robotron

Programmtechnische Beschreibung
für Bürocomputer A 5120.16

# SYSTEMHANDBUCH MUTOS 8000

Teil I: Abschnitt 2 · Systemrufe
Abschnitt 3 · Bibliotheksfunktionen

VEB Robotron-Buchungsmaschinenwerk
Karl-Marx-Stadt
Stand: 12/85

robotron

Programmtechnische Beschreibung

fuer Buerocomputer A 5120.16

# S Y S T E M H A N D B U C H

# M U T O S 8 0 0 0 .

Teil I:

Abschnitt 2 - Systemrufe

Abschnitt 3 - Bibliotheksfunktionen



VEB Robotron-Buchungsmaschinenwerk
Karl-Marx-Stadt
Stand: 12/85

Die vorliegende Dokumentation entspricht dem Stand 12/85.

Nachdruck, jegliche Vervielfaeltigung oder Auszuege daraus sind unzulaessig.

Im Interesse einer staendigen Weiterentwicklung werden alle Leser gebeten, ihre Vorschlaege bzw. Hinweise dem

VEB Robotron-Buchungsmaschinenwerk
9010 Karl-Marx-Stadt
Annabergerstr. 93

mitzuteilen.

Fuer das Betriebssystem MUTOS 8000 wurden folgende Dokumentationen erarbeitet:

- Anwendungsbeschreibung MUTOS 8000 Teil I,
  - Abschnitt 1 Kommandos
  - Abschnitt 2, 3 Systemrufe, Bibliotheksfunktionen
  - Abschnitt 4, 5, 7, 8 Special Files, Fileformate, Makropakete, Systemunterstuetzung

- Systemhandbuch MUTOS 8000 Teil II,
  - Abschnitt 1 Kommandointerpreter Shell
  - Abschnitt 2 Editor
  - Abschnitt 3 Testhilfsprogramm adb

- Systemhandbuch MUTOS 8000 Teil III,
  - Abschnitt 1 Textverarbeitung nroff

- Sprachbeschreibungen
  - Assemblersprachbeschreibungen U8000
  - Sprachbeschreibung C-Sprache

Das Sachwortverzeichnis ist als Anlage in der Anwendungsbeschreibung enthalten.

Zur Programmentwicklung und -Testung fuer den Prozessor U8000 werden zusaetzlich folgende Dokumentationen angeboten:

- Monitorsystem MON8000
- CROSS-Software U8000 zum Betriebssystem UDOS
- Assembler U8000ASM
- Binder ZLINK
- Absolutbinder IMAGER
- Preprozessor INCLUDE

Inhaltsverzeichnis

**Teil 2 - Systemrufe**

## NAME

intro,errno - Einfuehrung in die Systemrufe und Fehlernummern

## UEBERSICHT

**@include <errno.h>**

## BESCHREIBUNG

Dieser Teil des Systemhandbuches MUTOS 8000 beschreibt die Systemrufe des MUTOS-Kerns.
Die meisten Rufe liefern einen Rueckkehrwert. Eine Fehlerbedingung wird durch einen anderweitig nicht moeglichen Rueckkehrwert angezeigt. Dies ist fast immer -1. Die Fehlernummer wird auch in der externen Variablen errno bereitgestellt. Errno wird bei erfolgreichen Rufen nicht geloescht, deshalb sollte nur getestet werden, nachdem ein Fehler aufgetreten ist.

Es gibt eine Tabelle der Fehlermeldungen und eine Routine fuer das Drucken der Mitteilung (Siehe perror(3)). Moegliche Fehlernummern werden nicht in jeder Rufbeschreibung im Abschnitt 2 aufgezaehlt, obwohl mehrere Fehler fuer die meisten der Rufe moeglich sind. Nachfolgend sind eine Liste der Fehlernummern, ihre Namen, wie sie in <errno.h> definiert sind und die Mitteilungen aufgefuehrt, die bei Nutzung von perror verfuegbar sind.

0 Error 0
Nicht benutzt.

1 EPERM Not owner
Dieser Fehler ist typisch bei einem Versuch, ein File in irgendeiner Weise zu modifizieren, die ausser durch ihren Besitzer oder den Superuser verboten ist.

2 ENOENT No such file or directory
Dieser Fehler tritt auf, wenn ein Filename angegeben wird, der nicht existiert, oder wenn eines der Directories in einem Pfadnamen nicht existiert.

3 ESRCH No such process
Der Prozess, dessen Nummer in signal oder ptrace angegeben wird, existiert nicht oder ist bereits beendet.

24 EMFILE Too many open files
Die uebliche Konfigurationsgrenze ist 20 offene Files pro Prozess.

25 ENOTTY Not a typewriter
Das File, das in stty oder gtty angegeben wurde, ist kein Bediengeraet oder eines der anderen Geraete, auf die diese Rufe angewendet werden.

26 ETXTBSY Text file busy
Es wird versucht, ein reines Prozedurprogramm auszufuehren, das augenblicklich fuer Schreiben (oder Lesen!) offen ist. Auch ein Versuch, ein reines Prozedurprogramm, das ausgefuehrt wird, fuer Schreiben zu oeffnen, fuehrt zu diesem Fehler.

27 EFBIG File too large
Die Groesse eines Files uebersteigt das Maximum.

28 ENOSPC No space left on device
Waehrend eines write - Rufes auf ein gewoehnliches File gibt es keinen freien Bereich auf dem Geraet.

29 ESPIPE Illegal seek
Ein lseek Ruf wurde auf eine Pipe gegeben. Dieser Fehler wird auch fuer andere Geraete gegeben, die keinen seek-Ruf erlauben.

30 EROFS Read-only file system
Es wird versucht, ein File oder ein Directory auf einem Geraet zu modifizieren, das mit dem Attribut "nur Lesen" eingegliedert wurde.

31 EMLINK Too many links
Es wird versucht, mehr als 32767 Links zu einem File zu erzeugen.

32 EPIPE Broken pipe
Es soll auf eine Pipe geschrieben werden, fuer die es keinen Prozess gibt, die Daten zu lesen. Diese Bedingung erzeugt normalerweise ein Signal. Der Fehler wird zurueckgegeben, wenn das Signal ignoriert wird.

33. EDOM Math argument
Das Argument einer Funktion im mathematischen Paket (3M) ist ausserhalb des Wertevorrates der Funktion.

34 ERANGE Result too large
Der Wert einer Funktion im mathematischen Paket (3M) ist innerhalb der Maschinengenauigkeit nicht darstellbar.

**SIEHE AUCH**

intro(3)

**ASSEMBLER**

as /usr/include/sys.s file ...

Das Interface der Assemblersprache wird fuer jeden Systemruf angegeben.

Die Konstanten fuer die einzelnen Systemrufe sind vom Programmierer zu definieren. Zur Parameteruebergabe werden 4 Register (vom Register 0 bis maximal Register3) benutzt. Die Rueckkehrwerte erscheinen in den Registern 4 und 5. Obwohl alle Register, evtl. mit Ausnahme der Wortregister, nach einem Systemruf unveraendert sind, sollte von dieser Tatsache kein Gebrauch gemacht werden. Ein fehlerhafter Ruf wird durch Setzen des Curry-Bits im Programm-Status-Register angezeigt. Ob ein Fehler aufgetreten ist, kann mit den bedingten Sprunganweisungen jp, jr (mit Bedingungscodes C bzw. NC) ermittelt werden.

**ACCESS(2)** **ACCESS(2)**

---

**NAME**

access - Bestimmung der Zugriffsrechte und Verfuegbarkeit

**UEBERSICHT**

**access(name,mode)**
**char *name;**

**BESCHREIBUNG**

Access prueft fuer jedes File name die Zugriffsrechte entsprechend dem Modus mode, der die Werte 4 (Lesen), 2 (Schreiben), 1 (Ausfuehren) oder eine Kombination davon annehmen kann.
Bei Angabe von Modus 0 wird getestet, ob das Directory, das zu File fuehrt, gesucht werden kann und ob das File existiert.

Eine entsprechende Fehleranzeige wird zurueckgegeben, wenn der Filename name nicht gefunden oder irgendeiner der gewuenschten Zugriffsmodi nicht erteilt werden kann. Bei nichtgewaehrtem Zugriff wird -1 zurueckgegeben und der Fehlercode in errno abgelegt (siehe intro(2)).
Bei erfolgreichem Test wird 0 zurueckgegeben.

Der Systemruf ist nur dann sinnvoll, wenn er nicht vom Superuser ausgefuehrt wird.

Es ist anzumerken, dass nur die Zugriffsbits geprueft werden. Ein Directory kann als schreibbar durch access gemeldet werden, aber der Versuch, es fuer Schreiben zu oeffnen fuehrt zu einem Fehler (obgleich Files in ihm erzeugt werden koennen); ein File kann ausfuehrbar erscheinen, aber exec fuehrt zu einem Fehler, falls es nicht im korrekten Format ist.

**SIEHE AUCH**

stat(2), intro(2), perror(3)

**ASSEMBLER**

```
(ACCESS:=33)
(name in r0)
(mode in r1)
sc #ACCESS
```



---

**NAME**

acct - Ein- und Ausschalten der Abrechnung

**UEBERSICHT**

```
acct(file)
char *file;
```

**BESCHREIBUNG**

Das System wird angewiesen fuer jeden beendeten Prozess einen Satz in ein Abrechnungsfile file zu schreiben. Dieser Ruf schaltet die Abrechnung ein, wenn das Argument der Name eines mit einem Nullbyte endenden existierenden Files ist. Die Saetze fuer jeden beendeten Prozess werden an file angehaengt. Ist das Argument 0, wird die Abrechnung ausgeschaltet.

**DIAGNOSTIK**

Bei einem Fehler wird -1 zurueckgegeben. Das File muss existieren und der Ruf darf nur vom Superuser ausgefuehrt werden.
Es ist fehlerhaft zu versuchen, die Abrechnung einzuschalten, wenn sie schon eingeschaltet ist.

**FEHLERQUELLEN**

Die Abrechnung wird fuer laufende Programme nicht erzeugt, wenn ein Crash auftritt. Nicht endende Programme werden nicht abgerechnet.

**ASSEMBLER**

```
(ACCT:=51)
(file in r0)
sc #ACCT
```

**NAME**

alarm - Eintragen des Signals nach einer angegebenen Zeit

**UEBERSICHT**

```
alarm(seconds)
unsigned seconds;
```

**BESCHREIBUNG**

Alarm sendet das Signal SIGALRM (siehe signal(2)) nach einer Anzahl von Sekunden, die als Argument angegeben sind, an den rufenden Prozess. Wird es nicht behandelt oder ignoriert, beendet das Signal den Prozess.
Alarmanforderungen werden nicht im Stack abgelegt. Aufeinanderfolgende Rufe setzen die Alarmuhr neu. Ist das Argument 0, wird jede beliebige Alarmanforderung geloescht. Weil die Uhr eine Genauigkeit von einer Sekunde hat, kann das Signal bis zu einer Sekunde frueher auftreten. Bedingt durch die Arbeit des Shedulers kann die Wiederaufnahme der Ausfuehrung nach der Behandlung des Signals auf einen willkuerlichen Wert verzoegert werden. Die laengste angebbare Verzoegerungszeit ist 65535 Sekunden.

Der Rueckkehrwert ist der Betrag der Zeit, die zuvor in der Alarmuhr zurueckgeblieben ist.

**SIEHE AUCH**

pause(2), signal(2), sleep(3)

**ASSEMBLER**

```
(ALARM:=27)
(seconds in r0)
sc #ALARM
(der vorige Wert in r4)
```

---

**NAME**

brk, sbrk, break - Veraendern der Speicherzuteilung

**UEBERSICHT**

```
char *brk(addr)

char *sbrk(incr)
```

**BESCHREIBUNG**

Brk setzt den vom System gefuehrten Wert des niedrigsten Speicherplatzes (Break genannt), der nicht vom Programm benutzt wird, auf addr, gerundet auf das naechste Vielfache von 256 Bytes. Speicherplaetze, die nicht kleiner als addr sind und unterhalb des Stackpointers liegen, sind zwar auf dem BC A5120.16 vom Programm zugreifbar, werden aber beim Auslagern des Prozesses (swapout) nicht mit ausgelagert und enthalten somit nach Wiedereinlagern unbestimmte Werte.

In der alternativen Funktion sbrk werden incr Bytes zum Datenbereich des Programmes addiert und ein Pointer auf den Start des neuen Bereiches wird zurueckgegeben. Beginnt ein Programm die Ausfuehrung durch exec, wird der Break auf den hoechsten Speicherplatz gesetzt, der durch den Programm- und Datenbereich definiert ist. Normalerweise brauchen daher nur Programme mit zunehmenden Datenbereichen break anzuwenden.

**SIEHE AUCH**

exec(2), malloc(3), end(3)

**DIAGNOSTIK**

Null wird zurueckgegeben, wenn der break gesetzt werden konnte; -1, wenn das Programm mehr Speicher als verfuegbar verlangt.

**FEHLERQUELLEN**

Das Setzen des Breaks in den Bereich von 0177401 bis 0177777 ist identisch dem Setzen auf Null.

**ASSEMBLER**

```
(BREAK:=17)
(addr in r0)

sc #BREAK
```

Break leistet die Funktion von brk. Der Name ruehrt aus einer Zeit,als 'break' noch kein Schluesselwort in C war.

---

**NAME**

chdir,chroot - Aenderung des Standarddirectory

**UEBERSICHT**

```
chdir(dirname)
char *dirname;

chroot(dirname)
char *dirname;
```

**BESCHREIBUNG**

Dirname ist der Zeiger auf den Pfadnamen eines Directory, welcher durch ein Nullbyte beendet wird. Chdir veranlasst, dass dieses Directory das aktuelle Arbeitsdirectory wird. Der Startpunkt fuer den Pfadnamen beginnt im aktuellen Direktory oder mit '/'.

Chroot setzt das Rootdirectory. Der Startpunkt fuer den Pfadnamen beginnt mit '/'. Der Ruf ist auf den Superuser beschraenkt.

**SIEHE AUCH**

cd(1)

**DIAGNOSTIK**

Null wird zurueckgegeben, wenn das Directory veraendert ist; -1, wenn der uebergebene Name nicht der eines Directory ist, oder das Directory nicht auffindbar ist.

**ASSEMBLER**

```
(CHDIR :=12)
(dirname in r0)
   sc # CHDIR

(CHROOT :=61)
(dirname in rØ)
   sc # CHROOT
```

---

**NAME**

close - Schliessen eines Files

**UEBERSICHT**

**close(fildes)**

**BESCHREIBUNG**

Gegeben ist ein Filedeskriptor fildes, wie er von einem open, creat, dup oder pipe(2) Ruf zurueckgegeben wird. Close schliesst das damit verbundene File. Ein Schliessen aller Files erfolgt zwar automatisch bei exit, doch fuer jeden Prozess gibt es nur eine begrenzte Anzahl offener Files. Diese Anzahl ist in usr/sys/h/param.h festgelegt (nfile = 20 ist Standard). Close ist deshalb fuer Programme notwendig, die mit vielen Files arbeiten.

Files werden bei Beendigung eines Prozesses geschlossen. Bestimmte Filedeskriptoren koennen durch exec(2) geschlossen werden.

**SIEHE AUCH**

creat(2), open(2), pipe(2), exec(2)

**DIAGNOSTIK**

Null wird zurueckgegeben, wenn ein File geschlossen ist; -1 bei einem unbekannten Filedeskriptor.

**ASSEMBLER**

```
(CLOSE:=16)
(fildes in r0)
sc #CLOSE
```



---

**NAME**

creat - Erzeugen eines neuen Files

**UEBERSICHT**

```
creat(name,mode)
char *name;
```

**BESCHREIBUNG**

CREAT erzeugt ein neues File oder bereitet ein existierendes File, das mit name bezeichnet wird und als Adresse einer mit einem Nullbyte beendeten Zeichenkette gegeben ist, fuer Wiederbeschreiben auf. Existierte das File nicht ist der Modus mode gegeben, der durch die

Modusmaske des Prozesses (siehe umask(2)) modifiziert wird. Siehe auch chmod(2) fuer die Bildung des Arguments mode.

Existierte das File, bleibt sein Modus und der Eigentuemer unveraendert, aber es wird auf die Laenge 0 gesetzt.
Das File ist auch fuer Schreiben geoeffnet und sein Filedeskriptor fildes wird zurueckgegeben.

Der gegebene Modus modes ist willkuerlich; er braucht Schreiben nicht zu erlauben. Diese Eigenschaft wird bei Programmen verwendet, die mit zeitweiligen Files mit feststehenden Namen arbeiten. Die Erzeugung erfolgt mit einem Modus, der Schreiben verbietet. Versucht dann ein zweites Beispiel des Programmes ein creat ,wird ein Fehler zurueckgegeben und das Programm weiss, dass der Name momentan nicht verwendbar ist.

**SIEHE AUCH**

write(2), close(2), chmod(2), umask(2)

**DIAGNOSTIK**

Der Wert -1 wird zurueckgegeben, wenn ein benoetigtes Directory nicht auffindbar ist, das File nicht existiert und das Directory, in dem es erzeugt werden soll, nicht schreibbar ist oder das File existiert und nicht schreibbar ist oder das File ein Directory ist oder schon zu viele offene Files vorhanden sind.

**ASSEMBLER**

```
(CREAT:=4)
(name in r0)M
(mode in r1)
sc $CREAT
(Filedeskriptor in r4)
```

**DUP(2)** **DUP(2)**

---

**NAME**

dup, dup2 - Duplizieren eines offenen Filedeskriptors

**UEBERSICHT**

```
dup(fildes)
int fildes;

dup2(fildes,fildes2)
int fildes,fildes2;
```

**BESCHREIBUNG**

Gegeben ist ein Filedeskriptor, der von einem open, pipe oder creat - Ruf zurueckgegeben wurde. Dup teilt einen anderen Filedeskriptor zu, der mit dem Original uebereinstimmt. Der neue Filedeskriptor wird zurueckgegeben.

In der zweiten Form des Rufes ist fildes ein Filedeskriptor, der sich auf ein offenes File bezieht. Fildes2 ist ein nichtnegativer Integerwert, der kleiner ist als der Maximalwert, der fuer Filedeskriptoren erlaubt ist (annaehernd 19). Dup2 veranlasst, dass fildes2 sich auf das gleiche File wie fildes bezieht. Bezieht sich fildes2 schon auf ein offenes File, wird es zuerst geschlossen.

**SIEHE AUCH**

creat(2), open(2), close(2), pipe(2)

**DIAGNOSTIK**

Der Wert -1 wird zurueckgegeben, wenn der gegebene Filedeskriptor ungueltig ist und es schon zu viele offene Files gibt.

**ASSEMBLER**

```
(DUP:= 41)
(fildes  in r0)
(fildes2 in r1)
sc #DUP
(neuer Filedescriptor in r4)
```

Der dup2 - Eintrittspunkt wird durch Anf.- ODER von 0x40 auf fildes (in r0) erreicht.

**EXEC(2)** **EXEC(2)**

---

**NAME**

execl, execv, execle, execve, execlp, execvp, exec, exece, environ - Ausfuehren eines Files

**UEBERSICHT**

```
execl(name,arg0,arg1,...,argn,0)
char *name,*arg0,*arg1,...,*argn;

execv(name,argv)
char *name,*argv[];

execle(name,arg0,arg1,...,argn,0,envp)
char *name,*arg0,*arg1,...,*argn,*envp[];

execve(name,argv,envp);
char *name,*argv[],*envp[];

extern char **environ;
```

**BESCHREIBUNG**

Exec mit allen seinen Formen ueberlagert den rufenden Prozess mit dem bezeichneten File und springt zum Eintrittspunkt des Speicherabbildes dieses Files. Es gibt keine Rueckkehr von einem erfolgreichen exec, das rufende Speicherabbild geht verloren.

Files bleiben nach exec offen, wenn nicht explizite Vereinbarungen gemacht wurden (siehe ioctl(2)). Ignorierte Signale bleiben nach diesen Rufen ignoriert. Behandelte Signale (siehe signal(2)) werden auf ihre Standardaktion zurueckgesetzt.

Jeder Nutzer hat eine reale Nutzer- und Gruppenidentifikation und eine effektive Nutzer- und Gruppenidentifikation. Die reale Identifikation identifiziert die Person, die das System nutzt, die effektive Identifikation bestimmt ihre Zugriffsprivilegien. Exec veraendert die effektive Nutzer- und Gruppenidentifikation auf die des Nutzers des auszufuehrenden Files, wenn das File den Modus 'Setzen der Nutzeridentifikation' oder 'Setzen der Gruppenidentifikation' besitzt. Die reale Nutzeridentifikation wird nicht beeinflusst.

Das Argument name ist ein Zeiger auf den Namen des auszufuehrenden Files. Die Zeiger arg[0], arg[1], ... adressieren mit einem Nullbyte abgeschlossene Zeichenketten. Vereinbarungsgemaess ist arg[0] der Name des Files.

Von C sind zwei Interfaces verfuegbar. Execl ist nuetzlich, um ein bekanntes File mit bekannten Argumenten zu rufen.
Die Argumente von execl sind Zeichenketten, die das File und die Argumente bilden. Das erste Argument ist vereinbarungsgemaess das gleiche wie der Filename (oder seine letzte Komponente). Dem letzten Argument muss ein Nullzeiger folgen, um die Argumentenliste abzuschliessen.

Die execv Version wird verwendet, wenn die Zahl der Argumente im Voraus nicht bekannt ist. Die Argumente von execv sind der Name des auszufuehrenden Files und ein Feld von Zeichenkettenzeigern auf die Argumente. Das Feld der Zeichenkettenzeiger wird durch einen Nullzeiger beendet.

Wird ein C-Programm ausgefuehrt, ist es wie folgt zu rufen:

```
main(argc,argv,envp)
int argc;
char **argv,**envp;
```

Argc ist ein Argumentzaehler und argv ein Feld von Zeichenzeigern auf die Argumente selbst. Wie dargestellt ist argc vereinbarungsgemaess mindestens 1 und das erste Element des Feldes zeigt auf eine Zeichenkette, die den Namen des Files enthaelt.

Argv ist direkt in einem anderen execv nutzbar, da argv[argc] Null ist.

Envp ist ein Zeiger auf ein Feld von Zeichenketten, die die Umgebung (environ) des Prozesses bilden. Jede Zeichenkette besteht aus einem Namen, einem "=" und einem abschliessenden Nullbyte. Das Feld der Zeiger wird durch einen Nullzeiger beendet. Der Shell sh(1) uebergibt einen Eintrittspunkt der Umgebung fuer jede globale Shell-Variable, die definiert ist, wenn das Programm gerufen wird. Siehe environ(5) fuer einige vereinbarungsgemaess benutzte Namen. Die C-Laufzeit-Startroutine gibt eine Kopie von envp in die globale Zelle environ, die von execv und execl verwendet wird, um die Umgebung an jedes Subprogramm zu geben, das vom augenblicklichen Programm ausgefuehrt wird. Die exec Routinen nutzen Routinen auf niedrigem Niveau, wie die folgenden, um eine Umgebung explizit zu uebergeben:

```
execle(file,arg0,arg1,...,argn,0,environ);
execve(file,argv,environ);
```

Execlp und execvp werden mit den gleichen Argumenten wie execl und execv aufgerufen, aber duplizieren die Aktionen der Shell beim Suchen nach einem ausfuehrbaren File in einer Liste der Directories. Die Directory-Liste wird von der Umgebung erhalten.

**FILES**

/bin/sh Shell wird aufgerufen, wenn Kommandofiles von execlp oder execvp Kommandofiles erkannt werden.

**SIEHE AUCH**

fork(2), environ(5)

**DIAGNOSTIK**

Wenn das File nicht gefunden werden kann, wenn es nicht ausfuehrbar ist, wenn es nicht mit einer gueltigen Magic-Number gestartet wird (siehe a.out(5)), wenn das Maximum des Speichers ueberschritten wird oder wenn die Argumente zuviel Platz verlangen, bildet eine Rueckkehr die Diagnose; der Rueckkehrwert ist -1. Selbst fuer den Superuser muss mindestens eines der Ausfuehrungserlaubnis-Bits fuer ein File, das ausgefuehrt werden soll, gesetzt sein.

**FEHLERQUELLEN**

Wird execvp gerufen, um ein File auszufuehren, das eine Shell-Kommandofolge enthaelt, erfolgt eine Fehlermeldung. Ist es nicht moeglich, den Shell auszufuehren, werden die Werte von argv[0] und arv[-1] vor der Rueckkehr modifiziert.

```
(EXEC:=11)
(name in r0)
(argv in r1)
sc #EXEC

(EXECE:=59)
(name in r0)
(argv in r1)
(envp in r2)
sc #EXECE
```

Das reine exec ist gegenueber exece veraltet, bleibt aber aus historischen Gruenden erhalten.

Beginnt das gerufene File die Ausfuehrung, zeigt der Stackpointer auf ein Wort, das die Anzahl der Argumente enthaelt. Genau ueber dieser Zahl ist eine Liste von Zeigern auf Argumentzeichenketten, gefolgt von einem Nullzeiger, abgelegt. Nach diesem Nullzeiger folgen Zeiger auf Umgebungszeichenketten und ein weiterer Nullzeiger. Die Zeichenketten selbst folgen. Ein Nullbyte befindet sich am oberen Ende dieses Speicherbereiches.

```
sp-> nargs
     arg0
     ...
     argn
     0
     env0
     ...
     envm
     0

arg0:     <arg0\0>
     ...
env0:     <env0\0>
     0
```

Diese Anordnung wurde gewaehlt, um mit den von geforderten Konventionen uebereinzustimmen.

**NAME**

exit - Beenden eines Prozesses

**UEBERSICHT**

```
exit(status)
int status;

_exit(status)
int status;
```

**BESCHREIBUNG**

Exit ist die normale Moeglichkeit, einen Prozess zu beenden. Exit schliesst alle Prozessfiles und teilt dies dem Parent-Prozess mit, wenn dieser einen wait Ruf ausgefuehrt hat. Die niederen 8-Bits des Status werden zum Parent-Prozess gesendet.

Dieser Ruf hat niemals eine Rueckkehr.

Die C-Funktion exit kann Aktualisierungen vor dem abschliessenden 'sys exit' veranlassen. Die Funktion exit umgeht alle Aktualisierungen.

**SIEHE AUCH**

wait(2)

**ASSEMBLER**

```
(MEXIT:=1)
(status in r0)
sc #MEXIT
```

EXIT bzw. exit sind Schluesselwoerter in PLZ/ASM, deshalb wurde der Konstantenname MEXIT fuer MUTOS-EXIT eingefuehrt.

**NAME**

fork - Erzeugen eines neuen Prozesses

**UEBERSICHT**

fork()

**BESCHREIBUNG**

Fork ist die einzige Moeglichkeit, neue Prozesse zu erzeugen. Das Speicherabbild des neuen Prozesses ist eine Kopie des Prozesses, der fork aufruft. Der einzige Unterschied zwischen ihnen ist der Rueckkehrwert des fork-Rufes. In dem alten (Parent-) Prozess ist dieser Rueckkehrwert der Prozessidentifikator des neuen (Child-) Prozesses. Der Rueckkehrwert im Child-Prozess ist dagegen Ø. Die Prozessidentifikationen liegen im Bereich von 1 bis 30000. Diese Prozessidentifikation wird durch wait(2) zurueckgegeben.

Files, die vor dem Systemruf "fork" eroeffnet waren, werden nach dem Systemruf von beiden Prozessen geteilt, d.h. beide Prozesse benutzen fuer ein File denselben Lese-/Schreib-Pointer.
Dies ist auch die Moeglichkeit, mittels dessen Pipelines eingerichtet werden.
Im Child-Prozess kann der Prozessidentifikator des Parent-Prozesses der globalen Variablen par_uid entnommen werden.

**SIEHE AUCH**

wait(2), exec(2)

**DIAGNOSTIK**

Die Rueckgabe von -1 und Fehler beim Erzeugen eines Prozesses erfolgen, wenn nicht genuegend Swap-Bereich vorhanden ist, der Nutzer nicht der Superuser ist und zu viele Prozesse hat oder die Prozesstabelle des Systems voll ist. Nur der Superuser kann den letzten Punkt der Prozesstabelle nehmen.

**ASSEMBLER**

```
(FORK:=2)
sc #FORK
```

(Rueckkehradresse im Child-Prozess, Prozessidentifikator des Parent-Prozesses in r4)

(Rueckkehradresse im Parent-Prozess, Prozessidentifikator des Child-Prozesses in r4)

Die Rueckkehradresse des Betriebssystems nach einem fork-Systemruf ist im Child-Prozess zwei Bytes, im Parent-Prozess 4 Byte groesser als die Adresse des fork-Systemrufes.
Im Parent-Prozess ist das c-Bit gesetzt, wenn kein Child-Prozess erzeugt werden konnte.

**GETPID(2)** **GETPID(2)**

---

**NAME**

getpid - Erhalten der Prozessidentifikation

**UEBERSICHT**

**getpid()**

**BESCHREIBUNG**

Getpid gibt die Prozessidentifikation des laufenden Prozesses zurueck. Am haeufigsten wird er verwendet, um einmalige Namen temporaerer Files zu erzeugen, d.h. indem die Prozessidentifikation in den Namen der temporaeren Dateien integriert wird, kann verhindert werden, dass temporaere Files mit gleichem Namen durch verschiedene Prozesse erzeugt werden.

**SIEHE AUCH**

mktemp(3)

**ASSEMBLER**

```
(GETPID:=20)
sc #GETPID
(pid in r4)
```

---

**NAME**

getuid,getgid,geteuid,getegid - Erhalten der Nutzer- und Gruppenidentifikation

**UEBERSICHT**

```
getuid()

geteuid()

getgid()

getegid()
```

**BESCHREIBUNG**

Getuid gibt die reale Nutzeridentifikation des laufenden Prozesses zurueck. Die reale Nutzeridentifikation (UID) entspricht der wirklichen Nutzernummer, die mit login festgelegt wurde.
Getenid liefert die effektivere Nutzeridentifikation, welche der momentanen Nutzernummer entspricht, die der Prozess erhalten hat.

Das Kommando getuid kann angewendet werden um festzustellen, wer ein Programm benutzt, das seinerseits den Modus "Setzen Nutzeridentifikation des Eigentuemers" verwendet. Die aktuellen Zugriffsrechte werden durch die effektive UID bestimmt.
Getgid gibt die reale Gruppenidentifikation zurueck, getegid die effektive Gruppenidentifikation.

**SIEHE AUCH**

setuid(2),chmod(2)

**ASSEMBLER**

```
(GETUID:=24)
sc #GETUID
(reale Nutzeridentifikation in r4;
effektive Gruppenidentifikation in r5)

(GETUID:=47)
sc #GETGID
(reale Gruppenidentifikation in r4;
effektive Gruppenidentifikation in r5)
```

**NAME**

ioctl - Steuerung eines Geraetes

**UEBERSICHT**

```
#include <sgtty.h>

ioctl(fildes,request,argp)
struct sgttyb *argp;
```

**BESCHREIBUNG**

Ioctl fuehrt eine Reihe von Funktionen auf zeichenorientierten Zeichen-Spezial-Files (Geraete) aus. Die Ausfuehrung ueber verschiedene Geraete sind in Abschnitt 4 beschrieben, auch, wie ioctl von diesen Geraeten angewendet wird. (siehe tty (4)).

Die folgenden zwei Rufe koennen fuer jedes offene File angewandt werden:

```
ioctl(fildes,FIOCLEX,NULL);
ioctl(fildes,FIONCLEX,NULL);
```

Der erste versucht ein automatisches Schliessen des Files waehrend einer erfolgreichen exec Operation; der zweite Ruf kehrt die Wirkung des ersten um.

**SIEHE AUCH**

stty(1), tty(4), exec(2)

**DIAGNOSTIK**

Null wird zurueckgegeben, wenn der Ruf erfolgreich war; -1, wenn der Filedeskriptor sich nicht auf eine Fileart bezieht, fuer die er bestimmt war.

**FEHLERQUELLEN**

Da ioctl in unterschiedlicher Weise auf Geraete mit unterschiedlichen Eigenschaften erweitert werden kann, sollte argp eine offen endende Deklaration haben, wie

```
union{struct sgttyb ...; ... } *argp;
```

Wichtig ist, dass die Laenge durch 'struct sgttb' festgelegt ist.

**ASSEMBLER**

```
(IOCTL:=54)
(fildes in r0)
(request in r1)
(argp in r2)
sc #IOCTT
```

**NAME**

kill - Senden eines Signals zu einem Prozess

**UEBERSICHT**

**kill(pid,sig);**

**BESCHREIBUNG**

Kill sendet das Signal sig zu dem Prozess, der durch den Prozessidentifikator pid spezifiziert ist. Siehe signal(2) fuer die Liste der Signale.

Die sendenden und empfangenden Prozesse muessen die gleiche effektive Nutzeridentifikation haben, sonst ist dieser Ruf auf den Superuser beschraenkt.

Ist der Prozessidentifikator Null, wird das Signal zu allen anderen Prozessen in der Gruppe des Sendeprozesses geleitet (siehe tty(4)).

Ist er Prozessidentifikator -1 und der Nutzer der Superuser, wird das Signal an alle Prozesse mit Ausnahme der Prozesse 0 und 1, des Schedulers und des Initialisierungsprozesses (siehe init(8)) gesendet.

Prozesse koennen Signale zu sich selbst senden.

**SIEHE AUCH**

signal(2), kill(1)

**DIAGNOSTIK**

Null wird zurueckgegeben, wenn der Prozess beendet ist. -1 wird zurueckgegeben, wenn der Prozess nicht die gleiche effektive Nutzeridentifikation hat und der Nutzer nicht der Superuser ist, oder wenn der Prozess nicht existiert.

**ASSEMBLER**

```
(KILL:=37)
(pid in r0)
(sig in r1)
sc #KILL
```

---

**NAME**

link - Link zu einem File

**UEBERSICHT**

```
link(name1,name2)
char *name1, *name2;
```

**BESCHREIBUNG**

Link erzeugt eine Directoryeintragung (Eintrittspunkt) zu einem File.
Name_1 ist dabei eine bereits existierende Eintragung zum File, name_2 ist die zu erzeugende Directoryeintragung zu File. Der Linkzaehler im i-node des Files wird um 1 erhoeht.

Name_1 und name_2 sind Zeiger auf willkuerliche Pfadnamen, die durch Nullbyte abgeschlossen sind.

**SIEHE AUCH**

ln(1), unlink(2)

**DIAGNOSTIK**

Null wird zurueckgegeben, wenn ein Link erzeugt wird; -1 wird zurueckgegeben, wenn name1 nicht gefunden werden kann, wenn name2 schon existiert, wenn das Directory von name2 nicht beschrieben werden kann, wenn durch einen Nutzer, der nicht der Superuser ist, versucht wird, ein Link zu einem File auf einem anderen Filesystem zu erzeugen, wenn ein File zu viele Links hat.

**ASSEMBLER**

```
(LINK :=9)
(name 1 in r0)
(name 2 in r1)
 sc  #LINK
```

---

**NAME**

lock - Festhalten eines Prozesses im internen Speicher

**UEBERSICHT**

**lock(flag)**

**BESCHREIBUNG**

Wenn das flag Argument ungleich Null ist, wird der Prozess, der diesen Ruf ausfuehrt, nicht ausgelagert, ausser wenn verlangt wird, dass der Prozess sich vergroessert. Wenn das Argument Null ist, wird der Prozess nicht im internen Speicher festgehalten. Dieser Ruf kann nur vom Superuser ausgefuehrt werden.

**FEHLERQUELLEN**

Festgehaltene(locked) Prozesse beeinflussen die Verdichtung des internen Speichers und koennen eine Systemblockierung verursachen. Dieser Systemruf wird nicht als staendiger Teil des Systems beruecksichtigt.

**ASSEMBLER**

```
(LOCK:=53)
(flag in r0)
sc #LOCK
```



---

**NAME**

lseek - Stellen des Lese/Schreib-Zeigers

**UEBERSICHT**

```
long lseek(fildes,offset,whence)
long offset;
```

**BESCHREIBUNG**

Der Filedeskriptor fildes bezieht sich auf ein fuer Lesen oder Schreiben offenes File. Der Lese- (bzw. Schreib-) Zeiger fuer das File wird wie folgt gesetzt:

Ist whence 0, wird der Zeiger auf offset bytes gesetzt (Beginn + offset).

Ist whence 1, wird der Zeiger auf seinen augenblicklichen Speicherplatz plus offset gesetzt.

Ist whence 2, wird der Zeiger auf die Laenge des Files plus offset gesetzt (Ende + offset).

Der zurueckgegebene Wert ist der Lese- (bzw. Schreib-) Zeiger auf den sich ergebenden neuen aktuellen Speicherplatz fuer nachfolgende read(2) oder write(2).

Das Stellen des Zeigers weit hinter das Ende des Files und ein folgendes Schreiben erzeugt eine Luecke, die keinen physischen Speicherplatz belegt und als Null gelesen wird.

**SIEHE AUCH**

open(2), creat(2), fseek(3)

**DIAGNOSTIK**

-1 (Pruefung auf (0xFFFF0000 bei long) wird zurueckgegeben bei einem undefinierten Filedescriptor, beim Suchen auf eine Pipe oder beim Suchen nach einer Position vor dem Anfang des Files.

**FEHLERQUELLEN**

Lseek kann nicht fuer Zeichen-Special-Files angewendet werden.

**ASSEMBLER**

```
(LSEEK:=19)
(Filedeskriptor in r0)
(hoeheres Wort von offset in r1)
(niederes Wort von offset in r2)
(whence in r3)
sc #LSEEK
(Zeiger in r4)
```

**MKNOD(2)** **MKNOD(2)**

---

**NAME**

mknod - Erzeugen eines Directory oder eines Special-Files

**UEBERSICHT**

```
mknod(name,mode,addr)
char *name;
```

**BESCHREIBUNG**

Mknod erzeugt ein neues File, wobei name der Zeiger auf den Filenamens ist, welcher durch ein Nullbyte beendet wird. Der Modus des neuen Files (einschliesslich der Directory- und Special-File-Bits) wird von mode initialisiert. (Der Schutzteil des Modus wird durch die Modusmaske des Prozesses modifiziert; siehe umask(2)).

Der erste Blockzeiger des i-node wird von addr initialisiert. Fuer gewoehnliche Files und Directories ist addr normalerweise 0. Im Fall eines Special-Files spezifiziert addr die Art des Special-Files.

Mknod kann nur vom Superuser ausgefuehrt werden.

**SIEHE AUCH**

mkdir(1), mknod(1), filsys(5)

**DIAGNOSTIK**

Null wird zurueckgegeben, wenn das File erzeugt wurde; -1 wird zurueckgegeben, wenn das File schon existiert oder der Nutzer nicht der Superuser ist.

**ASSEMBLER**

```
(MKNOD :=14)
(name in r0)
(mode in r1)
(addr in r2)
 sc # MKNOD
```



---

**NAME**

mount, umount - Eingliedern oder Entfernen eines Filesystems

**UEBERSICHT**

```
mount(special,name,rwflag)
char *special, *name;

umount(special)
char *special;
```

**BESCHREIBUNG**

Mount teilt dem System mit, dass ein wechselbares Filesystem auf dem blockstruckturierten Special-File special eingegliedert werden soll. Von nun an kann das eingegliederte Filesystem unter den File name erreicht werden, d.h. name bezieht sich jetzt auf das Root-File des eingegliederten Filesystems. Special und name sind Zeiger auf Pfadnamen, welche durch ein Nullbyte abgeschlossen sind.

Name muss schon existieren und ein Directory sein (falls die Wurzel des eingegliederten Filesystems kein Directory ist). Sein urspruenglicher Inhalt ist, waehrend das Filesystem eingegliedert ist, unzugaenglich.

Das rwflag Argument bestimmt, ob das Filesystem beschrieben werden kann. Ist es Null, ist Schreiben erlaubt. Ist es ungleich Null, darf nicht mehr geschrieben werden. Physisch schreibgeschuetzte Filesysteme muessen mit dem Status "nur Lesen" eingegliedert werden, da sonst Fehler auftreten, wenn entweder die Zugriffszeiten aktualisiert werden oder irgendein explizites Schreiben versucht wird.

Umount teilt dem System mit, dass aus dem spezial File ein wechselbares Filesystem ausgegliedert werden soll. Der urspruengliche Inhalt des verbundenen Files (Directory) name wird wieder zugaenglich.

**SIEHE AUCH**

mount(1)

**FEHLERQUELLEN**

Mount gibt 0 zurueck, wenn die Aktion erfolgt ist. -1 wird zurueckgegeben, wenn special nicht zugreifbar ist oder kein passendes File ist, wenn name nicht existiert, wenn special schon eingegliedert ist, wenn name im Gebrauch ist, oder wenn es schon zu viele eingegliederte Filesysteme gibt.

Umount gibt 0 zurueck, wenn die Aktion erfolgt ist. -1 wird zurueckgegeben, wenn das Special-File nicht zugreifbar ist oder kein eingegliedertes Filesystem enthaelt oder wenn aktive Files im eingegliederten Filesystem sind.

**ASSEMBLER**

```
(MOUNT =21)
(special in r0)
(name    in r1)
(rw flag in r2)
 sc  #MOUNT


(UMOUNT :=22)
(special in r0)
 sc #UMOUNT
```

---

**NAME**

nice - Setzen der Programmprioritaet

**UEBERSICHT**

**nice(incr)**

**BESCHREIBUNG**

Die eingetragene Prioritaet des Prozesses wird durch incr veraendert. Positive Prioritaeten erhalten weniger Dienste als normal. Die Prioritaet 10 wird Nutzern empfohlen, die lang laufende Programme ohne Beeinflussung durch die Systemverwaltung auszufuehren wuenschen.

Negative Werte werden nur vom Superusers akzeptiert. Die Prioritaet ist begrenzt auf den Bereich -20 (am dringlichsten) bis 20 (am niedrigsten).

Die Prioritaet eines Prozesses wird durch fork(2) an den Child-Prozess gegeben. Um bei einem privilegierten Prozesses von einem unbekannten Status zur normalen Prioritaet zurueckzukehren, sollte nice nacheinander mit den Argumenten -40 (fuehrt zur Prioritaet -20 infolge der Begrenzung), 20 (um 0 zu erhalten) und dann 0 (um die Kompatibilitaet mit frueheren Versionen dieses Rufes aufrecht zu erhalten) gerufen werden.

**ASSEMBLER**

```
(NICE:=34)
(iner in r0)
sc #NICE
```



---

**NAME**

open - Oeffnen fuer Lesen oder Schreiben

**UEBERSICHT**

```
open(name,mode)
char *name;
```

**BESCHREIBUNG**

Open oeffnet das File name fuer Lesen (wenn mode 0 ist), Schreiben (wenn mode 1 ist) oder fuer Lesen und Schreiben (wenn mode 2 ist). Name ist ein Zeiger auf eine Zeichenkette, die einen Pfadnamen darstellt und durch ein Nullbyte beendet wird.

Das File wird auf den Anfang positioniert (Byte 0). Der zurueckgegebene Filedeskriptor fildes muss fuer nachfolgende Rufe fuer andere Ein-/Ausgabefunktionen auf das File benutzt werden.

**SIEHE AUCH**

create(2), read(2), write(2), dup(2), close(2)

**DIAGNOSTIK**

Der Wert -1 wird zurueckgegeben, wenn das File nicht existiert, wenn eine der notwendigen Directories nicht existiert oder nicht lesbar ist, wenn das File nicht lesbar (bzw. schreibbar) ist oder wenn zu viele Files offen sind.

**ASSEMBLER**

```
(OPEN:=5)
(name in r0)
(mode in r1)
sc #OPEN
(Fildeskriptor in r4)
```

**PAUSE(2)** **PAUSE(2)**

---

**NAME**

pause - Halt bis zum Signal

**UEBERSICHT**

**pause()**

**BESCHREIBUNG**

Pause kehrt niemals normal zurueck. Der Ruf dient dazu, die Steuerung abzugeben, waehrend auf den Empfang eines Signals von kill(2) oder alarm(2) gewartet wird.

**SIEHE AUCH**

kill(1), kill(2), alarm(2), signal(2), setjmp(3)

**ASSEMBLER**

```
(PAUSE:=29)
sc #PAUSE
```

**NAME**

pipe - Erzeugen eines Kanals zwischen Prozessen

**UEBERSICHT**

```
pipe(fildes)
int fildes[2];
```

**BESCHREIBUNG**

Der pipe Systemruf erzeugt einen E/A-Mechanismus, der als Pipe bezeichnet wird. Die zurueckgegebenen Filedeskriptoren koennen in Lese- und Schreiboperationen benutzt werden. Wird die Pipe unter Benutzung des Deskriptors fildes[1] beschrieben, werden bis zu 4096 Bytes gepuffert, bevor der Schreibprozess unterbrochen wird. Ein Lesen, das den Deskriptor fildes[0] benutzt, nimmt die Daten auf. Das Schreiben mit einem Zaehler von 4096 Bytes oder weniger ist nicht teilbar, d.h. kein anderer Prozess kann Daten einstreuen.

Es wird vorausgesetzt, dass, nachdem die Pipe aufgebaut wurde, zwei (oder mehr) kooperierende Prozesse, die durch nachfolgende fork Rufe erzeugt werden, die Daten durch die Pipe mit read und write Rufen geben werden.
Die Shell verfuegt ueber eine Syntax, mit der ein lineares Feld von Prozessen, die durch Pipes verbunden sind, erzeugt werden kann.

Leserufe auf eine leere Pipe (keine gepufferten Daten) mit nur einem Ende (alle Schreib-Filedeskriptoren sind geschlossen) geben ein End-of-file zurueck.

**SIEHE AUCH**

sh(1), read(2), write(2), fork(2)

**DIAGNOSTIK**

Der Funktionswert 0 wird zurueckgegeben, wenn die Pipe erzeugt wurde. Der Wert -1 wird zurueckgegeben, wenn schon zu viele Files offen sind. Ein Signal wird erzeugt, wenn ein Schreiben auf eine Pipe mit nur einem Ende versucht wird.

**FEHLERQUELLEN**

Sind mehr als 4096 Bytes in irgendeiner Pipe in einer Schleife von Prozessen notwendig, tritt eine Systemblockierung auf.

**ASSEMBLER**

```
(PIPE:=42)
sc #PIPE
(Lesefiledeskriptor in r4)
(Schreibfiledeskriptor in r5)
```

**PTRACE(2)** **PTRACE(2)**

---

**NAME**

ptrace - Prozessverfolgung

**UEBERSICHT**

```
#include <signal.h>

ptrace(request,pid,addr,data)
iut *addr;
```

**BESCHREIBUNG**

Ptrace liefert ein Mittel, mit dem ein Parent-Prozess die Ausfuehrung eines Child-Prozesses steuern, pruefen und sein Speicherabbild aendern kann. Seine primaere Anwendung ist der Test mit Unterbrechungspunkten. Es gibt vier Argumente, deren Interpretation von einem request Argument abhaengt. Allgemein ist pid die Prozessidentifikation des zu verfolgenden Prozesses, der ein Child-Prozess (kein entfernter Abkoemmling) des verfolgenden Prozesses sein muss. Ein Prozess, der verfolgt wird, verhaelt sich normal bis er irgendeinem Signal empfaengt, das intern erzeugt wird, wie "verbotener Befehl" - oder extern erzeugt wird, wie "Interrupt". Siehe signal(2) fuer die Liste der Signale. Dann tritt der zu verfolgende Prozess in einen Stopstatus und sein Parent-Prozess wird ueber wait(2) informiert. Ist der Child-Prozess im Stopstatus, kann sein Speicherabbild durch die Benutzung von ptrace geprueft und modifiziert werden. Wenn gewuenscht, kann dann eine andere ptrace- Anforderung bewirken, den Child-Prozess zu beenden oder fortzusetzen und moeglicherweise das Signal zu ignorieren.

Der Wert des Argumentes request bestimmt die genaue Aktion des Rufes:

0 Diese Anforderung wird nur einmal durch den Child-Prozess benutzt. Sie deklariert, dass der Prozess durch seinen Parent-Prozess verfolgt werden soll. Alle anderen Argumente werden ignoriert. Es ergeben sich eigenartige Ergebnisse, wenn der Parent-Prozess nicht erwartet, den Child-Prozess zu verfolgen.

1,2 Das Wort von addr im Adressbereich des Child-Prozesses wird zurueckgegeben. Sind der Befehls- und Datenbereich getrennt, zeigt die Anforderung 1 den Befehlsbereich an, Anforderung 2 den Datenbereich. Addr muss gerade sein. Der Child-Prozess muss gesoppt sein. Die Eingabedaten data werden ignoriert.

3 Das Wort des Prozessdatenbereiches des Systems, das mit addr uebereinstimmt, wird zurueckgegeben. Addr muss gerade sein und kleiner als 512. Dieser Bereich enthaelt die Register und andere Informationen ueber den Prozess; sein Aufbau stimmt mit der User-Struktur in dem System ueberein.

4,5 Die gegebenen Daten data werden auf das Wort im Adressbereich des Prozesses geschrieben, das mit addr uebereinstimmt und gerade sein muss. Es wird kein verwendbarer Wert zurueckgegeben. Sind der Befehls- und Datenbereich getrennt, zeigt die Anforderung 4 den Befehlsbereich an, Anforderung 5 den Datenbereich. Versuche, in eine Prozedur zu schreiben, fuehren zu einem Fehler, wenn ein anderer Prozess das gleiche File ausfuehrt.

6 Die Systemdaten des Prozesses werden geschrieben, wie sie mit Anforderung 3 gelesen werden. Nur wenige Speicherplaetze koennen in dieser Weise beschrieben werden: die allgemeinen Register, der Gleitkommastatus und die Register und besondere Bits des Prozessor-Statuswortes.

7 Das data Argument wird als eine Signalnummer genommen und die Ausfuehrung des Child-Prozesses bei dem Speicherplatz addr fortgesetzt, wenn er dieses Signal empfangen hat. Normalerweise wird die Signalnummer entweder Null sein, um anzuzeigen, dass das Signal, das den Halt verursacht, ignoriert werden soll - oder dieser Wert wird auf das Prozessabbild gebracht, um anzuzeigen, welches Signal den Stop verursacht. Wenn addr (int *)1 ist, wird die Ausfuehrung fortgesetzt, wo sie gestoppt wurde.

8 Der verfolgte Prozess endet.

9 Die Ausfuehrung des ueberwachten Prozesses wird, wie bei Anforderung 7, fortgesetzt, jedoch nach Ausfuehrung einer Instruktion sofort wieder gestoppt. Die Signalnummer des Halts ist SIGTRAP. (Fuer diese Einzelschrittabarbeitung wird eine spezielle Hardware benutzt, die, wenn eingeschaltet, nach einer feststehenden Zahl von Stackzugriffen eine Unterbrechnung ausloest.) Diese Anforderung ist Teil des Mechanismus fuer die Ausfuehrung von Unterbrechnungspunkten.

Wie angezeigt, koennen diese Rufe (ausser fuer Anforderung 0) nur benutzt werden, wenn der Subjektprozess gestoppt ist. Der wait- Ruf wird benutzt, um festzustellen, wenn ein Prozzess stoppt; in solchem Fall hat der Beendigungsstatus, der durch wait zurueckgegeben wird, den Wert 0177, um den Halt anzuzeigen - im Gegensatz zu einer echten Beendigung.

Um moeglichem Missbrauch zuvorzukommen, ist es fuer ueberwachte Prozesse verboten, durch einen exec(2)-Ruf die Nutzeridentifikation zu aendern (S.ISUID-Bit, siehe STAT(2), CHMOD(2)). Gibt ein verfolgter Prozess einen exec(2)-Ruf, so wird er unmittelbar vor Ausfuehrung der ersten Instruktion des neuen Programms gestoppt. Das dabei angezeigte Signal ist SIGTRAP. Durch diese Einrichtung wird es dem ueberwachenden Prozess ermoeglicht, Unterbrechungspunkte zu setzen.

**SIEHE AUCH**

wait(2), signal(2), adb(1)

**DIAGNOSTIK**

Der Wert -1 wird zurueckgegeben, wenn request ungueltig ist oder pid kein verfolgter Prozess ist oder addr ausserhalb der Grenzen ist oder data eine illegale Signalnummer angibt.

**FEHLERQUELLEN**

Der Ruf mit der Anforderung 0 ist in der Lage, Signale anzugeben, die normal behandelt werden soll und keinen Halt verursachen. In dieser Weise koennen Programme mit simuliertem Gleitkomma (die die Signale 'Verbotener Befehl' in hoher Anzahl benutzen) wirksam getestet werden. Die Fehleranzeige -1 ist ein legitimer Funktionswert; errno, siehe intro(2), kann unzweideutig benutzt werden.

Es ist moeglich, einen Prozess beim Auftreten eines Systemrufes zu stoppen; in dieser Weise kann eine vollstaendig gesteuerte Umgebung geliefert werden.

**ASSEMBLER**

```
(PTRACE:=26)
(reg in r0)
(pid in r1)
(addr in r2)
(data in r3)
sc #PTRACE
(Wert in r4)
```



---

**NAME**

read - Lesen eines Files

**UEBERSICHT**

```
read(fildes,buffer,nbytes)
char *buffer;
```

**BESCHREIBUNG**

Ein Filedeskriptor fildes ist ein Wort, das von einem erfolgreichen open, creat, dup oder pipe Ruf zurueckgegeben wird. Buffer ist die Speicherplatzadresse von nbytes zusammenhaengenden Bytes, die vom Eingabefile gelesen werden. Es ist nicht garantiert, dass alle nbytes Bytes gelesen werden. Bezieht sich z.B. das File auf die Tastatur, wird hoechstens eine Zeile eingelesen.

Ist der zurueckgegebene Wert 0, wurde End-of-file erreicht.
Ein zurueckgegebener Wert > 0 entspricht der Anzahl tatsaechlich gelesener Bytes.

**SIEHE AUCH**

open(2), creat(2), dup(2), pipe(2)

**DIAGNOSTIK**

Wie erwaehnt, wird 0 zurueckgegeben, wenn das Ende des Files erreicht worden ist. War das Lesen in anderer Weise nicht erfolgreich, ist der Rueckkehrwert -1. Viele Bedingungen koennen einen Fehler erzeugen: physische E/A-Fehler, falsche Pufferadresse, widersinnige Byteanzahl nbytes , ein Filedeskriptor, der sich nicht auf ein Eingangsfile bezieht.

**ASSEMBLER**

```
(READ:=3)
(fildes in r0)
(buffer in r1)
(nbytes in r2)
sc #READ
(Bytezaehler in r4)
```

---

**NAME**

setuid, setgid- Setzen der Nutzer- und Gruppenidentitaet

**UEBERSICHT**

```
setuid(uid)

setgid(gid)
```

**BESCHREIBUNG**

Die Nutzeridentifikation (Gruppenidentifikation) des laufenden Prozesses wird auf das Argument gesetzt. Sowohl die effektive als auch die reale Identifikation werden gesetzt. Diese Rufe sind nur dem Superuser erlaubt oder dann, wenn das Argument die reale Identifikation ist.

**SIEHE AUCH**

getuid(2)

**DIAGNOSTIK**

Null wird zurueckgegeben, wenn die Nutzer- (Gruppen-) identifikation gesetzt ist; andernfalls wird -1 zurueckgegeben.

**ASSEMBLER**

```
(SETUID:=23)
(uid in r0)
sc #SETUID

(SETGID:=46)
(gid in r0)
sc #SETGID
```



---

**NAME**

signal- Behandeln oder Ignorieren eines Signals

**UEBERSICHT**

```
#include <signal.h>

(*signal(sig,func))()
(*func)()
```

**BESCHREIBUNG**

Ein Signal wird durch irgendein nicht normales Ereignis erzeugt, das entweder durch einen Nutzer auf der Tastatur (Quittung, Interrupt), durch einen Programm bzw. Hardwarefehler oder durch die Anforderung von einem anderen Programm (kill) initiiert wird. Normalerweise verursachen alle Signale eine Beendigung des Prozesses, der die Signale empfaengt. Durch einen signal-Ruf kann veranlasst werden, dass sie entweder ignoriert werden oder einen Sprung zu einer Behandlungsroutine bewirken.

Die nachfolgende Liste enthaelt die Signale mit den Namen, wie sie in dem Include-File enthalten sind.

| | | |
|---|---|---|
| SIGHUP | 1 | DFUE: Verlust der Traegerfrequenz |
| SIGINT | 2 | Interrupt |
| SIGQUIT | 3* | Quittung |
| SIGILL | 4* | illegaler Befehl (nicht zurueckgesetzt, wenn behandelt) |
| SIGTRAP | 5* | Trace Trap (Istruktion '0x7c06', d.h., 'ei nvi'; im Nutzer-Modus gegeben, bewirkt dieses Signal, dass nach Behandlung nicht zurueckgegeben wird). |
| SIGIOT | 6* | IOT Trap (Istruktion '0x7c05', d.h., 'ei vi' im Nutzermodus gegeben, bewirkt dieses Signal). |
| SIGEMT | 7* | EMT Trap (Instruktion '0x7c04', d.h., 'ei vi,nvi' im Nutzermodus gegeben, bewirkt dieses Signal). |
| SIGFPE | 8* | Gleitkommaverletzung |
| SIGKILL | 9 | kill (kann nicht behandelt oder ignoriert werden) |
| SIGBUS | 10* | Busfehler |
| SIGSEGV | 11* | Segmentierungsverletzung |
| SIGSYS | 12* | falsche Argumente bei einem Systemruf |
| SIGPIPE | 13 | Schreiben in eine Pipe ohne Leseprozess |
| SIGALRM | 14 | Alarmuhr |
| SIGTERM | 15 | Software-Beendigungssignal |
| | 16 | nicht zugeordnet |

Die mit Stern gekennzeichneten Signale in der Liste verursachen einen Speicherabzug, wenn sie nicht behandelt oder ignoriert werden.

Durch func SIG_DFL wird die Standardaktion fuer das Signal sig wieder eingestellt. Diese Standardaktion bewirkt den Abbruch des Prozesses.
Ist func SIG_IGN wird das signal ignoriert. Tritt das Signal auf, wird die Funktion func mit der Signalnummer als Argument gerufen. Nach der Rueckkehr von der Funktion wird der Prozess an dem Punkt fortgesetzt, an dem er unterbrochen wurde.

Wird ein Signal nicht ignoriert, dann wird automatisch seine Standardaktion wieder eingesetzt. Soll das Signal weiterhin abgefangen werden, muss in der Behandlungsroutine ein erneuter signal-Ruf ausgefuehrt werden.

Tritt ein behandeltes Signal waehrend bestimmter Systemrufe auf, endet der Ruf vorzeitig. Im Besonderen kann dies eintreten waehrend eines read (2)oder write(2) auf ein langsames peripheres Geraet oder waehrend pause(2) oder wait(2). Tritt ein solches Signal auf, wird der gerettete Nutzerstatus in der Art angeordnet, wie bei einer Rueckkehr von der Signalbehandlung, d.h., der Systemruf kehrt mit einem Fehlerstatus zurueck. Das Programm des Nutzers kann, wenn gewuenscht, den Ruf erneut ausfuehren.
Der Wert von signal ist der vorhergehende (oder initialisierte) Wert von func fuer das einzelne Signal. Nach einem fork(2) erbt der Child-Prozess alle Signale. Exec(2) setzt alle behandelten Signale auf die Standardaktion zurueck.

**SIEHE ALSO**

kill(1), kill(2), ptrace(2), setjump(3)

**DIAGNOSTIK**

Der Wert (int) -1 wird zurueckgegeben, wenn das gegebene Signal ausserhalb des Bereiches liegt.

**FEHLERQUELLEN**

Tritt ein Signal erneut auf, bevor das letzte zurueckgesetzt werden konnte, besteht keine Moeglichkeit es zu behandeln.

Die Typspezifikation der Routine und ihr func Argument sind problematisch.

**ASSEMBLER**

```
(SIGNAL:=48)
(sig in r0)
(Marke in r1)
sc #SIGNAL
(die alte Marke in r4)
```

Wenn Marke 0 ist, wird die Standardaktion wieder eingesetzt.
Ist Marke ungerade, wird das Signal beim naechsten Auftreten ignoriert.
Jede andere gerade Marke wird als Adresse im Prozess interpretiert, an der (asynchron) die Arbeit des Prozesses fortgesetzt wird, nach dem das Signal aufgetreten ist. Diese asynchrone Exit-Routine findet auf dem Stack folgende Verhaeltnisse vor:

```
sp --> ident                        |
       Flag- und Steuerwort         |    Prozessorstatus, zum
       segment                      >    Zeitpunkt der letzten
       offset                       |    Unterbrechung des
                                    |    Prozessors
```

alter sp --> (vor letzter Unterbrechung)

Zur Rueckkehr zum unterbrochenen Befehl (oder zu einem anderen) mit eventuell modifiziertem Flag- und Steuerwort ist es (nach eventueller Modifikation des Stack) erforderlich, einen weiteren Systemruf zu geben:

```
(MIRET:=0)
(sp wie bei Eintritt in asynchrone Routine)
sc #MIRET
```



---

**NAME**

stat,fstat- Erhalten des Filestatus

**UEBERSICHT**

```
#include <sys/types.h>
#include <sys/stat.h>

stat(name,buf)
char *name;
struct stat *buf;

fstat(fildes,buf)
struct stat *buf;
```

**BESCHREIBUNG**

Stat stellt detaillierte Informationen ueber ein bezeichnetes File name bereit. Fstat stellt die gleichen Informationen ueber ein offenes File zur Verfuegung, das durch den Filedeskriptor nach einem erfolgreichen open, creat, dup oder pipe(2) Ruf bekannt ist.

Name zeigt auf eine durch ein Nullbyte abgeschlossene Zeichenkette, die ein File bezeichnet; buf ist die Adresse eines Puffers, in den Informationen gebracht werden, die das File betreffen. Es ist unnoetig, eine Erlaubnis fuer den Zugriff auf das File zu haben, aber alle Directories, die zu dem File fuehren, muessen suchbar sein. Der Aufbau der in <stat.h> definierten Struktur, auf die durch buf gezeigt wird, ist unten gegeben. St_mode wird entschluesselt, entsprechend den '#define' Anweisungen.

```
struct    stat
{
     dev_t     st_dev;
     ino_t     st_ino;
     unsigned short st_mode;
     short     st_nlink;
     short     st_uid;
     short     st_gid;
     dev_t     st_rdev;
     off_t     st_size;
     time_t    st_atime;
     time_t    st_mtime;
     time_t    st_ctime;
};

#define   S_IFMT    0170000        /* Filetyp          */
#define   S_IFDIR   0040000        /* Directory        */
#define   S_IFCHR   0020000        /* special zeichen- */
                                   /* orientiert       */
#define   S_IFBLK   0060000        /* special block-   */
                                   /* orientiert       */
#define   S_IFREG   0100000        /* regulaer         */
#define   S_IFMPC   0030000        /* special zeichen- */
                                   /* orientiert       */
                                   /* multiplex        */

#define   S_IFMPB   0070000        /* special block-   */
                                   /* orientiert       */
                                   /* multiplex        */
#define   S_ISUID   0004000        /* Setzen Eigen-    */
                                   /* tuemer ID bei    */
                                   /* Ausfuehrung      */
#define   S_ISGID   0002000        /* Setzen Gruppen   */
                                   /* ID bei Aus-      */
                                   /* fuehrung         */
#define   S_ISVTX   0001000        /* Retten Textab-   */
                                   /* bild nach Aus-   */
                                   /* fuehrung         */
#define   S_IREAD   0000400        /* Leseerlaubnis    */
                                   /* durch den Eigen- */
                                   /* tuemer           */
#define   S_IWRITE  0000200        /* Schreiberlaubnis */
                                   /* durch den Eigen- */
                                   /* tuemer           */
#define   S_IEXEC   0000100        /* Ausfuehren       */
                                   /* (Suchen) durch   */
                                   /* den Eigentuemer  */
```

Die Modusbits 0000070 und 0000007 entschluesseln die Erlaubnisse fuer die Gruppe und andere (siehe chmod(2)). Die definierten Typen ino_t, off_t, time_t bezeichnen verschieden lange Integerwerte; dev_t entschluesselt den Geraetetyp und die Geraetnummer; ihre exakten Definitionen sind in dem Include-File <sys/types.h> enthalten (siehe types(5)).

Ist fildes mit einer Pipe verbunden, berichtet fstat wie ueber ein normales File mit beschraenkten Erlaubnissen. Die Laenge ist die Anzahl der Bytes, die in der Schlange in der Pipe gefuehrt werden.

st_atime speichert die Zeit des letzten lesenden Zugriffs auf das File. Aus Gruenden der Wirksamkeit wird sie nicht gesetzt, wenn ein Directory gesucht wird, obwohl dies logischer waere. st_mtime ist die Zeit, wann das File zuletzt beschrieben bzw. erzeugt wurde. Sie wird nicht gesetzt bei Aenderungen des Eigentuemers, der Gruppe, Link-Zaehlungen oder des Modus. st_ctime wird sowohl beim Schreiben als auch beim Aendern des i-nodes gesetzt.

**SIEHE AUCH**

ls(1), filsys(5)

**DIAGNOSTIK**

Null wird zurueckgegeben, wenn der Status verfuegbar ist. -1, wenn das File nicht gefunden werden kann.

**ASSEMBLER**

```
(STAT:=18)
(name in r0)
(buf in r1)
sc #STAT

(FSTAT:=28)
(fildes in r0)
(buf in r1)
sc #FSTAT
```

**STIME(2)** **STIME(2)**

---

**NAME**

**stime - Setzen der Zeit**

**UEBERSICHT**

```
stime(tp)
long *tp;
```

**BESCHREIBUNG**

Stime setzt Zeit und Datum im System. Die Zeit, auf die durch tp gezeigt wird, wird in Sekunden vom 1. Jan 1970 00.00 Uhr Weltzeit an gemessen. Nur der Superuser kann diesen Ruf ausfuehren.

**SIEHE AUCH**

date(1), time(2), ctime(3)

**DIAGNOSTIK**

Null wird zurueckgegeben, wenn die Zeit gesetzt ist. -1, wenn der Nutzer nicht Superuser ist.

**ASSEMBLER**

```
(STIME := 25)
(Zeit in rr0)
 sc #STIME
```

**SYNC(2)** **SYNC(2)**

---

**NAME**

sync - Aktualisieren des Superblocks

**UEBERSICHT**

**sync()**

**BESCHREIBUNG**

Sync veranlasst, dass alle im Hauptspeicher befindlichen Informationen fuer Disketten auf diese geschrieben werden. Dies betrifft die modifizierten Superbloecke, geaenderte i-nodes und verzoegerte Ausgabebloecke.

Sync sollte von Programmen verwendet werden, die Filesysteme pruefen, z.B. icheck.

Sync bewirkt kein generelles Ungueltigmachen der im HS gehaltenen Diskettenin formationen.

**SIEHE AUCH**

sync(1), update(8)

**FEHLERQUELLEN**

Das Schreiben wird von sync angestossen, ist aber bei Rueckkehr von sync nicht notwendig abgeschlossen.

**ASSEMBLER**

```
(SYNC := 36)
 sc  # SYNC
```

---

**NAME**

time,ftime - Erhalten von Datum und Zeit

**UEBERSICHT**

```
long time(0)

long time(tloc)
long *tloc;

#include <sys/types.h>
#include <sys/timeb.h>
ftime(tp)
struct timeb *tp;
```

**BESCHREIBUNG**

Time gibt die Zeit vom 1. Januar 1970 00:00:00 Uhr Weltzeit an, gemessen in Sekunden, zurueck.

Wenn tloc ungleich 0 ist, wird der Rueckgabewert auch auf dem Platz gespeichert, auf den tloc zeigt.
Der ftime Eintrittspunkt fuellt eine Struktur, auf die durch sein Argument gezeigt wird und die durch <sys/timeb.h> definiert ist.

```
/*
 * Structure returned by ftime system call
 */
struct timeb {
     time_t   time;
     unsigned short millitm;
     short    timezone;
     short    dstflag;
};
```

Die lokale Struktur enthaelt die Zeit des Zeitabschnittes in Sekunden, bis zu 1000 Millisekunden geben dieses Intervall praeziser an, die lokale Zeitzone (gemessen in Minuten der Zeit westlich von Greenwich) und ein Flag, das, wenn ungleich Null, anzeigt, dass die Sommerzeit oertlich waehrend eines bestimmten Teil des Jahres eingefuehrt wird.

**SIEHE AUCH**

date(1), stime(2), ctime(3)

**ASSEMBLER**

```
(FTIME := 35)
(tp in r0)
sc # FTIME
(TIME := 13 !veralteter Ruf!)
sc # TIME
(Zeit seit 1.1. 1970 in Sekunden in r4)
```

**NAME**

times - Erhalten der Prozesszeiten

**UEBERSICHT**

```
times(buffer)
struct tbuffer *buffer;
```

**BESCHREIBUNG**

Times gibt Informationen ueber den Zeitablauf des laufenden Prozesses und die beendeten Child-Prozesse des laufenden Prozesses. Alle Zeiten sind in 1/60 Sekunde angegeben.

Nach dem Ruf wird der Puffer wie folgt gefuellt:

```
struct    tbuffer    {
     long proc_user_time;
     long proc_system_time;
     long child_user_time;
     long child_system_time;
};
```

Die 'child-time' sind die Summe der Zeiten der Child-Prozesse und der Zeiten ihrer Child-Prozesse.

**SIEHE AUCH**

time(2)

**ASSEMBLER**

```
(TIMES:=43)
(buffer in r0)
sc #TIMES
```

**UMASK(2)** **UMASK(2)**

**NAME**

umask - Setzen einer Modusmaske fuer die Fileerzeugung

**UEBERSICHT**

```
umask(complmode)
```

**BESCHREIBUNG**

Umask setzt eine Maske, die immer dann verwendet wird, wenn ein File durch creat(2) oder mknod(2) in einem Prozess erzeugt wird. Der aktuelle Modus (siehe chmod(2)) des neu erzeugten Files ist das logische UND des gegebenen Modus und des Komplements des Arguments. Nur die 9

niederwertigen Bits der Maske (die Schutzbits) werden berücksichtigt. Mit anderen Worten: die Maske zeigt die Bits, die ausgeschaltet werden, wenn Files erzeugt werden.

Der vorherige Wert der Maske wird durch den Ruf zurueckgegeben. Der Wert ist anfangs 0 (keine Beschraenkungen). Die Maske wird den Child-Prozessen vererbt.

**SIEHE AUCH**

creat(2), mknod(2), chmod(2)

**ASSEMBLER**

```
(UMASK:=  60)
(complmode in r0)
sc #UMASK
```



---

**NAME**

unlink - Loeschen von Directory-Eintragungen

**UEBERSICHT**

```
unlink(name)
char *name;
```

**BESCHREIBUNG**

Name ist der Zeiger auf einen Filenamen, der durch ein Nullbyte beendet wird. Unlink loescht die Eintragung fuer dieses File aus seiner Directory. War der Eintrag der letzte Link zu dem File, wird der Inhalt des Files freigegeben und das File zerstoert. Wurde jedoch das File in irgendeinem Prozess geoeffnet, wird die aktuelle Zerstoerung verzoegert bis es geschlossen ist, selbst wenn der Directory-Eintrag verschwunden ist.

**SIEHE AUCH**

rm(1), link(2)

**FEHLERQUELLEN**

Normalerweise wird Null zurueckgegeben; -1 zeigt an, dass das File nicht existiert, dass sein Directory nicht beschrieben werden kann oder dass das File einen reinen Prozedurtext enthaelt, der augenblicklich in Gebrauch ist. Schreiberlaubnis wird fuer das File selbst nicht verlangt. Nur der Superuser darf ein Directory loeschen.

**ASSEMBLER**

```
(UNLINK := 10)
(name in r0)
 sc  #UNLINK
```

---

**NAME**

utime - Setzen der Filezeiten

**UEBERSICHT**

```
#include <sys/types.h>
utime (file,timep)
char *file;
time_t timep[2];
```

**BESCHREIBUNG**

Der Ruf utime benutzt die Zeit des letzten Zugriffs und die Zeit der letzten Modifikation in dieser Reihenfolge von dem timep-Vektor, um die entsprechenden Zeiten fuer file (i-node Eintragung) zu setzen.
Die Zeit der letzten Aenderung des inodes wird auf die augenblickliche Zeit gesetzt.

Der Benutzer des Rufes muss der Eigentuemer des Files oder der Superuser sein.

**SIEHE AUCH**

stat(2)

**ASSEMBLER**

```
(UTIME:=30)
(file in r0)
(timep in r1)
sc #UTIME
```



---

**NAME**

wait - Warten auf die Beendigung eines Prozesses

**UEBERSICHT**

```
wait(status)
int *status;

wait(0)
```

**BESCHREIBUNG**

Wait veranlasst eine Verzoegerung des rufenden Prozesses bis ein Signal empfangen wird oder einer seiner Child-Prozesse endet. Ist irgendein Child-Prozess seit dem letzten wait beendet, erfolgt eine sofortige Rueckkehr. Gibt es keine Child-Prozesse, erfolgt die Rueckkehr sofort mit gesetztem Fehlerbit (bzw. mit dem zurueck-

gegebenen Wert -1). Die normale Rueckkehr liefert die Prozessidentifikation des beendeten Child- Prozesses. Falls mehrere Child-Prozesse aktiv sind, werden mehrere wait Rufe benoetigt, um das Ende aller Child-Prozesse zu ermitteln.

Ist status ungleich Null, empfaengt das hoehere Byte des Wortes, auf das gezeigt wird, das niedere Byte des Argumentes von exit(2), des beendeten Child-Prozesses. Das niedere Byte empfaengt den Beendigungsstatus des Prozesses. Siehe signal(2) fuer die Liste der Beendigungsstati (Signale); der Nullstatus zeigt normale Beendigung an. Der spezielle Status (0177) wird fuer einen gestoppten Prozess zurueckgegeben, der nicht beendet ist und wieder gestartet werden kann. Siehe ptrace(2). Ist das 0200 Bit des Beendigungsstatusses gesetzt, wurde ein Speicherabbild des Prozesses durch das System erzeugt.

Endet der Parent-Prozess, ohne auf seine Child-Prozesse zu warten, erbt der Initialisierungsprozess (Prozessidentifikator = 1) die Child-Prozesse.

**SIEHE AUCH**

exit(2), fork(2), signal(2)

**DIAGNOSTIK**

-1 wird zurueckgegeben, wenn es keine Child-Prozesse gibt, auf die zuvor nicht gewartet wurde.

**ASSEMBLER**

```
(WAIT:=7)
sc #WAIT
(Prozessidentifikation in r4)
(Status in r5)
```

Das hoehere Byte des Status ist das niedere Byte von r2 in dem Child-Prozess bei dessen Beendigung.

WRITE(2)

---

**NAME**

write - Schreiben auf ein File

**UEBERSICHT**

```
write(fildes, buffer, nbytes)
char *buffer;
```

**BESCHREIBUNG**

Ein Filedeskriptor fildes ist ein Wort, das von einem erfolgreichen open, creat, dup oder pipe(2)- Ruf zurueckgegeben wird.

Buffer ist die Adresse von nbytes zusammenhaengenden Bytes, die auf das Ausgabefile geschrieben werden. Die Zahl der aktuell geschriebenen Zeichen wird als Integerwert zurueckgegeben.
Differenzen zwischen beiden Werten signalisieren Fehler. Am effektivsten ist write bei Schreiben in Bloecken von 512 Byte.

**SIEHE AUCH**

creat(2), open(2), pipe(2)

**DIAGNOSTIK**

Rueckgabe von -1 bei Fehlern: falscher Deskriptor, Pufferadresse oder Bytezaehler; physischer E/A-Fehler.

**ASSEMBLER**

```
(WRITE:=4)
(fildes in r0)
(buffer in r1)
(nbytes in r2)
sc #WRITE
(Bytezaehler in r4)
```

INTRO(3) INTRO(3)

## NAME

intro - Einfuehrung in die Bibliotheksfunktionen

## UEBERSCHRIFT

```
#include<stdio.h>

#include<math.h>
```

## BESCHREIBUNG

Dieser Teil des Systemhandbuches MUTOS8000 beschreibt in verschiedenen Bibliotheken befindliche Funktionen, die im Gegensatz zu den Systemrufen nicht Bestandteil des MUTOS-Kerns sind.

Die Funktionen sind durch eine Kapitelnummer gekennzeichnet:

(3) Diese und die mit (3S) gekennzeichneten Funktionen bilden zusammen die Bibliothek libc. Sie wird automatisch durch den C-Compiler cc(1) geladen. Der Link-Editor ld(1) durchsucht diese Bibliothek bei der '-lc'-Option. Aus den include-Files, die auf den entsprechenden Seiten angegeben sind, kann man die Deklarationen einiger dieser Funktionen erhalten.
Darueberhinaus beinhaltet libc auch Systemrufe.

(3M) Diese Funktionen bilden die mathematische Bibliothek libm. Sie wird, falls benoetigt, automatisch geladen. Der Link-Editor durchsucht diese Bibliothek bei der '-lm'-Option. Deklarationen fuer diese Funktionen kann man aus dem include-File <math.h> erhalten.

(3S) Diese Funktionen bilden das Standard-E/A-Paket (standard I/O package), siehe stdio(3). Diese Funktionen wurden bereits in der Bibliothek libc erwaehnt. Deklarationen fuer diese Funktionen kann man aus dem include-File <stdio.h> erhalten.
Darueberhinaus koennen weitere Funktionen in beliebigen anderen Bibliotheken abgelegt werden.

**FILES**

/lib/libc.a
/lib/libm.a, /usr/lib/libm.a (die eine oder die andere)

**SIEHE AUCH**

stdio(3), nm(1), ld(1), cc(1), intro(2)

**FEHLERBEHANDLUNG**

Funktionen der mathematischen Bibliothek (3M) koennen herkoemmliche Werte zurueckgeben, auch wenn die Funktion fuer die angegebenen Argumente nicht definiert oder der Wert nicht darstellbar ist. In diesen Faellen wird die externe Variable errno (siehe intro(2)) auf den Wert EDOM oder ERANGE gesetzt. Die Werte EDOM und ERANGE werden im include-File <math.h> definiert.

**ASSEMBLER**

In der Assemblersprache kann auf diese Funktion zugegriffen werden, indem man die Aufruffolge, die sonst vom C-Compiler erzeugt wird, selbst notiert. Zum Beispiel kann ecvt(3) auf folgende Weise aufgerufen werden:

```
EXTERN

- ecvt  procedure
        .
        .
        .
  push  @r15, sig
  push  @r15, decpt
  push  @r15, ndigit
  push  @r15, value+4

  call  - ecvt
  inc   r15,#14
```

**ABORT(3)** **ABORT(3)**

---

**NAME**

abort - Erzeugen eines Fehlers durch IOT - Trap

**BESCHREIBUNG**

Abort fuehrt zur Generierung eines IOT - Trap. Durch diesen Befehl wird ein Signal erzeugt, das normalerweise zur Beendigung des Prozesses mit einem Speicherabzug (Core -Dump) fuehrt. Dieser Speicherabzug kann dann zu Testzwecken benutzt werden.

**SIEHE AUCH**
adb(1), signal(2), exit(2)

**FEHLERBEHANDLUNG**
Durch Shell erfolgt die Meldung 'IOT trap - core dumped'.

**ABS(3)** **ABS(3)**

---

**NAME**
**abs** - Absoluter Betrag ganzer Zahlen

**UEBERSCHRIFT**
**abs(i)**

**BESCHREIBUNG**
**Abs** liefert den absoluten Betrag einer ganzen Zahl.

**SIEHE AUCH**
floor(3) fuer fabs

**ASSERT(3X)** **ASSERT(3X)**

---

**NAME**
assert - Pruefung auf Programmabbruch

**UEBERSCHRIFT**
**#include <assert.h>**

**assert (expression)**

**BESCHREIBUNG**
Assert ist ein Macro. Er verursacht ein exit(2) mit einem Fehlerkommentar, falls expression falsch, d.h. gleich Null ist. Ist expression ungleich Null, erfolgt keine Unterbrechung der Programmabarbeitung. Das Compilieren des Programms mit der cc(1) -Option **-DNDEBUG** loescht das assert aus dem Programm.

**FEHLERBEHANDLUNG**
Man erhaelt die Ausschrift:
'Assertion failed: file f line n.'
F ist das Quellfile und n die Quellzeile des assert - statements.

**NAME**

atof, atoi, atol - Konvertiert ISO-Zeichen in Zahlen

**UEBERSCHRIFT**

```
double atof(nptr)
char *nptr;

atoi(nptr)
char *nptr;

long atol(nptr)
char *nptr;
```

**BESCHREIBUNG**

Diese Funktionen wandeln ihre entsprechenden Zahlen, die durch Zeichenketten gegeben sind, in Float-, Integer- bzw. Longintegerzahlen um. Der Zeiger nptr verweist auf diese Zeichenkette. Die Zeichenkette wird durch das erste nicht zu einer Zahlendarstellung gehoerende Zeichen beendet (z.B. 123*).

Atof erkennt eine beliebige Folge von Tabulatoren und Leerzeichen, ein eventuelles Vorzeichen, eine Folge von Ziffern, die einen Dezimalpunkt beinhalten kann, sowie ein moegliches 'e' oder 'E' fuer die Darstellung des Exponenten, gegebenenfalls von einer vorzeichenbehafteten Integerzahl gefolgt.

Atoi und atol erkennen eventuelle Tabulatoren und Leerzeichen, gegebenenfalls ein Vorzeichen, sowie eine Folge von Ziffern.

**SIEHE AUCH**

scanf(3)

**FEHLERHINWEIS**

Es gibt keine Vorsichtsmassnahmen gegen overflows.

---

**NAME**

**crypt, setkey, encrypt**
- Verschluesselung und Entschluesselung

**UEBERSCHRIFT**

```
char *crypt(key, salt)
char *key, *salt;

setkey(key)
char *key;

encrypt(block, edflag)
char *block;
```

**BESCHREIBUNG**

**Crypt** ist die Passwortentschluesselungsroutine.

Das erste Argument von **crypt** ist das vom Nutzer geschriebene Passwort. Das zweite ist eine 2-zeichenlange Zeichenkette, ausgewaehlt aus dem Zeichenvorrat [a-zA-Z0-9./]. Die salt Zeichenkette wird benutzt, um den Algorithmus auf einem von 4096 verschiedenen Wegen zu permutieren. Danach wird das Passwort benutzt, um eine konstante Zeichenkette mehrmals zu verschluesseln. Das verschluesselte Passwort besteht aus Zeichen desselben Alphabetes wie salt. Die ersten zwei Zeichen sind salt selbst.

Die anderen Funktionen liefern einen (wenn auch primitiven) Zugang zum aktuellen Verschluesselungsalgorithmus. Das Argument von **setkey** ist ein Zeichenfeld der Laenge 64, das nur Zeichen mit dem numerischen Wert 0 oder 1 beinhaltet. Diese Feld wird in Gruppen zu je 8 aufgeteilt und das niederwertigste Bit in jeder Gruppe ignoriert. Das fuehrt zu einem 56 Bit langen Schluessel, der gespeichert wird.

Das Argument von **encrypt** ist ebenfalls ein Zeichenfeld der Laenge 64, das nur Nullen und Einsen enthaelt. Dieses Feld wird am Platz modifiziert. Es entsteht ein Feld, das die Bits des Argumentes nach Anwendung des Algorithmus enthaelt. Der Algorithmus verwendet dabei den mit **setkey** erzeugten Schluessel. Ist edflag 0, wird das Argument verschluesselt. Ist es von 0 verschieden, wird das Argument entschluesselt.

**SIEHE AUCH**

passwd(1), passwd(5), login(1), getpass(3)

**FEHLERHINWEISE**

Die Zeiger auf den Inhalt von Staticvariablen werden bei jedem Aufruf ueberschrieben.

---

**NAME**

ctime, localtime, gmtime, asctime, timezone
Konvertierung von Datum und Uhrzeit in und aus ISO

**UEBERSCHRIFT**

```
char *ctime(clock)
long *clock;

#include <time.h>

struct tm *localtime(clock)
long *clock;

struct tm *gmtime(clock)
long *clock;

char *asctime(tm)
struct tm *tm;

char *timezone(zone, dst)
```

**BESCHREIBUNG**

Ctime konvertiert eine Zeit clock, so wie sie durch time(2) erzeugt wird und liefert einen Zeiger auf eine 26-Zeichen-lange Zeichenkette der folgenden Form. Saemtliche Felder haben konstante Laenge.

```
Sun Sep 16 01:03:52 1973\n\0
```

Localtime und gmtime liefern Zeiger auf Strukturen, die eine aufgespaltene Zeitdarstellung enthalten. Localtime korrigiert die Zeitangabe entsprechend der Zeitzone und der eventuellen Sommerzeit, gmtime konvertiert direkt in GMT, die Zeit, die MUTOS verwendet. Asctime konvertiert eine Zeitangabe in ISO und liefert einen Zeiger auf eine 26-Zeichen-lange Zeichenkette.

Die Strukturdeklaration vom Includefile ist:

```
struct tm { /* see ctime(3) */
        int   tm_sec;
        int   tm_min;
        int   tm_hour;
        int   tm_mday;
        int   tm_mon;
        int   tm_year;
        int   tm_wday;
        int   tm_yday;
        int   tm_isdst;
};
```

Diese Werte geben die Uhrzeit ( 24h ), den Tag des Monats (1-31), den Monat, den Wochentag (Sonntag=0), das Jahr (ab 1900), den Tag des Jahres (0-365) und ein Flag an, das von 0 verschieden ist, falls gerade Sommerzeit ist.

Timezone liefert den Namen der Zeitzone, die dem ersten Argument entspricht, das in Minuten westlich von Greenwich gemessen wird. Ist das zweite Arguments 0, wird der Standardname verwendet, andernfalls die Sommerzeitvariante. Falls sich zu der angegebenen Zeitzone kein Name in der zugehoerigen Tabelle in timezone() befindet, wird das Resultat nur als Differenz zur GMT ausgegeben, beispielsweise fuer Afghanistan liefert timezone(-(60*4+30),0) die Zeichenkette **GMT+4:30**, weil die Differenz zwischen der Ortszeit von Afghanistan und GMT 4:30 betraegt.

**SIEHE AUCH**

time(2)

**FEHLERBEHANDLUNG**

Die Zeiger auf den Inhalt von Staticvariablen werden bei jedem Aufruf ueberschrieben.

**CTYPE(3)** **CTYPE(3)**

---

**NAME**

**isalpha, isupper, islower, isdigit, isalnum, isspace, ispunct, isprint, iscntrl, isascii** - Zeichenklassifizierung

**UEBERSCHRIFT**

```
#include <ctype.h>

isalpha(c)

. . .
```

**BESCHREIBUNG**

Diese Makros klassifizieren ISO-kodierte Integerwerte, mit Hilfe des Durchsuchens einer Tabelle. Jedes dieser Makros liefert im Fall true (wahr) einen Wert ungleich Null und im Fall false (falsch) den Wert Null.

| | |
|---|---|
| isalpha | c ist ein Buchstabe. |
| isupper | c ist ein Grossbuchstabe. |
| islower | c ist ein Kleinbuchstabe. |
| isdigit | c ist eine Ziffer. |
| isalnum | c ist ein alphanumerisches Zeichen. |

isspace — c ist ein Leerzeichen, ein Tabulator, ein Wagenruecklauf, ein Newline oder ein Seitenvorschub.

ispunct — c ist ein Interpunktionzeichen (weder Steuer- noch Alphazeichen).

isprint — c ist ein druckbares Zeichen, vom Code 040 (Leerzeichen) bis 0176 ('~').

iscntrl — c ist das Loeschzeichen (0177) oder ein allgemeines Steuerzeichen (< 040).

isascii — c ist ein ISO-Zeichen (< 0200).

**SIEHE AUCH**

ascii(7)

**ECVT(3)** **ECVT(3)**

---

**NAME**

ecvt, fcvt, gcvt - Konvertieren Gleitkommazahlen in ISO-Zahlen

**UEBERSCHRIFT**

```
char *ecvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt,

char *fcvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *gcvt(value, ndigit, buf)
double value;
char *buf;
```

**BESCHREIBUNG**

Ecvt konvertiert value zu einer mit '0' abgeschlossenen Kette von ndigit ISO-Dezimalen und gibt einen Zeiger auf diese Zeichenkette zurueck. Durch **decpt** wird die Anzahl der Dezimalstellen vor (bei negativem Wert) bzw. nach dem Dezimalpunkt festgelegt. Das Ergebnis ist positiv, wenn **sign** den Wert Null enthaelt, ansonsten ist es negativ. Die letzte Dezimalstelle wird gerundet.

Ecvt ist identisch mit **ecvt** , wobei der exakte Wert auf die durch ndigit spezifizierte Anzahl von Dezimalstellen gerundet wird (entsprechend Fortran F-Ausgabeformat).

Gcvt konvertiert value zu einer mit '0' abgeschlossenen ISO-Zeichenkette in buf und liefert einen Zeiger auf buf. Gvct versucht moeglichst ndigit signifikante Ziffern im Fortran F- Format zu erzeugen, ansonsten wird E-Format erzeugt. Nachfolgende Nullen bei den Dezimalstellen koennen weggelassen werden.

**SIEHE AUCH**

printf(3)

**FEHLERQUELLEN**

Die Zeiger auf den Inhalt von Staticvariablen werden bei jedem Aufruf ueberschrieben.



---

**NAME**

end, etext, edata - Letzte Speicherplaetze im Programm

**UEBERSCHRIFT**

```
extern end;
extern etext;
extern edata;
```

**BESCHREIBUNG**

Diese Namen verweisen auf Speicherplaetze. Die Adresse von etext ist die erste Adresse nach dem Textsegment des Programmes, edata die erste Adresse nach dem Bereich der initialisierten Daten und end die erste Adresse nach den nichtinitialisierten Daten und somit dem Programm ueberhaupt.

Zu Beginn der Ausfuehrung eines Programms stimmt das Programmende mit end ueberein. Das Programmende kann durch viele Funktionen, so zum Beispiel durch die Routinen brk (2) und malloc (3), veraendert werden. Die Funktion sbrk (0) liefert eine Adresse, die dem aktuellen Programmende entspricht.

**SIEHE AUCH**

brk(2), malloc(3)

**NAME**

**exp, log, log10, pow, sqrt**
Logarithmus, Exponential- und Potenzfunktion, Quadratwurzel

**UEBERSCHRIFT**

```
#include <math.h>

double exp(x)
double x;

double log(x)
double x;

double log10(x)
double x;

double pow(x, n)
double x, n;

double sqrt(x)
double x;
```

**BESCHREIBUNG**

**xp** liefert den Wert der Exponentialfunktion von x, d.h. $exp(x) = e^x$.

**log** berechnet den natuerlichen Logarithmus von x, $ln(x)$.

**log10** berechnet den Logarithmus von x zur Basis 10, $lg(x)$.

**pow** berechnet die n.-Potenz von x, $x^n$. pow

**sqrt** berechnet die Quadratwurzel aus x.

**SIEHE AUCH**

hypot(3), sinh(3), intro(2)

**FEHLERBEHANDLUNG**

**Exp** und **pow** geben eine "unendlich" grosse Zahl zurueck, wenn ihr darstellbarer Wert ueberschritten wird (overflow). errno wird auf ERANGE gesetzt.

**Pow** uebergibt 0 und setzt errno auf EDOM, wenn das zweite Argument keine natuerliche Zahl ist oder wenn beide Argumente 0 sind.
**Log** gibt 0 zurueck, wenn x kleiner oder gleich Null ist. errno wird EDOM gesetzt.
**Sqrt** gibt eine 0 zurueck, wenn x negativ ist; errno wird EDOM gesetzt.

---

**NAME**

fclose, fflush - Schliessen oder Entleeren eines Files (gepufferte Ausgabe)

**UEBERSICHT**

```
#include <stdio.h>

fclose(filep)
FILE *filep;

fflush(filep)
FILE *filep;
```

**BESCHREIBUNG**

Fclose veranlasst, dass alle Puffer, die Daten zur Ausgabe in das durch filep gekennzeichnete File beinhalten, geleert werden und das File geschlossen wird. Puffer, die vom Standard E/A-System zugewiesen wurden, werden freigegeben.

Ein Aufruf von exit(2) beinhaltet die Ausfuehrung von fclose (3).
Fflush veranlasst, dass alle fuer die Ausgabe gepufferten Daten in das durch filep gekennzeichnete File geschrieben werden. Das File bleibt geoeffnet.

**SIEHE AUCH**

close(2), fopen(3), setbuf(3)

**MERKMALE**

Als Rueckkehrwert liefern diese Routinen **EOF**, filep nicht einem Ausgabe-File zugeordnet ist oder die gepufferten Daten nicht zu diesem File uebertragen werden koennen.

NAME

feof, ferror, clearerr, fileno - File-Statusabfragen

UEBERSICHT

```
#include <stdio.h>

feof(filep)
FILE *filep;

ferror(filep)
FILE *filep;

clearerr(filep)
FILE *filep;

fileno(filep)
FILE *filep;
```

BESCHREIBUNG

Feof hat einen Rueckkehrwert ungleich NULL, wenn bei einem bereits erfolgten Lesen des durch filep gekennzeichneten Files das Ende des Files (EOF) erkannt wurde. Wurde kein EOF gelesen, so ist der Wert gleich NULL.

Ferror liefert einen Wert ungleich Null, falls sich waehrend des Lesens bzw. Schreibens des angegebenen files ein Fehler ereignet hat, sonst Null. Wird das gesetzte Fehlermerkmal nicht mit clearerr zurueckgesetzt, so bleibt es bis zum Schliessen des Files bestehen. Nachfolgende Aufrufe von ferror liefern immer einen Wert ungleich Null.

Clrerr setzt das Fehlermerkmal des zu filep gehoerigen Files zurueck.

Fileno hat als Rueckkehrwert den (integer) Filedescriptor, der filep zugeordnet ist; siehe open(2).

Diese vier Funktionen sind als Makros in <stdio.h> implementiert. Sie koennen nicht neu vereinbart werden.

SIEHE AUCH

fopen(3), open(2)

**NAME**

fabs, floor, ceil - Absoluter Betrag, ganzzahliger Anteil

**UEBERSCHRIFT**

```
#include <math.h>

double floor(x)
double x;

double ceil(x)
double x;

double fabs(x)
double(x);
```

**BESCHREIBUNG**

**fabs** liefert den absoluten Betrag |x|.

**floor** liefert die groesste ganze Zahl, die nicht groesser als x ist.

**ceil** liefert die kleinste ganze Zahl,die groesser oder gleich x ist.

**SIEHE AUCH**

abs(3)



**NAME**

fopen, freopen, fdopen - Oeffnen eines Files zur gepufferten Ein-/Ausgabe

**UEBERSICHT**

```
#include <stdio.h>

FILE *fopen(filename, type)
char *filename, *type;

FILE *freopen(filename, type, filep)
char *filename, *type;
FILE *filep;

FILE *fdopen(fildes, type)
char *type;
```

**BESCHREIBUNG**

Fopen oeffnet das File mit dem Namen filename und stellt als Rueckkehrwert einen Filepointer filep (Zeiger auf die Struktur FILE) zur Verfuegung, der in darauffolgenden Operationen zur Identifizierung des Files benutzt wird.

Type steht fuer eine Zeichenkette, deren erstes Zeichen folgende Werte haben kann:

"r" read - Oeffnen zum Lesen

"w" write - Oeffnen zum Schreiben

"a" append - Oeffnen zum Schreiben ab File-Ende oder Oeffnen zum Schreiben, falls das File filename noch nicht existiert.

Ist das zweite Zeichen der Zeichenkette type ein '+', so wird das File sowohl zum Lesen als auch zum Schreiben geoeffnet.

"r+" oeffnet das schon existierende File zum Lesen und Schreiben, wobei der Lese/Schreib-Pointer auf File-Anfang gesetzt wird.

"w+" erzeugt das File filename und oeffnet es zum Lesen und Schreiben.

"a+" oeffnet das bereits existierende File zum Lesen und Schreiben und setzt den Pointer auf das File-Ende.

Freopen ordnet den Filepointer filep eines geoeffneten Files dem angegebenen File filename zu.
Der urspruengliche Filepointer filep des bis zum Zeitpunkt des Aufrufens dieser Funktion geoeffneten Files ist der Rueckkehrwert dieser Funktion und wird dem neuen File filename zugeordnet. Das urspruengliche File wird geschlossen.

Eine typische Anwendung von freopen liegt darin, die standardmaessig geoeffneten Files stdin, stdout und stderr speziellen Files zuzuweisen.

Fdopen liefert einen Filepointer filep und stellt die Verbindung zwischen einem File und einem Filedeskriptor her, der von einem der Systemrufe creat, dup,open oder pipe(2) geliefert wurde. Dabei muss der type des File mit dem Modus des geoeffneten Files uebereinstimmen.

**SIEHE AUCH**

open(2), fclose(3)

**MERKMALE**

Fopen und freopen liefern den Pointer **NULL** als Rueckkehrwert, falls auf filename nicht zugegriffen werden kann.

**FEHLERQUELLEN**

Fdopen ist nicht portabel zu anderen Betriebssystemen.

**FREAD(3S)** **FREAD(3S)**

---

**NAME**

fread, fwrite - Gepufferte binaere Ein-/Ausgabe

**UEBERSICHT**

```
#include <stdio.h>

fread(ptr, sizeof(*ptr), nitems, filep)
FILE *filep;

fwrite(ptr, sizeof(*ptr), nitems, filep)
FILE *filep;
```

**BESCHREIBUNG**

Fread liest von dem durch filep gekennzeichneten Eingabefile nitems Daten des Typs *ptr in einen Block, der bei ptr beginnt.
Rueckkehrwert ist die tatsaechlich eingelesene Anzahl von Daten.

Fwrite fuegt an das durch filep gekennzeichnete File nitems Daten des Typs *ptr an.
Die Daten werden aus einem Block bereitgestellt, der bei ptr beginnt.
Rueckkehrwert ist die tatsaechliche Anzahl der geschriebenen Daten.

**SIEHE AUCH**

read(2), write(2), fopen(3), getc(3), putc(3), gets(3), puts(3), printf(3), scanf(3)

**MERKMALE**

Fread und fwrite haben 0 als Rueckkehrwert bei festgestelltem End-of-File oder Fehler.

---

**NAME**

**frexp, ldexp, modf**
- Aufspalten einer Gleitkommazahl in Mantisse und Exponent

**UEBERSCHRIFT**

```
double frexp(value, eptr)
double value;
int *eptr;

double ldexp(value, exp)
double value;

double modf(value, iptr)
double value, *iptr;
```

**BESCHREIBUNG**

**frexp** liefert die Mantisse einer doppelt genauen Zahl *value* als doppelt genauen Wert *x*, mit $x < 1$ und speichert indirekt in *eptr* eine Integerzahl *n* so dass

$$value = x * 2^n$$

**ldexp** liefert den Wert $value * 2^{exp}$. ldexp

**modf** liefert den gebrochenen Anteil von *value* und speichert den ganzzahligen Anteil indirekt in *iptr*.



---

**NAME**

**fseek, ftell, rewind** - Positionieren innerhalb eines Files

**UEBERSICHT**

```
#include <stdio.h>

fseek(filep, offset, ptrname)
FILE *filep;
long offset;

long ftell(filep)
FILE *filep;

rewind(filep)
```

**BESCHREIBUNG**

Fseek bestimmt die Position innerhalb des Files, ab der nachfolgende Ein- oder Ausgabe-Operationen auf das File, das durch filep gekennzeichnet wird, realisiert werden. Die neue Position ist offset Bytes vom Anfang, der aktuellen Position oder dem Ende des Files entfernt, abhaengig davon, ob ptrname den Wert 0, 1 oder 2 hat.

Fseek macht die Wirkungen von ungetc(3) rueckgaengig.

Ftell liefert als Rueckkehrwert den aktuellen Offset relativ zum Anfang des Files, auf das sich filep bezieht. Der Offset wird in Bytes angegeben.

Rewind(filep) hat die gleiche Wirkung wie fseek(filep, 0L, 0).

**SIEHE AUCH**

lseek(2), fopen(3)

**MERKMALE**

Fseek hat -1 als Rueckkehrwert fuer nichtrealisierbare Positionierungen.

**GETC(3S)** **GETC(3S)**

---

**NAME**

getc, getchar, fgetc, getw - Empfangen eines Zeichens oder Wortes aus einem FILE

**UEBERSICHT**

```
#include <stdio.h>

int getc(filep)
FILE *filep;

int getchar()

int fgetc(filep)
FILE *filep;

int getw(filep)
FILE *filep;
```

**BESCHREIBUNG**

Getc stellt das naechste Zeichen aus dem angegebenen File (gekennzeichet durch filep) bereit.

Getchar() ist identisch mit getc(stdin), d.h., das Zeichen wird aus dem Standard-Input-File bereitgestellt.

fgetc realisiert das gleiche wie getc, ist aber als Funktion und nicht als Makro implementiert. Bei Benutzung von fgetc wird ein kuerzerer Objektcode bei der Implementierung erreicht.

Getw stellt das naechste Wort aus dem angegebenen File (gekennzeichnet durch filep) bereit. Bei Erreichen von End-of-File oder bei einem Fehler ist dieser Rueckkehrwert die Konstante **EOF**. Da EOF aber auch als Integerwert innerhalb eines Files auftreten kann, ist es noetig, feof und ferror(3) zu benutzen, um den Erfolg einer getw-Operation zu ueberpruefen. Getw verlangt keine spezielle Strukturierung innerhalb des Files.

**SIEHE AUCH**

fopen(3), putc(3), gets(3), scanf(3), fread(3), ungetc(3)

**MERKMALE**

Diese Funktionen liefern **EOF** bei Erreichen von End-of-File oder beim Auftreten eines Lesefehlers.

Ein Abbruch mit der Meldung 'Reading bad file' erfolgt, wenn von einem File gelesen werden soll, das nicht zum Lesen geoeffnet wurde.

**FEHLERQUELLEN**

Da getc als Makro implementiert ist, koennen filep-Argumente, die sich aus Ausdruecken ergeben, falsch behandelt werden. So arbeitet zum Beispiel 'getc(*f++);' nicht wie erhofft.

**GETENV(3)** **GETENV(3)**

---

**NAME**

getenv - Wert eines Environment-Namens

**UEBERSICHT**

```
char *getenv(name)
char *name;
```

**BESCHREIBUNG**

Getenv durchsucht die Environment-Liste (siehe environ(5)), deren Elemente in der Form name = value eingetragen sind, nach der Teilzeichenkette name und liefert value als Rueckkehrwert, falls die Teilzeichenkette vorhanden ist. Andernfalls wird ein Nullzeiger zurueckgegeben.

**SIEHE AUCH**

environ(5), exec(2)

---

**NAME**

getgrent, getgrgid, getgrnam, setgrent, endgrent - Bereitstellen von Eintraegen im Gruppenfile

**UEBERSCHRIFT**

```
#include <grp.h>

struct group *getgrent();

struct group *getgrgid(gid);
int gid;

struct group *getgrnam(name)

int setgrent();

int endgrent();
```

**BESCHREIBUNG**

Getgrent, getgrgid und getgrnam stellen jeweils als Rueckkehrwert einen Zeiger zu einem Objekt vom Typ struct_group zur Verfuegung. Diese Struktur entspricht einer Zeile (Eintragung) im Gruppenfile:

```
struct    group {
     char *gr_name;
     char *gr_passwd;
     int  gr_gid;
     char **gr_mem;
};
```

Die einzelnen Bestandteile dieser Struktur sind:

gr_name - Der Name der Gruppe.

gr_passwd - Das verschluesselte Passwort der Gruppe.

gr_gid - Der numerische Gruppenidentifikator.

gr_mem - Ein mit einem Null-Zeiger abgeschlossenes Feld von Zeigern, die auf die Namen der Gruppenmitglieder verweisen.

Getgrent liest einfach den naechsten Eintrag im group-File. Getgrgid und getgrnam suchen nach einem bestimmten Gruppenidentifikator (gid) bzw. einem bestimmten Namen (name), bis dieser gefunden oder Fileende erreicht ist. Jede dieser Routinen setzt bei dem Eintrag fort, bei dem die vorangegangene aufgehoert hat, so dass durch aufeinanderfolgende Rufe das ganze File durchsucht werden kann.

Der Ruf von setgrent bewirkt das Zuruecksetzen auf den Fileanfang, um wiederholtes Suchen zu ermoeglichen. Endgrent kann zum Schliessen des group-Files am Ende der Arbeit benutzt werden.

**FILES**

/etc/group

**SIEHE AUCH**

getlogin(3), getpwent(3), group(5)

**MERKMALE**

Ein Nullzeiger (0) wird zurueckgegeben, falls das Ende des group-Files erreicht (EOF) oder ein Fehler erkannt wurde.

**FEHLERQUELLEN**

Alle Informationen werden nur statisch abgespeichert, muessen also kopiert werden, wenn sie ueber mehrere Aufrufe erhalten bleiben sollen.

**GETLOGIN(3)** **GETLOGIN(3)**

---

**NAME**

getlogin - Ermitteln des Login-Namen

**UEBERSCHRIFT**

```
char *getlogin();
```

**BESCHREIBUNG**

Getlogin ist sinnvoll bei Multiuserbetrieb - bei einem existierenden File /etc/utmp.

Getlogin liefert als Rueckkehrwert einen Zeiger zum Login-Namen, der in /etc/utmp gespeichert ist. Diese Routine kann zusammen mit getpwnam benutzt werden, um den gueltigen dazugehoerigen Passwort-File-Eintrag zu ermitteln, wenn ein Nutzeridentifikator fuer verschiedene Login-Namen benutzt wird.

Wenn getlogin in einem Prozess aufgerufen wird, der keinem Terminal zugeordnet ist, ist der Rueckkehrwert Null. Die richtige Benutztung der Prozeduren zur Ermittlung des Login-Namen waere, zunaechst getlogin aufzurufen, und falls dies scheitert, als naechstes die Prozedur getpwuid zu benutzen.

**FILES**
/etc/utmp

**SIEHE AUCH**
getpwent(3), getgrent(3), utmp(5)

**Merkmale**
Beim Rueckkehrwert Null wurde der Login-Name nicht gefunden.

**FEHLERQUELLEN**
Alle Informationen werden nur statisch abgespeichert und muessen gerettet werden, wenn sie ueber mehrere Aufrufe erhalten bleiben sollen.

**GETPASS(3)** **GETPASS(3)**

---

**NAME**
getpass - Einlesen eines Passwortes

**UEBERSCHRIFT**
```
char *getpass(prompt)
char *prompt;
```

**BESCHREIBUNG**
Getpass liest ein Passwort vom File /dev/tty, oder, wenn dieses nicht geoeffnet werden kann, vom Standardinput. Zuvor wird die Zeichenkette prompt ausgegeben. Bei der Eingabe des Passwortes ist die Echofunktion nicht wirksam. Rueckkehrwert der Funktion ist ein Zeiger zu einer Zeichenkette, die die ersten 8 Zeichen des eingegebenen Passwortes enhaelt und mit Null begrenzt ist.

**FILES**
/dev/tty

**SIEHE AUCH**
crypt(3)

**FEHLERHINWEISE**
Der Rueckkehrwert verweist auf eine "static" - Variable, deren Inhalt bei jedem Aufruf ueberschrieben wird.

---

**NAME**

getpw - Anfordern des Passwort - File - Eintrags zum Nutzeridentifikator (UID)

**UEBERSCHRIFT**

```
getpw(uid, buf)
char *buf;
```

**BESCHREIBUNG**

Getpw sucht im Passwort - File nach dem angegebenen (numerischen) Nutzeridentifikator und uebertraegt die zu diesem Identifikator gehoerige Zeile in buf. Wird der Nutzeridentifkator nicht gefunden, ist der Rueckkehrwert ungleich Null. Der uebergebene Passwort-File-Eintrag ist mit Null abgeschlossen.

**FILES**

/etc/passwd

**SIEHE AUCH**

getpwent(3), passwd(5)

**FEHLERBEHANDLUNG**

Der Rueckkehrwert ist ungleich Null bei Fehler.



---

**NAME**

getpwent, getpwuid, getpwnam, setpwent, endpwent - Bereitstellen eines Eintrags im Passwort-File

**UEBERSCHRIFT**

```
#include <pwd.h>

struct passwd *getpwent();

struct passwd *getpwuid(uid) int uid;

struct passwd *getpwnam(name) char *name;

int setpwent();

int endpwent();
```

**BESCHREIBUNG**

Getpwent, getpwuid und getpwnam stellen jeweils als Rueckkehrwert einen Zeiger zu einem Objekt vom Typ struct passwd bereit. Diese Struktur entspricht dem Inhalt einer Zeile aus dem Passwort-File:

```
struct    passwd {
     char *pw_name;
     char *pw_passwd;
     int  pw_uid;
     int  pw_gid;
     int  pw_quota;
     char *pw_comment;
     char *pw_gecos;
     char *pw_dir;
     char *pw_shell;
};
```

Die Strukturelemente pw_quota und pw_comment werden nicht benutzt; die Bedeutung der anderen Felder ist in passwd(5) beschrieben.

Getpwent liest die naechste Zeile (das Oeffnen des File ist vorher noetig); setpwent setzt auf den Anfang zurueck und endpwent schliesst das File.

Getpwuid und getpwnam suchen von Beginn an nach einem uid (Nutzeridentifikator) oder dem Namen name. Die Suche wird am Fileende abgebrochen.

**FILES**

/etc/passwd

**SIEHE AUCH**

getlogin(3), getgrent(3), passwd(5)

**MERKMALE**

Bei Fehler oder Fileende wird ein Zeiger mit dem Wert Null zurueckgegeben.

**FEHLERQUELLEN**

Alle Informationen werden nur statisch abgespeichert und muessen gerettet werden, wenn sie ueber mehrere Aufrufe erhalten bleiben sollen.

---

**NAME**

gets, fgets - Empfangen einer Zeichenkette aus einem File

**UEBERSICHT**

```
#include <stdio.h>

char *gets(s)
char *s;

char *fgets(s, n, filep)
char *s;
FILE *stream;
```

**BESCHREIBUNG**

Gets liest eine Zeichenkette aus dem Standard-Eingabe-File **stdin** und speichert sie nach s. Die Zeichenkette muss mit 'newline' abgeschlossen sein. In s wird das 'newline'-Zeichen durch das Null-Zeichen ('\0') ersetzt. Rueckkehrwert von gets ist sein Argument.

Fgets liest n-1 Zeichen aus dem File (gekennzeichnet durch filep) nach s. Befindet sich unter diesen Zeichen ein 'newline'-Zeichen, so wird nur bis einschliesslich dorthin gelesen. In s wird ein NULL-Zeichen an die eingelesene Zeichenkette angehaengt.
Rueckkehrwert der Funktion fgets ist ihr erstes Argument.

**SIEHE AUCH**

puts(3), getc(3), scanf(3), fread(3), ferror(3)

**MERKMALE**

Beim Erreichen von End-of-File oder bei einem Fehler liefern gets und fgets den konstanten Zeiger **NULL** als Rueckkehrwert.

**FEHLERQUELLEN**

Gets loescht das 'newline'-Zeichen, fgets belaesst es in der Zeichenkette.

**NAME**

hypot, cabs - Euklidischer Abstand, Betrag komplexer Zahlen

**UEBERSICHT**

```
#include <math.h>

double hypot(x, y)
double x, y;

double cabs(z)
struct { double x, y;} z;
```

**BESCHREIBUNG**

Hypot und cabs berechnen

```
sqrt(x*x + y*y).
```

Man verhindere das Auftreten eines Ueberlaufs durch geeignete Wahl der Argumente.

**SIEHE AUCH**

exp(3) fuer sqrt



**NAME**

j0, j1, jn, y0, y1, yn - Bessel-Funktionen

**UEBERSICHT**

```
#include <math.h>

double j0(x)
double x;

double j1(x)
double x;

double jn(n, x);
double x;

double y0(x)
double x;

double y1(x)
double x;
```

```
double yn(n, x)
double x;
```

**BESCHREIBUNG**

Diese Funktionen berechnen Bessel-Funktionen erster und zweiter Art ganzzahliger Ordnung fuer reelle Argumente.

**FEHLER**

Negative Argumente liefern bei den Funktionen **y0**, **y1** und **yn** eine "unendlich" grosse Zahl, und errno wird EDOM gesetzt.

**L3TOL(3)** **L3TOL(3)**

---

**NAME**

**l3tol, ltol3** -
Konvertiert zwischen 3-Byte Integers und Longintegers

**UEBERSCHRIFT**

```
l3tol(lp, cp, n)
long *lp;
char *cp;

ltol3(cp, lp, n)
char *cp;
long *lp;
```

**BESCHREIBUNG**

**L3tol** konvertiert eine Folge von n 3-Byte-Integers, die sich in einer Zeichenkette befinden, auf die cp zeigt, in eine Folge von Long-Integers, auf die lp zeigt.

**Ltol3** konveriert entgegengesetzt, von Long-Integers (lp) in 3-Byte-Integers (cp).

Diese Funktionen koennen zur Aktualisierung von File-systemen verwendet werden, Plattenadressen sind 3 Bytes lang.

**SIEHE AUCH**

filsys(5)

---

**NAME**

malloc, free, realloc, calloc - Routinen zur Hauptspeicherzuordnung

**UEBERSCHRIFT**

```
char *malloc(size)
unsigned size;

free(ptr)
char *ptr;

char *realloc(ptr, size)
char *ptr;
unsigned size;

char *calloc(nelem, elsize)
unsigned nelem, elsize;
```

**BESCHREIBUNG**

Malloc und free sind zwei einfache Funktionen zur Hauptspeicherzuordung fuer ein Programm. Malloc hat als Rueckkehrwert einen Zeiger zu einem Speicherbereich mit der Laenge von size Byte, beginnend an einer Wortgrenze.

Das Argument von free ist ein Zeiger, der vorher durch malloc zugeordnet wurde. Dieser Speicherbereich wird fuer erneute Speicherzuordnug zur Verfuegung gestellt. Der bisherige Inhalt wird unveraendert hinterlassen.

Malloc ordnet das erste kontinuierliche Speicherstueck, das mindestens der Groesse size entspricht, zu. Dabei wird der vorher freigegebene Speicherplatz zyklisch durchsucht. Steht nicht genuegend Speicherplatz durch vorhergehende Freigaben zur Verfuegung, wird sbrk (siehe break(2)) aufgerufen, um vom System neuen Speicherplatz anzufordern.

Realloc aendert die Groesse des durch ptr angegebenen Blocks auf size und hat als Rueckkehrwert einen Zeiger zum Block neuer Groesse ( eventuell in einem anderen Speicherbereich). Der Inhalt des bestehend bleibenden Teils wird nicht zerstoert. Realloc arbeitet auch so, wenn ptr zu einem Block zeigt, der vorher durch malloc,realloc oder calloc freigegeben wurde, so das free,malloc und realloc die Suchstrategie von malloc beeinflussen koennen, um eine hoehere Kompaktheit zu erreichen.

Calloc ordnet Speicher fuer ein Feld von nelem Elementen der Groesse elsize zu. Der Speicherbereich wird mit Null initialisiert.

Alle diese Routinen haben als Rueckkehrwert einen Zeiger zum entspechenden Speicherbereich.

**FEHLERBEHANDLUNG**

Malloc,realloc und calloc geben Null zurueck, wenn kein Speicher mehr zu Verfuegung steht oder bei anderen Fehlern.

**FEHLERQUELLEN**

Wenn realloc Null zurueckgibt, ist der Block, zu dem ptr zeigt, zerstoert.

**MKTEMP(3)** **MKTEMP(3)**

---

**NAME**

mktemp - Erzeugt einen einmaligen Filenamen

**UEBERSCHRIFT**

```
char *mktemp(template)
char *template;
```

**BESCHREIBUNG**

Mktemp ersetzt das template durch einen einmaligen Filenamen und gibt die Adresse des Templates zurueck. Das Template sollte ein Filename mit sechs nachfolgenden "X" sein. Diese werden dann durch den aktuellen Prozessidentifikator ersetzt.

**SIEHE AUCH**

getpid(2)

**NLIST(3)** **NLIST(3)**

---

**NAME**

nlist - Bereitstellen von Eintraegen aus der Namensliste

**Syntax**

```
#include <a.out.h>
nlist(filename, nl)
char *filename;
struct nlist nl[ ];
```

**BESCHREIBUNG**

Nlist untersucht die Namesliste des ausfuehrbaren Files filename und waehlt daraus die gewuenschten Informationen aus. Die Namensliste besteht aus einem Feld von Strukturen, die den Symbolnamen, den Symboltyp und den zuge-

hoerigen Wert enthalten. Diese Struktur ist im File /usr/indude/a.out.h definiert.
Diese interessierenden Namen sind in die Liste einzutragen und mit dem Wert Null abzuschliessen. Fuer jeden gefundenen Namen werden Typ und Adresse in den naechsten beiden Positionen eingetragen. Wird ein Name nicht gefunden, werden diese Werte auf 0 gesetzt.

**SIEHE AUCH**

a.out(5)

**FEHLERMELDUNGEN**

Ist ein File nicht auffindbar oder fehlt die Namensliste, werden alle zum Namen gehoerigen Variablen auf Null gesetzt.

**PERROR(3)** **PERROR(3)**

---

**NAME**

perror, sys_errlist, sys_nerr - System-Fehlermeldungen

**UEBERSCHRIFT**

```
perror(s)
char *s;

int sys_nerr;
char *sys_errlist[];
```

**BESCHREIBUNG**

Perror schreibt eine Fehlermeldung in die Standardfehlerdatei (standard error file), die den letzten Fehler beschreibt, der waehrend eines Aufrufs des Systems von einem C-Programm aus auftrat. Zuerst wird die Zeichenkette s, die Argument ist, ausgegeben, dann ein Doppelpunkt, dann die Fehlermeldung und ein Newline. Am besten ist es, als Argument den Name des Programms zu verwenden, der den Fehler hervorgerufen hat. Die Fehler-Nummer wird von der externen Variablen errno gelesen (siehe intro(2)). Sie wird gesetzt, wenn ein Fehler auftritt. Diese Variable wird nicht zurueckgesetzt, wenn Aufrufe folgen, die keine Fehler erzeugen.

Um die Formatierung der Meldungen zu vereinfachen, wird ein Vektor der Fehlertexte sys_errlist bereitgestellt. Errno kann als Index verwendet werden, um den zugehoerigen Fehlertext (ohne Newline) aus dieser Liste zu erhalten. Sys_nerr ist die Zahl der Fehlertexte, die in die Tabelle aufgenommen wurden.

Diese Anzahl sollte geprueft werden, da neue Fehlercodes zum System hinzugefuegt worden sein koennen, bevor sie in die Tabelle eingefuegt wurden.

**SIEHE AUCH**

intro(2)

---

**POPEN(3S)** **POPEN(3S)**

---

**NAME**

popen, pclose - E/A von/zu einem Prozess

**UEBERSCHRIFT**

```
FILE *popen(command, type)
char *command, *type;

pclose(filep)
FILE *filep;
```

**BESCHREIBUNG**

Die Argumente von popen sind Zeiger zu mit Null abgeschlossen Zeichenketten, die eine Shell-Kommandozeile und einen E/A- Modus enthalten ("r" fuer Lesen und "w" fuer Schreiben bezogen auf den rufenden Prozess). Popen erzeugt einen neuen Prozess und startet die Ausfuehrung des angegebenen Programms. Es wird ein Pipe zwischen dem rufenden Prozess und dem aktivierten Prozess geschaffen. Der Rueckkehrwert ist ein Filepointer, der zum Schreiben zum Standard-Input oder Lesen vom Standard-Output des aufgerufenen Kommandos benutzt werden kann.

Eine durch popen geoeffnete gepufferte Datei sollte durch pclose geschlossen werden. Pclose wartet auf die Beendigung des ueber die Pipe verbundenen Prozesses und gibt den exit-Status dieses Kommandos zurueck.

Da die geoeffneten Files geteilt werden, kann ein Kommando vom Typ "r" als Input-Filter und eines vom Typ "w" als Output-Filter benutzt werden.

**SIEHE AUCH**

pipe(2), fopen(3), fclose(3), system(3), wait(2)

**FEHLERMELDUNGEN**

Popen hat als Rueckkehrwert 0, wenn der Prozess nicht geoeffnet oder auf Shell nicht zugegriffen werden kann.

Pclose gibt -1 zurueck, wenn filep nicht mit einem durch popen aufgerufenen Kommando verbunden ist.

**FEHLERHINWEISE**

Gepuffertes Lesen vor dem Eroeffnen eines Input-Filters kann dazu fuehren, dass der Standard-Input-Pointer des Filters falsch positioniert wird. Probleme bei Output-Filtern koennen vermieden werden, wenn existierende Puffer vorher mit fflush geleert werden. Siehe fclose(3).

**PRINTF(3S)** **PRINTF(3S)**

---

**NAME**

printf, fprintf, sprintf - formatierte Ausgabekonvertierung

**UEBERSCHRIFT**

```
#include <stdio.h>

printf(format [ , arg ] ... )
char *format;

fprintf(filep, format [ , arg ] ... )
FILE *filep;
char *format;

sprintf(s, format [ , arg ] ... )
char *s, format;
```

**BESCHREIBUNG**

Printf gibt auf das Standardausgabefile stdout aus. Fprintf gibt auf das durch filep gekennzeichnete Ausgabefile aus. Sprintf gibt auf die Zeichenkette s aus, die mit dem Zeichen '\0' abgeschlossen wird.

Jede dieser Funktionen konvertiert, formatiert und gibt seine Argumente nacheinander aus, wobei das erste Argument eine Steuerfunktion hat. Das erste Argument ist eine Zeichenkette, die zwei Typen von Objekten enthaelt: einfache Zeichen, die genauso in den Ausgabestrom kopiert werden und Umwandlungsspezifikationen, die die Konvertierung und das Ausgeben des naechsten arg printf verursachen.

Jede Umwandlungsspezifikation wird durch das Zeichen '%' eingeleitet. Diesem Prozentzeichen kann folgen

- ein Minuszeichen '-', das den konvertierten Wert als unmittelbar naechstes Zeichen in dem bezeichneten Feld eintraegt;

- **eine Ziffernfolge, die eine Feldlaenge spezifiziert.** Umfasst der konvertierte Wert weniger Zeichen als die Feldlaenge, so werden links Leerzeichen aufgefuellt (ist das '-' -Zeichen angegeben, wird rechts mit Leerzeichen aufgefuellt). Beginnt die Feldlaengenangabe mit einer Null, so werden anstelle der Leerzeichen entsprechend Nullen eingefuegt.

- ein Punkt '.' , der dazu dient, die Feldlaenge von der naechsten Ziffernfolge zu trennen.

- eine Ziffernfolge, die eine Genauigkeit bestimmt. Hier wird die Anzahl der Ziffern, die nach dem Dezimalpunkt folgen sollen, festgelegt (fuer e- und f- Umwandlungen). Fuer andere Umwandlungen wird hier die max. zu druckende Anzahl der Zeichen einer Zeichenkette bestimmt.

- der Buchstabe **l**. Folgen diesem Zeichen die Buchstaben **d**, **o**, **x**, oder **u**, so wird dadurch festgelegt, dass diese sich auf ein long integer arg beziehen.

- ein Zeichen, das den Typ der anzuwendenden Konvertierung bestimmt.

Anstelle einer Ziffernfolge fuer eine Feldlaenge oder eine Genauigkeit kann ein '*' stehen. In diesem Fall liefert ein Integer arg den gewuenschten Wert.

Die Konvertierungszeichen und ihre Bedeutung sind:

**dox** Das Integer arg wird entsprechend in dezimale, oktale oder hexadezimale Schreibweise konvertiert.

**f** Das float oder double arg wird in die dezimale Schreibweise der Art '[-]ddd.ddd' umgewandelt. Hierbei ist die Anzahl der d's nach dem Dezimalpunkt gleich der Genauigkeit, die fuer das Argument festgelegt wurde. Ist die Genauigkeit nicht angegeben, werden 6 Ziffern ausgegeben. Es werden weder Ziffern noch ein Dezimalpunkt ausgegeben, wenn die Genauigkeit explizit Null ist.

**e** Das float oder double arg wird in die Schreibweise '[-]d.ddde+_dd' umgewandelt. Hierbei erscheint eine Ziffer vor dem '.' und die Anzahl der Ziffern danach ist gleich der Genauigkeitsfestlegung fuer das Argument. Wird keine Genauigkeit angegeben, so werden wie oben 6 Ziffern ausgegeben.

**g** Das float oder double arg wird entweder in die Schreibweise **d**, **f**, oder **e** umgewandelt, je nachdem welche der 3 Schreibweisen bei voller Genauigkeit den geringsten Speicherplatzbedarf hat.

c Das Zeichen (character) wird ausgegeben. Nullzeichen werden ignoriert.

s Arg wird als Zeichenkette verwendet ( Zeiger auf ein Zeichen bzw. charater pointer). Von dieser Zeichenkette werden so viele Zeichen ausgegeben, wie durch die Genauigkeitsangabe festgelegt wurden. Falls die Genauigkeit nicht angegeben oder Null ist, so werden alle Zeichen bis zu dem Nullcharacter ('\0') ausgegeben.

u Das unsigned (vorzeichenlose) Integer arg wird in eine Dezimalzahl umgewandelt und ausgegeben.

% Gibt ein '%' aus; es wird kein Argument konvertiert.

Eine nicht existierende oder eine zu kleine Feldlaenge kann die Beschneidung eines Feldes zur Folge haben. Es muss darauf geachtet werden, dass dies nicht vorkommt. Ein Auffuellen mit Null- oder Leerzeichen erfolgt nur dann, wenn die aktuelle Feldlaenge kleiner als die spezifizierte Feldlaenge ist. Zeichen, die durch printf erzeugt wurden, werden durch putc(3) ausgegeben.

**Beispiele**

Um das Datum und die Zeit in der Form 'Sunday, July 3, 10:02' zu drucken (weekday und month sind, Zeiger auf Zeichefolgen, die durch '\0' begrenzt sind) wird folgendes geschrieben:

```
printf("%s, %s %d, %02d:%02d", weekday, month, day,
hour, min);
```

Um die Konstante pi auf 5 Stellen genau auszugeben schreibt man:

```
printf("pi = %.5f", 4*atan(1.0));
```

**SIEHE AUCH**

putc(3), scanf(3), ecvt(3)

**FEHLERBEHANDLUNG**

Sehr grosse Felder ( ) 128 Zeichen ) fuehren zu Fehlern.

---

**NAME**

putc, putchar, fputc, putw - Ausgabe eines Zeichens oder Wortes in ein File

**UEBERSICHT**

```
#include <stdio.h>

int putc(c, filep)
char c;
FILE *filep;

putchar(c)

fputc(c, filep)
FILE *filep;

putw(w, filep)
FILE *filep;
```

**BESCHREIBUNG**

Putc fuegt das Zeichen c an das durch filep bezeichnete File an. Rueckkehrwert ist das ausgegebene Zeichen.

Putchar(c) ist als putc(c, stdout) in <stdio.h> als Makro definiert.

Fputc realisiert das gleiche wie putc, ist aber als Funktion und nicht als Makro implementiert. Bei Benutzung von fputc wird ein optimierter Objektcode erreicht.

Putw fuegt das Wort w (Typ int) an das durch filep gekennzeichnete File an.
Rueckkehrwert ist das ausgegebene Wort. Putw verlangt und erzeugt keine spezielle Strukturierung innerhalb des Files.

Das Standard-File stdout arbeitet dann und nur dann gepuffert, wenn es nicht einem Terminal zugeordnet ist. Diese Standardfestlegung kann mit setbuf(3) geaendert werden. Das File stderr ist standardmaessig ungepuffert. Die Anwendung von freopen auf dieses File (siehe fopen(3)) bewirkt eine Pufferung bei darauffolgenden Ausgaben. Mit setbuf kann diese Festlegung wieder beliebig geaendert werden.

Arbeitet ein Ausgabedatenstrom ungepuffert, so ist die zu uebertragende Information unmittelbar nach dem Senden im Ziel-File erreichbar. In einer gepufferten Ausgabe werden erst viele gesendete Zeichen gesammelt und dann als Block zum Ziel-File geschrieben. Fflush (siehe fclose(3)) kann benutzt werden, um das Schreiben des Blockes frueher zu realisieren.

**SIEHE AUCH**

fopen(3), fclose(3), getc(3), puts(3), printf(3), fread(3)

**MERKMALE**

Beim Auftreten eines Fehlers ist die Konstante **EOF** der Rueckkehrwert dieser Funktionen. Da **EOF** aber auch eine tatsaechlich uebertragene Integer-Groesse sein kann, sollte das Auftreten von Uebertragungsfehlern bei putw mit ferror(3) ueberprueft werden.

**FEHLERQUELLEN**

Da putc als Makro implementiert ist, koennen filep-Argumente, die sich aus Ausdrucken ergeben, falsch behandelt werden. So arbeitet zum Beispiel 'putc(c, *f++);' nicht wie erhofft.

**PUTS(3S)** **PUTS(3S)**

---

**NAME**

puts, fputs - Ausgabe einer Zeichenkette in ein File

**UEBERSICHT**

```
#include <stdio.h>

puts(s)
char *s;

fputs(s, filep)
char *s;
FILE *filep;
```

**BESCHREIBUNG**

Puts kopiert die mit Null abgeschlossene Zeichenkette s zum Standard-Ausgabe-File stdout und fuegt das 'newline'-Zeichen an.

Fputs kopiert die mit Null abgeschlossene Zeichenkette s zu dem Ausgabe-File, das durch filep gekennzeichnet ist. Keine der beiden Routinen kopiert das als Abschluss der Zeichenkette verwendete Null-Zeichen ('\Ø').

**SIEHE AUCH**

fopen(3), gets(3), putc(3), printf(3), ferror(3)
fread(3) fwrite

**FEHLERQUELLEN**

Puts fuegt ein 'newline'-Zeichen an, fputs nicht. Beides hat historische Gruende.

**QSORT(3)** **QSORT(3)**

---

**NAME**

qsort - Schnelles Sortieren (quicker sort)

**UEBERSCHRIFT**

```
qsort(base, nel, width, compar)
char *base;
int (*compar)( );
```

**BESCHREIBUNG**

Qsort ist eine Implementation des Quicker-Sort-Algorithmus. Das erste Argument ist ein Zeiger auf den Bereich, in dem sich die zu sortierenden Daten befinden, das zweite ist die Anzahl der Elemente, das dritte ist die Byteanzahl fuer ein Element und das letzte ist der Name der Vergleichsroutine. Die Vergleichsroutine wird mit zwei Argumenten aufgerufen, die Zeiger auf die zu vergleichenden Elemente sind. Die Routine muss je nachdem, ob das erste Argument groesser, gleich oder kleiner dem zweiten Argument ist, eine entsprechende ganze Zahl, die groesser, gleich bzw. kleiner als 0 ist, zurueckgeben.

**SIEHE AUCH**

sort(1)

**RAND(3)** **RAND(3)**

---

**NAME**

**rand, srand** - Zufallszahlengenerator

**UEBERSCHRIFT**

```
srand(seed)
int seed;

rand()
```

**BESCHREIBUNG**

**Rand** verwendet einen Zufallszahlengenerator auf der Basis von Modulo-Multiplikation mit der Periodenlaenge von 2 ** 32 und liefert sukzessive Pseudo-Zufallszahlen im Intervall von 0 bis 2 **15 - 1.

Der Generator wird reinitialisiert durch den Auruf von **srand** mit 1 als Argument. Er kann auf einen zufaelligen Startpunkt gesetzt werden (mit einem beliebigen Argument).

**SCANF(3S)** **SCANF(3S)**

---

**NAME**

scanf, fscanf, sscanf - Formatierte Eingabekonvertierung

**UEBERSCHRIFT**

```
#include <stdio.h>

scanf(format [ , pointer ] . . . )
char *format;

fscanf(filep, format [ , pointer ] . . . )
FILE *filep;
char *format;

sscanf(s, format [ , pointer ] . . . )
char *s, *format;
```

**BESCHREIBUNG**

Scanf liest vom Standard-Eingabe-File stdin. Fscanf liest von dem durch filep gekennzeichneten Eingabefile. Sscanf liest von der Zeichenkette (character string) s. Jede Funktion liest Zeichen, interpretiert sie entsprechend einem Format und speichert die Ergebnisse in ihren Argumenten. Jede Funktion verlangt als Argumente eine Steuerzeichenkette format und eine Reihe von pointer, die angeben, wo die umgewandelte Eingabe gespeichert werden soll.

Die Steuerzeichenkette enthaelt meistens Umwandlungsspezifikationen, die zur direkten Interpretation der Eingabesequenzen verwendet werden. Die Steuerzeichenkette kann folgendes enthalten:

1. Leerzeichen, Tabulatoren oder Newlines ("white space").

2. Ein gewoehnliches Zeichen, ausser %, welches dem naechsten Zeichen im Eingabestrom entspricht.

3. Umwandlungsspezifikationen, die aus dem Zeichen %, einer wahlweisen Zuweisung, die durch das Zeichen '*' unterdrueckt werden kann, einer wahlweisen numerischen maximalem Feldgroesse und einem Umwandlungszeichen bestehen.

Eine Umwandlungsspezifikation gilt fuer die Umwandlung des naechsten Eingabefeldes. Das Ergebnis wird der Variablen, auf die das entsprechende Argument zeigt, zugewiesen, falls nicht die Zuweisungsunterdrueckung durch das Zeichen * angegeben wurde. Ein Eingabefeld ist definiert als eine Zeichenkette ohne Leerzeichen. Es erstreckt sich bis zum naechsten unpassenden Zeichen oder bis die Feldgroesse, falls spezifiziert, erreicht ist.

Das Umwandlungszeichen kennzeichnet die Interpretation des Eingabefeldes. Das entsprechende Zeiger-Argument muss ein Zeiger auf einen definierten Typ sein. Die folgenden Umwandlungszeichen sind zulaessig:

% ein einzelnes '%' wird in der Eingabe an dieser Stelle erwartet. Eine Zuweisung wird nicht ausgefuehrt.

d ein dezimales Integer wird erwartet. Das entsprechende Argument sollte ein Integer-Pointer sein.

o ein oktales Integer wird erwartet. Das entsprechende Argument sollte ein Integer-Pointer sein.

x ein hexadezimales Integer wird erwartet. Das entsprechende Argument sollte ein Integer-Pointer sein.

s eine Zeichenkette (character string) wird erwartet. Das entsprechende Argument sollte ein Zeiger auf ein aus Zeichen bestehendes Array (character pointer) sein, dass gross genug ist, um diese Zeichenkette und ein abschliessendes '\0', das hinzugefuegt wird, aufzunehmen. Das Eingabefeld wird durch ein Leerzeichen oder ein Newline beendet.

c ein Zeichen (character) wird erwartet. Das entsprechende Argument sollte ein Character-Pointer sein. Das normale Uebergehen von Leerzeichen wird in diesem Fall unterdrueckt. Um das naechste Nicht-Leerzeichen zu lesen, kann '%1s'verwendet werden. Wird eine Feldgroesse angegeben, sollte sich das entsprechende Argument auf ein Character-Array beziehen. Die angegebene Anzahl von Zeichen (character) wird gelesen.

f,e eine Floating-Point-Zahl wird erwartet. Das naechste Feld wird dementsprechend konvertiert und im entsprechenden Argument gespeichert, welches ein Zeiger auf eine float sein sollte. Das Eingabeformat einer Floating-Point-Zahl ist eine wahlweise mit einem Vorzeichen versehene, aus Ziffern bestehende Zeichenkette, die moeglicherweise einen Dezimalpunkt enthaelt und der wahlweise ein Exponentenfeld folgen kann. Das Exponentenfeld besteht aus einem E oder e, dem eine, moeglicherweise mit einem Vorzeichen versehene ganze Zahl folgt.

[ kennzeichnet eine Zeichenkette, die nicht durch Leerzeichen begrenzt werden soll. Der linken Klammer folgt eine Menge von Zeichen und eine rechte Klammer. Die Zeichen zwischen den Klammern definieren eine Menge, deren Elemente eine Zeichenkette bilden koennen. Ist das erste Zeichen nicht das Circumflex '^', besteht das Eingabefeld aus allen Zeichen, bis ein Zeichen erscheint, das nicht Element der zwischen den Klammern angegebenen Menge ist. Ist das erste Zeichen nach der linken Klammer ein '^', besteht das Eingabefeld aus allen Zeichen, bis ein Zeichen auftritt, das Element der uebriggebliebenen Menge von Zeichen zwischen den Klammern ist. Das zugehoerige Argument muss auf ein Character-Feld zeigen.

Die Umwandlungszeichen **d**, **o** und **x** koennen gross geschrieben werden. Ihnen kann ein **l** vorangestellt werden, um anzugeben, dass der Zeiger auf **long** anstatt auf **int** in der Argumentliste steht. Die Umwandlungszeichen **e** oder **f** koennen ebenfalls grossgeschrieben werden, oder ihnen kann ein **l** vorangestellt werden, um anzuzeigen, dass ein Zeiger auf **double** anstatt auf **float** vorliegt. Wird den Umwandlungszeichen **d**, **o** und **x** ein **h** vorangestellt, so wird damit angegeben, dass ein Zeiger auf **short** anstatt auf **int** vorliegt.

Die scanf -Funktionen geben die Anzahl der erfolgreich verarbeiteten Formatspezifikationen zurueck. Dies kann verwendet werden, um zu entscheiden, wieviel Elemente gefunden wurden. Die Konstante **EOF** wird bei Erreichen des Fileendes zurueckgegeben. Beachte, dass dies etwas anderes ist als '\0'!
Dies bedeutet, dass keine Umwandlung durchgefuehrt wurde. War eine Umwandlung vorgesehen, wurde sie durch ein unpassendes Zeichen in der Eingabe verhindert.

Zum Beispiel wird der Aufruf

```
int i; float x; char name[50];
scanf( "%d%f%s", &i, &x, name);
```

bei der Eingabe-Zeile

```
25   54.32E-1  thompson
```

dem i den Wert 25, x den Werte 5.432 zuweisen und name wird 'thompson\0' erhalten. Oder

```
int i; float x; char name[50];
scanf("%2d%f%d%[1234567890]", &i, &x, name);
```

wird bei der Eingabe

```
56789 0123 56a72
```

die 56 dem i zuweisen, die 789.0 dem x, die '0123' ueberspringen und die Kette '56\0' in name unterbringen. Der naechste Aufruf von getchar wird 'a' zurueckuebergeben.

**SIEHE AUCH**

atof(3), getc(3), printf(3)

**FEHLERBEHANDLUNG**

Die scanf -Funktionen uebergeben **EOF** bei Erreichen des Fileendes und ein Zaehler (Typ short) gibt die Anzahl der fehlenden oder unzulaessigen Daten-Elemente an.

**FEHLERQUELLEN**

Der Erfolg der Literal-Vergleiche und der unterdrueckten Zuweisungen wird nicht sofort bestimmt.

---

**NAME**

setbuf - Festlegen der Pufferung fuer ein File

**UEBERSICHT**

```
#include <stdio.h>

setbuf(filep buf)
FILE *filep;
char *buf;
```

**BESCHREIBUNG**

Der Aufruf von setbuf erfolgt nach dem Oeffnen des Files, aber bevor Lese- oder Schreiboperationen auf das File ausgefuehrt werden. Er bewirkt, dass das Zeichenfeld buf anstelle des sonst automatisch zugewiesenen Puffers benutzt wird. Ist buf der konstante Zeiger **NULL**, so erfolgen die weiteren Ein- und Ausgaben auf dieses File ungepuffert.

Die Konstante BUFSIZ legt die Groesse des benoetigten Feldes fest.

```
char buf[BUFSIZ];
```

Im Standardfall erfolgt die Zuweisung eines Puffers an ein File durch malloc(3) waehrend des ersten getc- oder putc(3) - Zugriffes auf das zugehoerige File. Ausnahmen sind Ausgaben, die zu Terminals gerichtet sind, sowie das Standard-Ausgabe-File stderr, die ungepuffert arbeiten.

**SIEHE AUCH**

fopen(3), getc(3), putc(3), malloc(3)



---

**NAME**

setjmp, longjmp - Nicht-lokales goto

**UEBERSCHRIFT**

```
#include <setjmp.h>

setjmp(env)
jmp_buf env;

longjmp(env, val)
jmp_buf env;
```

**BESCHREIBUNG**

Diese Routinen koennen zur Behandlung von Fehlern und Interrupts, die in niederen Unterroutinen eines Programmes eintreten, verwendet werden.

Setjmp kellert das Stack-Environment in env zur spaeteren Benutzung durch longjmp. Sein Rueckkehrwert ist 0.

Longjmp speichert das durch das letzte setjmp gerettete Stack-Environment zurueck. Die Programmausfuehrung wird so fortgesetzt, als waere der Systemruf setjmp gerade erfolgt. Dabei wird val als Rueckkehrwert geliefert. Alle zugreifbaren Daten haben die Werte, die sie waehrend des Systemrufes longjmp besassen.

**SIEHE AUCH**

signal(2)

**SIN(3M)** **SIN(3M)**

---

**NAME**

**sin, cos, tan, asin, acos, atan, atan2**
**- Trigonometrische Funktionen**

**UEBERSICHT**

```
#include <math.h>

double sin(x)
double x;

double cos(x)
double x;

double tan(x)
double x;

double asin(x)
double x;

double acos(x)
double x;

double atan(x)
double x;

double atan2(x, y)
double x, y;
```

**BESCHREIBUNG**

**Sin, cos** und **tan** liefern die trigonometrischen Funktionen fuer Argumente im Bogenmass. Die Groesse der Argumente sollte vom Anwender geprueft werden, um ein sinnvolles Resultat zu gewaehrleisten.

**asin** liefert den arcsin im Intervall -pi/2 bis pi/2.

**acos** liefert den arccos im Intervall 0 bis pi.

**atan** liefert den arctan fuer x im Intervall -pi/2 bis pi/2.

**atan2** liefert den arctan von x/y im Intervall -pi bis pi.

**FEHLERBEHANDLUNG**

**asin** und **acos** liefern fuer Argumente groesser als 1 den Wert 0, und errno wird EDOM gesetzt. Der Wert von **tan** in seinen singulaeren Punkten ist eine "unendlich" grosse Zahl, und errno wird ERANGE gesetzt.

**FEHLERHINWEISE**

**tan** liefert fuer Argumente groesser $2^{31}$ sinnlose Werte.

**SINH(3M)** **SINH(3M)**

---

**NAME**

**sinh, cosh, tanh** - Hyperbolische Funktionen

**UEBERSICHT**

```
#include <math.h>

double sinh(x)
double x;

double cosh(x)
double x;

double tanh(x)
double x;
```

**BESCHREIBUNG**

Diese Funktionen berechnen die angegebenen hyperbolischen Funktionen fuer reelle Argumente.

**FEHLER**

**Sinh** und **cosh** liefern eine "unendlich" grosse Zahl mit entsprechendem Vorzeichen, wenn der korrekte Wert wegen Ueberlauf nicht mehr darstellbar ist.

**NAME**

sleep - Aussetzen der Ausfuehrung fuer eine bestimmte Zeit

**UEBERSCHRIFT**

**sleep(seconds)**
**unsigned seconds;**

**BESCHREIBUNG**

Der aktuelle Prozess wird fuer die angegebene Anzahl von Sekunden nicht abgearbeitet. Die tatsaechliche Pause kann bis zu einer Sekunde kuerzer sein, da der Scheduler in Ein-Sekunden Intervallen Prozesse wieder aktiviert. Durch andere Systemaktivitaeten kann die Pause unbestimmbar laenger sein.

Die Routine ist durch Setzen eines Alarm-Signals und einer Pause bis zu dessen Eintreffen implementiert. Der vorangegangene Status dieses Signals wird gerettet und bei Fortsetzung des Prozesses zurueckgespeichert. Ist die angegebene Zeit groesser als der Rest einer vorher eingestellten Zeit fuer das Alarm-Signal, wird die Pause nach Ablauf der vorher eingestellten Zeit beendet und das Signal eine Sekunde spaeter gesendet.

**SIEHE AUCH**

alarm(2), pause(2)

---

**NAME**

stdio - Standardpaket fuer gepufferte Ein- und Ausgaben

**UEBERSICHT**

```
#include <stdio.h>

FILE *stdin;
FILE *stdout;
FILE *stderr;
```

**BESCHREIBUNG**

Die Funktionen, die unter (3S) beschrieben sind, stellen ein wirksames Mittel zur Realisierung gepufferter Ein- und Ausgaben durch den Nutzer dar. Zur zuegigen Behandlung einzelner Zeichen dienen die Makros getc und putc(3). Eine hoehere Ebene stellen die Routinen gets, fgets, scanf, fscanf, fread, fwrite, puts, fputs, printf und fprintf dar, die alle auf getc und putc basieren und in beliebiger Reihenfolge aufgerufen werden koennen.

Ein File mit zugeordneter Pufferung wird Stream genannt und ist als Zeiger filep auf einen definierten Typ **FILE** vereinbart. Fopen(3) erstellt verschiedene einen Stream beschreibende Daten und liefert einen Zeiger filep, der der Identifizierung des Streams in allen darauffolgenden Operationen dient. Drei geoeffnete Streams mit konstanten Zeigern, die im Include-File vereinbart werden, sind verbunden mit folgenden standardmaessig geoeffneten Files:

| | |
|---|---|
| **stdin** | Standard-Eingabefile |
| **stdout** | Standard-Ausgabefile |
| **stderr** | Standard-Fehlerfile |

Der konstante 'Zeiger' **NULL** (0) kennzeichnet einen nichtvorhandenen Stream.

Integer - Funktionen, die sich auf Streams beziehen, liefern beim Auftreten eines Fehlers oder bei Erreichen des File-Endes die Integer-Konstante **EOF** (-1) als Rueckkehrwert.

Jede Routine, die das Standardpaket fuer Ein-/Ausgaben benutzt, muss das Header-File <stdio.h> enthalten, welches die noetigen Macro-Definitionen beinhaltet. Auch die Funktionen und Konstanten, die in den mit (3S) bezeichneten Abschnitten aufgefuehrt werden, sind in diesem Include-File vereinbart und bedueerfen keiner weiteren Vereinbarungen. Eine Neuvereinbarung der Konstanten sowie der folgenden Funktionen, die als Makros implementiert sind, ist gefaehrlich: getc, getchar, putc, putchar, feof, ferror, fileno.

**SIEHE AUCH**

open(2), close(2), read(2), write(2)

**MERKMALE**

Als Rueckkehrwert wird einheitlich **EOF** geliefert, wenn angezeigt werden soll, dass ein Pointer auf **FILE** nicht durch fopen initialisiert wurde, dass eine Eingabe (Ausgabe) an einen Ausgabe- (Eingabe)-Stream gerichtet war oder dass ein Zeiger auf **FILE** auf verfaelschte **FILE**-Daten zeigt, die nicht mehr in der geforderten Weise interpretiert werden koennen.

**STRING(3)** **STRING(3)**

---

**NAME**

**strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, index, rindex** - Zeichenkettenoperationen

**UEBERSCHRIFT**

```
char *strcat(s1, s2)
char *s1, *s2;

char *strncat(s1, s2, n)
char *s1, *s2;

strcmp(s1, s2)
char *s1, *s2;

strncmp(s1, s2, n)
char *s1, *s2;

char *strcpy(s1, s2)
char *s1, *s2;

char *strncpy(s1, s2, n)
char *s1, *s2;

strlen(s)
char *s;
```

```
char *index(s, c)
char *s, c;

char *rindex(s, c)
char *s;
```

**BESCHREIBUNG**

Diese Funktionen operieren mit Zeichenketten, die mit '0' abgeschlossen sind. Es wird bei den resultierenden Zeichenketten nicht auf overflow geprueft.

**Strcat** fuegt eine Kopie der Zeichenkette s2 an das Ende der Zeichenkette s1 an. **Strncat** kopiert hoechstens n Zeichen. Beide liefern einen Zeiger auf eine durch '0' beendete Zeichenkette.

**Strcmp** vergleicht seine Argumente und kehrt mit einer ganzen Zahl goesser, gleich oder kleiner Null zurueck. Je nachdem, ob s1 lexikographisch groesser, gleich oder kleiner als s2 ist. **Strncmp** arbeitet analog, vergleicht aber hoechstens n Zeichen.

**Strcpy** kopiert Zeichenkette s2 zu s1, bis das Zeichen '0' erkannt wird.
**Strncpy** kopiert exakt n Zeichen, wobei s2 entweder abgeschnitten oder mit Nullen aufgefuellt wird. Es kann sein, dass das Ergebnis (S1) nicht mit 0 endet, wenn die Laenge von s2 n oder mehr betraegt.

**Strlen** liefert die Anzahl der von 0 verschiedenen Zeichen von s.

**Index** (**rindex**) liefert einen Zeiger auf die erste (letzte) Position, an der das Zeichen c in der Zeichenkette s auftritt bzw. 0, wenn c in der Zeichenkette nicht vorkommt.



---

**NAME**

system - Abarbeiten eines Shell - Kommandos

**UEBERSCHRIFT**

```
system(command)
char *command;
```

**BESCHREIBUNG**

System uebergibt command dem Kommandointerpreter sh(1) als Eingabe. Die Zeichenkette enthaelt den Kommandonamen und die Argumente in der gleichen Art wie sie ueber Tastatur eingegeben werden. Der aufrufende Prozess

wartet, bis das Shell - Kommando abgearbeitet ist. Rueckkehrwert der Funktion ist der Exit - Status des Shell-Kommandos.

**SIEHE AUCH**

popen(3), exec(2), wait(2)

**FEHLERBEHANDLUNG**

Exit - Status 127 bedeutet, dass Shell nicht ausgefuehrt werden konnte.

**TTYNAME(3)** **TTYNAME(3)**

---

**NAME**

ttyname, isatty, ttyslot - Ermitteln des Terminalnamens zum Filedeskriptor

**UEBERSCHRIFT**

**char *ttyname(fildes)**

**isatty(fildes)**

**ttyslot()**

**BESCHREIBUNG**

Ttyname hat als Rueckkehrwert einen Zeiger zu dem mit Null begrenzten Pfadnamen des Terminals, welches dem Filedeskriptor fildes zugeordnet ist.

Isatty hat den Rueckkehrwert 1, wenn fildes einem Terminal zugeordnet ist. Andernfalls ist der Rueckkehrwert 0.

Ttyslot hat als Rueckkehrwert die Zeilennummer des Eintrags in ttys(5) fuer dasjenige Terminal, von dem aus der betreffende Prozess gestartet wurde.

**FILES**

/dev/*
/etc/ttys

**SIEHE AUCH**

ioctl(2), ttys(5)

**FEHLERBEHANDLUNG**

Ttyname hat als Rueckkehrwert einen Zeiger mit dem Wert 0 wenn fildes kein Terminal aus dem Directory '/dev' zugeordnet ist.

Ttyslot gibt 0 zurueck, wenn auf '/etc/ttys' nicht zugegriffen werden kann oder wenn das Terminal dort nicht gefunden wird.

**FEHLERHINWEISE**

Der Rueckkehrwert verweist auf eine "static" - Variable, deren Inhalt bei jedem Aufruf ueberschrieben wird.



---

**NAME**

ungetc - Rueckschreiben eines Zeichens zum Eingabe-File

**UEBERSICHT**

```
#include <stdio.h>

ungetc(c, filep)
FILE *filep;
```

**BESCHREIBUNG**

Ungetc schreibt das Zeichen c in das Eingabefile zurueck, so dass der naechste getc-Ruf, der sich auf dieses File bezieht, c als Rueckkehrwert liefert. Der Rueckkehrwert von ungetc ist c.

Voraussetzung fuer das korrekte Rueckschreiben des Zeichens ist, dass das File gepuffert arbeitet und dass schon aus dem File gelesen wurde. **EOF** kann nicht zurueckgeschrieben werden.

Fseek(3) hebt die Wirkungen von ungetc auf.

**SIEHE AUCH**

getc(3), setbuf(3), fseek(3)

**MERKMALE**

Ungetc liefert **EOF** als Rueckkehrwert, wenn ein Zeichen nicht zurueckgeschrieben werden kann.

**NAME**

**swab** - Vertauschen von Bytes

**UEBERSCHRIFT**

```
swab(from, to, nbytes)
char *from, *to;
```

**BESCHREIBUNG**

Swab kopiert nbytes Bytes ab der Adresse, auf die durch from verwiesen wird zu dem Platz auf den to zeigt. Dabei werden die Bytes in einem Wort jeweils vertauscht. Nbytes muss geradzahlig sein.