Funktionen `wait()` oder `waitpid()` ausführt.

Zugriffsmodus
access mode

Die Methode, wie auf Datensätze einer Datei zugegriffen wird.

zusätzliche Gruppennummer
supplementary group ID

Ein Attribut eines Prozesses, mit dem die Dateizugriffsrechte bestimmt werden. Ein Prozess besitzt, zusätzlich zur effektiven Gruppennummer, bis zu {`NGROUPS_MAX`} weitere Gruppennummern. Die zusätzlichen Gruppennummern eines Prozesses werden bei dessen Erzeugung auf die Werte der zusätzlichen Gruppennummern des Vaterprozesses gesetzt. Ob die effektive Gruppennummer eines Prozesses mit in die Liste der weiteren Gruppennummern aufgenommen wird oder nicht, ist nicht festgelegt.

Zwischenraum
white space

Eine Folge von einem oder mehreren Zeichen, die zur Zeichenklasse `space` gehören. Diese Klasse wird durch die Kategorie `LC_CTYPE` in der aktuellen Lokalität definiert. In der POSIX-Lokalität besteht ein Zwischenraum aus einem oder mehreren Leerzeichen oder horizontalen oder vertikalen Tabulatoren, Zeilenendezeichen, Wagenrücklaufzeichen, Seitenvorschubzeichen und vertikalen Tabulatoren.

Literatur

Bei Fujitsu-Siemens Computers bestellbare Handbücher

Zum Bestellen von Handbüchern wenden Sie sich bitte an die für Sie zuständige Geschäftsstelle der Fujitsu-Siemens Computers GmbH.

[1] POSIX im BS2000/OSD
Die Öffnung zur UNIX-Welt
Allgemeine Beschreibung

[2] **POSIX** (BS2000/OSD)
Grundlagen für Anwender und Systemverwalter
Benutzerhandbuch

Zielgruppe
BS2000-Systemverwalter, POSIX-Verwalter, BS2000-Benutzer,
Benutzer von UNIX-/SINIX-Workstations
Inhalt
- Einführung und Arbeiten mit POSIX
- BS2000-Softwareprodukte im Umfeld von POSIX
- POSIX installieren
- POSIX steuern und Dateisysteme verwalten
- POSIX-Benutzer verwalten
- BS2000-Kommandos für POSIX

[3] **POSIX V1.1A** (BS2000/OSD)
Kommandos

Benutzerhandbuch

Zielgruppe
Das Handbuch wendet sich an alle Benutzer der POSIX-Shell.
Inhalt
Dieses Handbuch ist ein Nachschlagewerk. Es beschreibt das Arbeiten mit der POSIX-Shell sowie die Kommandos der POSIX-Shell in alphabetischer Reihenfolge.

[4] **C** (BS2000)
C-Compiler
Benutzerhandbuch

Zielgruppe
C-Anwender im BS2000
Inhalt
– Beschreibung aller Tätigkeiten zum Erzeugen eines ablauffähigen C-Programms: Übersetzen, Binden, Laden, Testen;
– Programmierhinweise und weitergehende Informationen zu: Optimierung, Programmablaufsteuerung, Funktions- und Sprachverknüpfung, Sprachumfang des C-Compilers.

[5] **C/C++ V3.0B** (BS2000/OSD)
C/C++-Compiler
Benutzerhandbuch

Zielgruppe
C- und C++-Anwender im BS2000/OSD.
Inhalt
– Beschreibung aller Tätigkeiten zum Erzeugen von ablauffähigen C- und C++-Programmen: Übersetzen, Binden, Laden, Testen;
– Programmierhinweise und weitergehende Informationen zu: Optimierung, Programmablaufsteuerung, Funktions- und Sprachverknüpfung, C- und C++-Sprachunterstützung des Compilers.

[6] **C/C++ V3.0B** (BS2000/OSD)
C/C++-Compiler
Benutzerhandbuch

Zielgruppe
C- und C++-Anwender im BS2000/OSD.
Inhalt
– Beschreibung aller Tätigkeiten zum Erzeugen von ablauffähigen C- und C++-Programmen: Übersetzen, Binden, Laden, Testen;
– Programmierhinweise und weitergehende Informationen zu: Optimierung, Programmablaufsteuerung, Funktions- und Sprachverknüpfung, C- und C++-Sprachunterstützung des Compilers.

[7] **C/C++ V3.0B** (BS2000/OSD)
POSIX-Kommandos des C/C++-Compilers
Benutzerhandbuch

Zielgruppe
C- und C++-Anwender im BS2000/OSD.
Inhalt
– Einführung in die C/C++-Programmentwicklung in POSIX-Shell-Umgebung.
– Übersetzen und Binden von C- und C++-Programmen mit den POSIX-Kommandos cc, c89 und CC.
– Steuern des globalen C/C++-Listengenerators mit dem POSIX-Kommando cclistgen.

[8] **C/C++ V3.0B** (BS2000/OSD)
POSIX-Kommandos des C/C++-Compilers
Benutzerhandbuch

Zielgruppe
C- und C++-Anwender im BS2000/OSD.
Inhalt
– Einführung in die C/C++-Programmentwicklung in POSIX-Shell-Umgebung.
– Übersetzen und Binden von C- und C++-Programmen mit den POSIX-Kommandos cc, c89 und CC.
– Steuern des globalen C/C++-Listengenerators mit dem POSIX-Kommando cclistgen.

[9] **C-Bibliotheksfunktionen** (BS2000/OSD)
Zielgruppe
C- und C++-Anwender im BS2000/OSD
Inhalt
– Beschreibung aller C-Funktionen und Makros, die im BS2000-Betriebssystem ohne POSIX genutzt werden können.
– Grundlegende Informationen, Programmierhinweise und Beispiele zu: BS2000-Dateiverarbeitung, STXIT- und Contingency-Routinen, Lokalität.

[10] **CRTE (BS2000/OSD)**
Common RunTime Environment
Benutzerhandbuch
Zielgruppe
Programmierer und Systemverwalter im BS2000/OSD
Inhalt
Beschreibung der gemeinsamen Laufzeitumgebung für COBOL85-, COBOL2000-, C-, und C++-Objekte sowie für "Fremdsprachenmix":
– Komponenten des CRTE
– Programmkommunikationsschnittstlle ILCS
– -Bindebeispiele

[11] **DCE** (BS2000)
POSIX-Programmschnittstelle
Benutzerhandbuch

Zielgruppe
Programmierer von DCE-Anwendungen
Inhalt
Beschreibung der C-Bibliotheksfunktionen mit POSIX-Funktionalität, die zum Programmieren von DCE-Anwendungen in BS2000/OSD V1.0 benötigt werden.

[12] **SDF-P** (BS2000/OSD)
Programmieren in der Kommandosprache
Benutzerhandbuch

Zielgruppe
BS2000-Anwender und Systembetreuung.
Inhalt
SDF-P ist eine strukturierte Prozedursprache im BS2000. Nach einführenden Kapiteln zum Prozedur- und Variablenkonzept werden Kommandos, Funktionen und Makros ausführlich beschrieben.

[13] **BS2000/OSD**
Makroaufrufe an den Ablaufteil
Benutzerhandbuch

Zielgruppe
Das Handbuch wendet sich an alle BS2000/OSD-Assembler-Programmierer.
Inhalt
Das Handbuch enthält eine Zusammenstellung der Makroaufrufe an den Ablaufteil, die ausführliche Beschreibung jedes Makroaufrufs mit Hinweisen und Beispielen sowie einen ausführlichen allgemeinen Lernteil.

[14] **BS2000/OSD-BC**
Einführung in das DVS
Benutzerhandbuch

Zielgruppe
Das Handbuch wendet sich an den nichtprivilegierten Anwender und an die Systembetreuung.
Inhalt
Es beschreibt die Dateiverwaltung und -verarbeitung im BS2000.
Themenschwerpunkte:
– Datenträger und Dateien
– Datei- und Katalogverwaltung
– Datei- und Datenschutz
– OPEN-, CLOSE-, EOV-Verarbeitung
– DVS-Zugriffsmethoden (SAM, ISAM,...)
Wesentliche Neuheiten in der OSD-BC V3.0 sind das SMS-Konzept und der XCS-Verbund.
Einführung in das DVS
Benutzerhandbuch

Zielgruppe
Das Handbuch wendet sich sowohl an den nichtprivilegierten Anwender als auch an die Systembetreuung.
Inhalt
Dateiverarbeitung im BS2000: Dateien, Datei- und Katalogverwaltung, Dateischutz - Dateien und Datenträger - Datei- und Datenschutz -OPEN-, CLOSE-, EOV-Verarbeitung - DVS-Zugriffsmethoden (SAM, ISAM ...).

[15] **JV** (BS2000/OSD)
Jobvariablen
Benutzerhandbuch

Zielgruppe
Das Handbuch wendet sich sowohl an den nichtprivilegierten Anwender als auch an die Systembetreuung.
Inhalt
Es beschreibt die Verwaltung und die verschiedenen Einsatzmöglichkeiten von Jobvariablen. Die Kommandobeschreibungen sind getrennt nach den Funktionsbereichen der JVs aufgeführt. Die Makroaufrufe sind in einem eigenen Kapitel beschrieben.

Sonstige Literatur

X/Open CAE Specification
System Interfaces and Headers, Issue 4 , Version 2

ISBN: 1-85912-037-7
X/Open Document Number: C435

X/Open CAE Specification
System Interface Definitions, Issue 4

ISBN: 1-872630-46-4
X/Open Document Number: C204

X/Open CAE Specification
Commands and Utilities, Issue 4

ISBN: 1-872630-48-0
X/Open Document Number: C203

Stichwörter (Band 1 und Band 2)

B

C

E

F

Inhalt (Band 1 und Band 2)

C-Bibliotheksfunktionen V2.5A (BS2000/OSD) **für POSIX-Anwendungen**

Referenzhandbuch

Zielgruppe
C- und C++-Programmierer

Inhalt
Das Handbuch dokumentiert die XPG4-konforme C-Programmierschnittstelle, die vom POSIX-Subsystem im BS2000 unterstützt wird. Mit dieser Programmierschnittstelle kann sowohl auf das POSIX-Dateisystem als auch auf BS2000-Dateien zugegriffen werden. Zusätzlich enthält die Programmierschnittstelle Erweiterungen, die die Kompatibilität mit der bisherigen C-Bibliothek gewährleisten.

Ausgabe: August 2001

Datei: c_plib.pdf

Fujitsu Siemens Computers GmbH
Handbuchredaktion
81730 München

Kritik
Anregungen
Korrekturen

Fax: 0 700 / 372 00000

e-mail: manuals@fujitsu-siemens.com
http://manuals.fujitsu-siemens.com

Absender

Kommentar zu C-Bibliotheksfunktionen V2.5A
für POSIX-Anwendungen

Information on this document

On April 1, 2009, Fujitsu became the sole owner of Fujitsu Siemens Computers. This new subsidiary of Fujitsu has been renamed Fujitsu Technology Solutions.

This document from the document archive refers to a product version which was released a considerable time ago or which is no longer marketed.

Please note that all company references and copyrights in this document have been legally transferred to Fujitsu Technology Solutions.

Contact and support addresses will now be offered by Fujitsu Technology Solutions and have the format *...@ts.fujitsu.com*.

The Internet pages of Fujitsu Technology Solutions are available at *http://ts.fujitsu.com/*...
and the user documentation at *http://manuals.ts.fujitsu.com*.

Hinweise zum vorliegenden Dokument

Zum 1. April 2009 ist Fujitsu Siemens Computers in den alleinigen Besitz von Fujitsu übergegangen. Diese neue Tochtergesellschaft von Fujitsu trägt seitdem den Namen Fujitsu Technology Solutions.

Das vorliegende Dokument aus dem Dokumentenarchiv bezieht sich auf eine bereits vor längerer Zeit freigegebene oder nicht mehr im Vertrieb befindliche Produktversion.

Bitte beachten Sie, dass alle Firmenbezüge und Copyrights im vorliegenden Dokument rechtlich auf Fujitsu Technology Solutions übergegangen sind.

Kontakt- und Supportadressen werden nun von Fujitsu Technology Solutions angeboten und haben die Form *...@ts.fujitsu.com*.

Die Internetseiten von Fujitsu Technology Solutions finden Sie unter *http://de.ts.fujitsu.com/*..., und unter *http://manuals.ts.fujitsu.com* finden Sie die Benutzerdokumentation.