Deutsch

FUJITSU Software BS2000

C-Bibliotheksfunktionen V3.1B

C-Bibliotheksfunktionen

Referenzhandbuch

Ausgabe November 2014

Kritik… Anregungen… Korrekturen…

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

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

Zertifizierte Dokumentation nach DIN EN ISO 9001:2008

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

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

Copyright und Handelsmarken

Inhalt

1 Einleitung

Das C-Laufzeitsystem ist Bestandteil des Common Runtime Environment CRTE. Bei Einsatz des CRTE in BS2000-Betriebssystemen ohne POSIX stellt das C-Laufzeitsystem über 300 vordefinierte Funktionen zur Verfügung. Dies sind alle im ANSI/-ISO-Standard definierten Funktionen (inklusive dem ISO-C Amendment 1 zum ISO/IEC 9899:1990) sowie ca. 50 BS2000-spezifische Erweiterungen. Diese Funktionen erlauben die komfortable Programmierung vieler Aufgaben, für die die Sprache C selbst keine höheren Sprachmittel vorsieht. Einige Beispiele für solche Programmieraufgaben sind:

– Verarbeitung von Dateien (Öffnen, Schließen, Positionieren, Lesen, Schreiben etc.)
– Verarbeitung von einzelnen Zeichen oder Zeichenketten (Suchen, Ändern, Kopieren, Löschen etc.)
– Verarbeitung von Multibyte-Zeichen und Langzeichen
– Verarbeitung des Datentyps `long long integer`
– Dynamische Speicherverwaltung (Speicherbereiche anlegen, freigeben etc.)
– Zugriffe auf Betriebssystem-Funktionen (Systemkommandos, Dienstprogramme etc.)
– Mathematische Funktionen (trigonometrische, logarithmische etc.)

Die Funktionen liegen entweder als Quellprogrammteile (Makros) oder als bereits übersetzte Programmteile (Module) vor. In diesem Handbuch wird für beide Ausführungen der Begriff „Funktion“ verwendet, sofern es auf den Unterschied in der Ausführung nicht ankommt.

Die erforderlichen Deklarationen der Funktionen, die Definitionen von Konstanten, Datentypen und Makros sowie die Funktionsmakros selbst stehen in den „Include-Dateien“ (in der C-Literatur häufig auch „Standard-Include-Dateien“, „Include-Header“ o.ä. genannt). Die Include-Dateien sind Quellprogrammteile, die - im C-Programm mit der `#include`-Anweisung angesprochen - bei jeder Übersetzung temporär in das Programm kopiert werden.

Alle Funktionen und alle Include-Dateien sind in CRTE-Bibliotheken als Bibliothekselemente abgespeichert.

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

1.1 Zielsetzung und Zielgruppen des Handbuchs

Dieses Handbuch beschreibt alle C-Funktionen und Makros des C-Laufzeitsystems, die im BS2000-Betriebssystem ohne POSIX genutzt werden können. Es wendet sich an C-Anwender, die CRTE V2.9B in BS2000-Betriebssystemen einsetzen, in denen kein POSIX-Subsystem verfügbar ist, und an die Entwickler von C-Applikationen, die ohne POSIX ablauffähig sein sollen.

Voraussetzung für die Arbeit mit diesem Handbuch sind Kenntnisse der Programmiersprache C sowie des Betriebssystems BS2000.

1.2 Konzept des Handbuchs

Kapitel 2 enthält Hinweise, was generell bei der Benutzung der Bibliotheksfunktionen zu beachten ist, z.B. Unterschiede zwischen Funktionen und Makros, Einfügen von Include-Dateien, Fehlerbehandlung.

Kapitel 3 gibt einen Überblick über die Bibliotheksfunktionen nach inhaltlichen Gesichtspunkten.

Die Kapitel 4 bis 6 enthalten grundlegende Informationen, Programmierhinweise und Beispiele zur Dateiverarbeitung, zu STXIT-/Contingency-Routinen und zur Lokalität.

Kapitel 7 ist der Nachschlageteil und enthält die Einzelbeschreibungen der Bibliotheksfunktionen in alphabetischer Reihenfolge.

Im Anhang finden Sie eine Übersicht, welche der beschriebenen Bibliotheks-Funktionen im ANSI-Standard bzw. im Amendement 1 definiert sind.
Außerdem enthält der Anhang eine Beschreibung zur Auswahl der KR-Funktionalität, die von C/C++-Compilerversionen kleiner V3.0 unterstützt wird.

Literaturhinweise werden im Text in Kurztiteln angegeben. Im Literaturverzeichnis ist der vollständige Titel jeder Druckschrift aufgeführt. Daran anschließend finden Sie Hinweise zur Bestellung von Handbüchern.

Die mit CRTE angebotenen POSIX-Bibliotheksfunktionen des C-Laufzeitsystems (ca. 300 im XPG4-Standard definierte Funktionen sowie einige UNIX/SINIX-spezifische Erweiterungen) sind im Handbuch „C-Bibliotheksfunktionen für POSIX-Anwendungen“ beschrieben.

Readme-Datei

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

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

Informationen unter BS2000/

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

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

Diese Datei enthält eine kurze Information zur Readme-Datei in deutscher oder englischer Sprache (<lang>=D/E). Die Information können Sie am Bildschirm mit dem Kommando `/SHOW-FILE` oder mit einem Editor ansehen.
Das Kommando `/SHOW-INSTALLATION-PATH INSTALLATION-UNIT=<product>` zeigt, unter welcher Benutzerkennung die Dateien des Produkts abgelegt sind.

Ergänzende Produkt-Informationen

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

1.3 Änderungen gegenüber dem Vorgänger-Handbuch

Die Änderungen des vorliegenden Handbuchs gegenüber dem C-Bibliothekshandbuch V3.1A sind auf folgende wesentliche Neuerungen des C-Laufzeitsystems V3.1B in CRTE V2.9B zurückzuführen:

- Unterstützung des Last Byte Pointers (LBP)
  - Einführung der Umgebungsvariablen LAST_BYTE_POINTER
  - *lbp*-Schalter bei den Funktionen `fopen`, `fopen64`, `freopen`, `freopen64`, `creat`, `creat64` und `open`, `open64`
- Beschreibung der Funktionen `environ`, `getenv` und `putenv`

1.4 Darstellungsmittel

In diesem Handbuch werden folgende Mittel zur Darstellung von funktional wichtigen Textteilen verwendet:

für Hinweistexte

ACHTUNG!

für Warnhinweise

kursive Schrift
: kennzeichnet eine Variable, wenn Sie dafür einen Wert einsetzen müssen.

`dicktengleiche Schrift`
: für die Darstellung von Eingaben für das System, Systemausgaben und für Dateinamen in Beispielen.

kommando
: In der Syntaxbeschreibung für Kommandos werden diejenigen Bestandteile (Bezeichnungen von Kommandos und Parametern) fett dargestellt, die unverändert eingegeben werden müssen.

2 Benutzung der Bibliotheksfunktionen

Dieses Kapitel enthält Hinweise, was generell bei der Benutzung der Bibliotheksfunktionen zu beachten ist.

2.1 Funktionen und Makros

Die meisten Bibliotheksfunktionen sind als C-Funktionen realisiert, einige als Makros. Einige Bibliotheksfunktionen sind sowohl als Funktion als auch als Makro realisiert (siehe Liste unten).

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

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

Gesteuert über Compileroptionen zur Optimierung können einige Bibliotheksfunktionen inline generiert werden. In diesem Fall wird der Funktionscode direkt in die Aufrufstelle eingesetzt und es entfallen die o.g. Verwaltungsaktivitäten.
Derzeit können folgende Funktionen inline generiert werden: `strcpy`, `strcmp`, `strlen`, `strcat`, `memcpy`, `memcmp`, `memset`, `abs`, `fabs`, `labs`.

Weitere Einzelheiten hierzu finden Sie in den Benutzerhandbüchern zu den Compilern C, C++ oder C/C++.

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

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

Bei der Benutzung eines Makros sind außerdem die folgenden Punkte zu beachten:

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

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

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

Liste der Bibliotheksfunktionen, die als Makro bzw. als Makro und als Funktion realisiert sind:

Makro:	**Funktion:**	**Makro:**	**Funktion:**
clearerr	(clearerr)	iswdigit	(iswdigit)
clock	(clock)	iswgraph	(iswgraph)
feof	(feof)	iswlower	(iswlower)
ferror	(ferror)	iswprint	(iswprint)
getc	(getc)	iswpunct	(iswpunct)
getchar	(getchar)	iswspace	(iswspace)
getwc	(getwc)	iswupper	(iswupper)
isalnum	(isalnum)	iswxdigit	(iswxdigit)
isalpha	(isalpha)	putc	(putc)
iscntrl	(iscntrl)	putchar	(putchar)
isdigit	(isdigit)	toebcdic	(toebcdic)
isebcdic	(isebcdic)	tolower	(tolower)
isgraph	(isgraph)	toupper	(toupper)
islower	(islower)	towlower	(towlower)
ispunct	(ispunct)	towupper	(towupper)
isspace	(isspace)	assert	
isupper	(isupper)	offsetof	
isxdigit	(isxdigit)	va_arg	
iswalnum	(iswalnum)	va_end	
iswalpha	(iswalpha)	va_start	
iswcntrl	(iswcntrl)		

2.2 Include-Dateien

Jede Bibliotheksfunktion ist in einer Include-Datei deklariert. Viele Bibliotheksfunktionen benutzen symbolische Konstanten und Datentypen, die in Include-Dateien definiert sind. Die als Makros realisierten Bibliotheksfunktionen stehen ebenfalls in Include-Dateien.

Die für die Verwendung von Bibliotheksfunktionen notwendigen Include-Dateien sind Bestandteile der CRTE-Bibliothek SYSLIB.CRTE. Sie sind dort als Quellprogrammelemente (Typ S) abgespeichert und werden bei der Übersetzung auf Grund der Präprozessoranweisung `#include` in das Programm kopiert. Wie dies geschieht, ist ausführlich in den Compiler-Benutzerhandbüchern dargestellt.

Folgende Include-Dateien stehen zur Verfügung:

<ascii_ebcdic.h>	<assert.h>	<bs2cmd.h>	<cglobals.h>	<cont.h>
<crduc.h>	<ctype.h>	<errno.h>	<float.h>	<ieee_390.h>
<ilcs.h>	<inttypes.h>	<iso646.h>	<limits.h>	<locale.h>
<math.h>	<setjmp.h>	<signal.h>	<stdarg.h>	<stddef.h>
<stdio.h>	<stdlib.h>	<string.h>	<strings.h>	<stxit.h>
<sys.timeb.h>	<sys.types.h>	<time.h>	<timeb.h>	<wchar.h>
<wctype.h>				

Die Include-Dateien enthalten u.a. Folgendes:

– Definitionen von symbolischen Konstanten mit den für die ordnungsgemäße Funktionsausführung notwendigen Werten, z.B.:

BUFSIZ	Betriebssystemspezifische Standardgröße des Ein-/Ausgabepuffers (8192 Bytes).
EOF	Dateiende-Kriterium (-1)
WEOF	Dateiende-Kriterium für Langzeichen-Dateien (L“-1“)
_NFILE	Maximal erlaubte Anzahl der gleichzeitig geöffneten Dateien einschließlich der Standarddateien `stdin`, `stdout` und `stderr` (2048).
NULL	Nullzeiger (0)

– Definitionen von Datentypen und Strukturen, die von den Funktionen benutzt werden, z.B.:

FILE	Die meisten Ein-/Ausgabefunktionen benutzen einen Zeiger auf eine Struktur vom Typ FILE (siehe auch Abschnitt „FILE-Struktur“ auf Seite 64)
mbstate_t	Dieser Datentyp wird von den Multibyte-Funktionen benutzt und entspricht in dieser Implementierung dem Typ `char`.
size_t	Dieser Datentyp wird von vielen Zeichenkettenfunktionen benutzt und entspricht in dieser Implementierung dem Typ `unsigned`.
ptrdiff_t	Dieser Datentyp wird für das Ergebnis einer Subtraktion von Zeigern verwendet und entspricht in dieser Implementierung dem Typ `integer`.
wint_t	Dieser Datentyp enthält Werte, die den Elementen des erweiterten Character-Sets entsprechen oder den Wert WEOF (Ende der Eingabe). In dieser Implementierung entspricht dieser Datentyp dem Typ `integer`.
wchar_t	Dieser Datentyp wird von den Multibyte-Funktionen benutzt und entspricht in dieser Implementierung dem Typ `long`.
wctrans_t	Skalarer Datentyp, der Lokalitäten-spezifische Zeichenabbildungen repräsentiert.
wctype_t	Skalarer Datentyp, der Lokalitäten-spezifische Zeichenklassen repräsentiert. In dieser Implementierung entspricht dieser Datentyp dem Typ `long`.
clock_t	Dieser Datentyp wird von der Funktion clock benutzt und entspricht in dieser Implementierung dem Typ `integer`.
time_t	Dieser Datentyp wird von vielen Zeitfunktionen benutzt und entspricht in dieser Implementierung dem Typ `long`.
va_list	Dieser Datentyp wird von den Funktionen benutzt, die variable Argumentenlisten bearbeiten (z.B. `vprintf`).

– Die Prototyp-Deklaration aller Funktionen

  Vor Aufruf einer Funktion muss der Datentyp bekannt, d.h. deklariert sein. Dies ist mit dem Einfügen der entsprechenden Include-Datei gewährleistet. In den Übersetzungsmodi „ANSI“ und „STRICT-ANSI“ erhält man eine Warnung, wenn die Deklaration fehlt. Im Übersetzungsmodus „CPLUSPLUS“ erhält man bei einer fehlenden Deklaration einen Fehler und es wird kein Modul erzeugt.

  Siehe auch Abschnitt „Präprozessor-Define _STRICT_STDC“ auf Seite 42.

– Die Definition aller Makros

  Einige Bibliotheksfunktionen sind als Makros realisiert. Um ein Makro benutzen zu können, muss die jeweilige Include-Datei in das Programm eingefügt werden.

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

Include-Datei iso646.h

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

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

2.3 Behandlung von Fehlern

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

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

Die meisten Funktionen liefern bei Auftreten eines Fehlers einen Fehler-Returnwert. Zusätzlich wird in vielen Fällen die C-interne Variable `errno` (Typ `integer`) auf einen entsprechenden Fehlercode gesetzt. Auf Grund dieses Fehlercodes werden dann intern (in einer Struktur) Informationen aufbereitet, die den Fehler näher spezifizieren. Die von der Funktion `perror` ausgegebene Information beinhaltet:

– einen kurzen Fehlertext, der den Fehler erläutert,
– den Namen der Funktion, bei der der Fehler aufgetreten ist,
– ggf. den DVS-Fehlercode (sedezimal) bei fehlerhaften Dateizugriffen.

Alle Fehlercodes sowie die dafür vorgesehenen Fehlerinformationen sind in der Include-Datei <errno.h> definiert.

Wenn bei einer Funktion verschiedene Arten von Fehlern und damit Fehlercodes möglich sind, kann es sinnvoll sein, die `errno`-Variable auf den Fehlercode abzufragen, um dann ggf. unterschiedlich darauf reagieren zu können. Jeder Fehlercode wird durch eine in <errno.h> definierte symbolische Konstante repräsentiert, z.B. bedeutet ERANGE Überlauffehler (Wert 2). Eine Abfrage könnte etwa folgendermaßen aussehen, z.B. hier bei der Funktion `signal`:

```
#include <errno.h>
       .
       .
if(signal(sig, fkt) == 1) /* Abfrage des Fehlerergebnisses */
{
  if(errno == EFAULT)
       .           /* Reaktionen auf EFAULT (unzulässige Adresse) */
       .
  else if(errno == EINVAL)
       .           /* Reaktionen auf EINVAL (unzulässiges Argument) */
       .
}
else...
```

Zusätzlich zur `errno`-Variablen sind in der Include-Datei <errno.h> zwei weitere Variablen definiert:

Mit `_ _errcmd` ist der Name der fehlerhaften Funktion ansprechbar und mit `_ _errhex` der sedezimale DVS-Code. Beide Variablen sind vom Typ `char[8]`.

Hinweise

– Die Variablen `errno`, `_ _errcmd` und `_ _errhex` dürfen nicht explizit vom Anwender definiert werden. Für die Abfrage dieser Variablen muss die Include-Datei <errno.h> in das Programm übernommen werden.

– Der Inhalt des Bereichs, in dem Fehlerinformationen intern abgespeichert werden, bleibt solange erhalten, bis er bei neuerlichem Auftritt eines Fehlers mit der aktuellen Information überschrieben wird. `perror`-Aufrufe sowie die Abfrage der Variablen `errno`, `_ _errcmd` und `_ _errhex` sind daher nur sinnvoll, unmittelbar nachdem eine Funktion einen Fehler-Returnwert geliefert hat.

In den Beispielen zu den einzelnen Funktionsbeschreibungen wurden die Fehlerabfragen häufig weggelassen, um die Beispiele nicht unnötig aufzublähen.

2.4 Zeiger als Returnwert und Ergebnisparameter

Returnwert Zeiger

<typ> *funkt(...)

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

Returnwert void *

void * funkt(...)

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

Beispiel

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

Ergebnisparameter Zeiger

<typ1> funkt(<typ2> *variable)

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

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

Beispiele

```
struct timeb tp;    /* Struktur */
ftime( &tp);
char erg;           /* char-Variable */
scanf("%c", &erg);
char array[10];     /* Zeichenketten-Variable */
scanf("%s", array);
```

2.5 IEEE-Gleitpunkt-Arithmetik

Die IEEE-Gleitpunkt-Arithmetik wird auf folgende Weise unterstützt:

– Es gibt eine Compiler-Option des C/C++-Compilers, mit der Gleitpunkzahlen im IEEE-Format erzeugt werden können (siehe Seite 28).

– Zu jeder Bibliotheksfunktion im C-Laufzeitsystem, die mit Gleitpunktzahlen arbeitet oder Gleitpunktzahlen zurückliefert, gibt es eine Variante für die Bearbeitung von IEEE-Gleitpunktzahlen sowie ein Makro-Define, das die Standard-Variante (/390-Variante) der Funktion auf die zugehörige IEEE-Variante abbildet (siehe Seite 29).

Per Compiler-Option können Sie die komplette IEEE-Funktionalität aktivieren: Der C/C++-Compiler erzeugt dann in allen Modulen Gleitpunktzahlen im IEEE-Format und stellt automatisch die passenden IEEE-Funktionen zur Bearbeitung der IEEE-Gleitpunktzahlen bereit.

Daneben haben Sie die Möglichkeit, die zur Verfügung gestellte IEEE-Funktionalität modifiziert zu nutzen:

– Mithilfe des Präprozessor-Defines `_IEEE_SOURCE` können Sie festlegen, ob die Bibliotheksfunktionen für /390-Gleitpunkt-Arithmetik auf die zugehörige IEEE-Varianten abgebildet werden (siehe Seite 30).

– Mithilfe von Konvertierungsfunktionen können Sie Gleitpunktzahlen explizit vom /390-Format in das IEEE-Format konvertieren (siehe Seite 31).

Hinweise zur Nutzung der IEEE-Gleitpunkt-Arithmetik

Bei Nutzung der IEEE-Gleitpunkt-Arithmetik ist Folgendes zu beachten:

– IEEE-Gleitpunkt-Operationen unterscheiden sich semantisch von den entsprechenden /390-Gleitpunkt-Operationen, z.B. beim Runden. So wird im IEEE-Format standardmäßig "Round to Nearest" angewendet anstatt "Round to Zero" wie im /390-Format.

– In Fehler- und Ausnahmefällen (z.B. Argument nicht im zulässigen Wertebereich) unterscheiden sich die Reaktionen der IEEE-Funktionen von denen der /390-Funktionen, z.B. liefern einige Funktionen den Wert `NaN` zurück.

– Inkludieren Sie für jede in Ihrem Programm verwendete C-Bibliotheksfunktion, die mit Gleitpunkzahlen arbeitet, die zugehörige Include-Datei. Andernfalls können diese Funktionen die Gleitpunktzahlen nicht korrekt verarbeiten.
Insbesondere müssen Sie für die Funktion `printf` die Include-Datei `<stdio.h>` mit `#include <stdio.h>` inkludieren.

2.5.1 Erzeugen von IEEE-Gleitpunktzahlen via Compiler-Option

Für Gleitpunktzahlen erzeugt der C/C++-Compiler wahlweise Code im /390-Format oder im IEEE-Format. Das gewünschte Format legen Sie mit der FP-ARITHMETICS-Klausel der Compiler-Option MODIFY-MODULE-PROPERTIES fest.

```
MODIFY-MODULE-PROPERTIES          -
...
FP-ARITHMETICS= {*390-FORMAT  },  -
                {*IEEE-FOR-   }
LOWER-CASE-NAMES=*YES,            -
SPECIAL-CHARACTERS=*KEEP,         -
...
```

FP-ARITHMETICS=*390-FORMAT
: Der Compiler erzeugt Code für Konstanten und Arithmetik-Operationen im /390-Format. *390-FORMAT ist Standard.

FP-ARITHMETICS=*IEEE-FORMAT
: Der Compiler erzeugt Code für Konstanten und Arithmetik-Operationen im IEEE-Format. Außerdem wird das Präprozessor-Define `_IEEE` auf 1 gesetzt. Sofern nicht das Präprozessor-Define `_IEEE_SOURCE` auf 0 gesetzt ist (siehe Seite 30), werden die Original-/390-Bibliotheksfunktionen automatisch auf die zugehörigen IEEE-Funktionen gesteuert.

LOWER-CASE-NAMES=*YES
SPECIAL-CHARACTERS=*KEEP
: Durch diese Angaben verhindern Sie, dass

  – die Namen der IEEE-Funktionen (siehe Seite 29) auf acht Zeichen gekürzt werden,

  – in den Funktionsnamen Kleinbuchstaben in Großbuchstaben umgewandelt und die Zeichen „_“ durch „$“ ersetzt werden.

In POSIX legen Sie das IEEE-Format mit der folgenden Option fest:

```
-K ieee_floats
```

Für die korrekte Verarbeitung der IEEE-Funktionsnamen spezifizieren Sie:

```
-K llm_keep
-K llm_case_lower
```

2.5.2 C-Bibliotheksfunktionen, die IEEE-Gleitpunktzahlen unterstützen

Im C-Laufzeitsystem gibt es zu jeder Funktion, die mit Gleitpunktzahlen arbeitet oder eine Gleitpunktzahl zurückliefert,

– eine Implementierung der Funktion mit /390-Arithmetik,

– eine Implementierung der Funktion mit IEEE-Arithmetik,

– ein Makro-Define, das die Originalfunktion (/390-Funktion) auf die zugehörige IEEE-Funktion abbildet.

Prototyp einer IEEE-Funktion und zugehöriges Define sind in derselben Include-Datei abgelegt, in der auch die korrespondierende Originalfunktion deklariert ist. Dies hat den Vorteil, dass, außer ggf. <ieee_390.h> (siehe Seite 31), für die Nutzung der IEEE-Gleitpunkt-Arithmetik keine zusätzlichen Include-Dateien benötigt werden.

Namen der IEEE-Funktionen

Die Namen der IEEE-Funktionen sind nach folgender Syntax aufgebaut:

`__`*originalfunktion*`_ieee()`

Dabei ist für *originalfunktion* der Name der Originalfunktion einzusetzen.

Die IEEE-Variante von `sin()` beispielsweise lautet `__sin_ieee()`.

C-Bibliotheksfunktionen, zu denen es eine IEEE-Funktion gibt

Zu folgenden C-Bibliotheksfunktionen gibt es jeweils eine IEEE-Variante:

acos	asin	atan	atan2	atof	ceil
cos	cosh	difftime	difftime64	ecvt	erf
erfc	exp	fabs	fcvt	floor	fprintf
frexp	fscanf	gamma	gcvt	hypot	j0
j1	jn	ldexp	llrint	llrintf	llrintl
llround	llroundf	llroundl	log	log10	lrint
lrintf	lrintl	lround	lroundf	lroundl	modf
pow	printf	rint	rintf	rintl	round
roundf	roundl	scanf	sin	sinh	snprintf
sprintf	sqrt	sscanf	strtod	tan	tanh
vfprintf	vprintf	vsnprintf	vsprintf	y0	y1
yn					

2.5.3 Steuerung von Originalfunktionen auf die zugehörigen IEEE-Varianten

Mit dem Präprozessor-Define `_IEEE_SOURCE` legen Sie fest, ob die Original-Bibliotheksfunktionen (/390-Funktionen) für Gleitpunkt-Arithmetik auf die zugehörigen IEEE-Varianten abgebildet werden. Die Prototypen der IEEE-Funktionen werden in jedem Fall generiert.

`_IEEE_SOURCE` kann folgende Werte annehmen:

_IEEE_SOURCE == 0

Die /390-Funktionen werden nicht auf die korrespondierenden IEEE-Varianten abgebildet. Die parallele Verwendung von /390- und IEEE-Funktionen ist somit möglich. Diese Einstellung gilt unabhängig von den Einstellungen des Compilers (Define `_IEEE`, siehe Seite 28).

_IEEE_SOURCE == 1

Die /390-Funktionen werden auf die korrespondierenden IEEE-Varianten abgebildet. Die parallele Verwendung von /390- und IEEE-Funktionen ist nicht möglich. Diese Einstellung gilt unabhängig von den Einstellungen des Compilers (Define `_IEEE`, siehe Seite 28).

Mit dem Präprozessor-Define `_MAP_NAME` können Sie wählen, ob die Abbildung der /390-Funktionen auf die IEEE-Funktionen via Namens-Define-Technik oder via Makro-Define-Technik erfolgen soll (siehe Seite 43).

Wenn Sie die Abbildung der Originalfunktionen auf die zugehörigen IEEE-Funktionen via Präprozessor-Define steuern wollen, müssen Sie die Funktionsdeklarationen der Standard-Include-Dateien verwenden, d.h. Sie müssen die Standard-Include-Dateien inkludieren.

_IEEE_SOURCE ist nicht definiert

In diesem Fall wird in Abhängigkeit vom Compiler-Schalter (Define `_IEEE`, siehe Seite 28) wie folgt verfahren:

_IEEE == 0 oder _IEEE nicht definiert

Die /390-Funktionen werden nicht auf die korrespondierenden IEEE-Varianten abgebildet.

_IEEE == 1

Die /390-Funktionen werden auf die korrespondierenden IEEE-Varianten abgebildet.

Die Steuerung der Originalfunktionen auf die zugehörigen IEEE-Varianten setzt voraus, dass die Compiler-Option MODIFY-MODULE-PROPERTIES mit den folgenden Angaben spezifiziert wird:

```
MODIFY-MODULE-PROPERTIES          -
...
LOWER-CASE-NAMES=*YES,            -
SPECIAL-CHARACTERS=*KEEP,         -
...
```

Dadurch wird verhindert, dass

- die Namen der IEEE-Funktionen (siehe Seite 29) auf acht Zeichen gekürzt werden,
- in den Funktionsnamen Kleinbuchstaben in Großbuchstaben umgewandelt und die Zeichen „_“ durch „$“ ersetzt werden.

In POSIX spezifizieren Sie zu diesem Zweck:

```
-K llm_keep
-K llm_case_lower
```

2.5.4 Explizite Konvertierung von Gleitpunktzahlen

Neben den in den vorhergehenden Abschnitten genannten Compiler- und Laufzeitsystem-Erweiterungen für die IEEE-Unterstützung stehen auch Funktionen zur expliziten Konvertierung von Gleitpunktzahlen zwischen /390- und IEEE-Format zur Verfügung.

Folgende Konvertierungsfunktionen sind in der Include-Datei <ieee_390.h> deklariert:

```
extern float float2ieee(float num);
extern float ieee2float(float num);

extern double double2ieee(double num);
extern double ieee2double(double num);
```

Ausführlich beschrieben sind Konvertierungsfunktionen im alphabetischen Nachschlageteil (siehe Seite 119).

2.6 ASCII-Codierung

Neben der standardmäßigen EBCDIC-Codierung von Zeichen und Zeichenketten wird auch die ASCII-Codierung von Zeichen und Zeichenketten unterstützt:

– Es gibt eine Compiler-Option des C/C++-Compilers, mit der Zeichen und Zeichenketten im ASCII-Format erzeugt werden können (siehe Seite 33).

– Zu jeder Bibliotheksfunktion im C-Laufzeitsystem, die mit Zeichen oder Zeichenketten arbeitet oder ein Zeichen bzw. eine Zeichenkette zurückliefert, gibt es eine Variante für die Bearbeitung von ASCII-Zeichen(ketten) sowie ein Makro-Define, das die EBCDIC-Variante der Funktion auf die zugehörige ASCII-Variante abbildet (siehe Seite 36).

Per Compiler-Option können Sie die komplette ASCII-Funktionalität aktivieren: Der C/C++-Compiler erzeugt dann in allen Modulen Zeichen und Zeichenketten im ASCII-Format und stellt automatisch die passenden ASCII-Funktionen zur Bearbeitung der ASCII-Zeichen(ketten) bereit.

Daneben haben Sie die Möglichkeit, die zur Verfügung gestellte ASCII-Funktionalität modifiziert zu nutzen:

– Mithilfe des Präprozessor-Defines `_ASCII_SOURCE` können Sie festlegen, ob die Bibliotheksfunktionen für EBCDIC-Darstellung auf die zugehörigen ASCII-Varianten abgebildet werden (siehe Seite 36).

– Mithilfe von Konvertierungsfunktionen können Sie ASCII-Zeichen und -Zeichenketten explizit vom EBCDIC-Format in das ASCII-Format konvertieren (siehe Seite 38).

2.6.1 Erzeugen von ASCII-Zeichen und -Zeichenketten via Compiler-Option

Code für Zeichen und Zeichenketten erzeugt der C/C++-Compiler wahlweise im EBCDIC-Format (Standard) oder ASCII-Format. Das gewünschte Format legen Sie mit der Option LITERAL-ENCODING der Compiler-Anweisung MODIFY-SOURCE-PROPERTIES fest.

```
MODIFY-SOURCE-PROPERTIES ..., LITERAL-ENCODING=*NATIVE|*ASCII-FULL
```

LITERAL-ENCODING=*NATIVE
: Der Compiler erzeugt Code für Zeichen und Zeichenketten im EBCDIC-Format. *NATIVE ist Standard.

LITERAL-ENCODING=*ASCII-FULL
: Der Compiler erzeugt Code für Zeichen und Zeichenketten im ASCII-Format. Außerdem wird das Präprozessor-Define `_LITERAL_ENCODING_ASCII` auf 1 gesetzt. Sofern nicht das Präprozessor-Define `_ASCII_SOURCE` auf 0 gesetzt ist (siehe Seite 36), werden die EBCDIC-Bibliotheksfunktionen somit automatisch auf die zugehörigen ASCII-Funktionen gesteuert.

In POSIX legen Sie die ASCII-Codierung mit der folgenden Option fest:

```
-K literal_encoding_ascii_full
```

Wenn Sie ASCII-Unterstützung nutzen wollen, müssen Sie die Compiler-Anweisung MODIFY-MODULE-PROPERTIES mit den folgenden Angaben spezifizieren:

```
MODIFY-MODULE-PROPERTIES          -
...
LOWER-CASE-NAMES=*YES,            -
SPECIAL-CHARACTERS=*KEEP,         -
...
```

Dadurch wird verhindert, dass

- die Namen der ASCII-Funktionen (siehe Seite 34) auf acht Zeichen gekürzt werden,
- in den Funktionsnamen Kleinbuchstaben in Großbuchstaben umgewandelt und die Zeichen „_“ durch „$“ ersetzt werden.

In POSIX spezifizieren Sie zu diesem Zweck:

```
-K llm_keep
-K llm_case_lower
```

Parameterübergabe, Umgebungsvariablen und globale Variable *tzname*

Die Option LITERAL-ENCODING legt auch das Format fest, in dem diese Zeichenketten an die main-Funktion übergeben werden. Bei LITERAL-ENCODING= *ASCII-FULL werden die genannten Zeichenketten also standarmäßig im ASCII-Format an die main-Funktion übergeben. Sie können Anwendungen, die ins BS2000 portiert oder ursprünglich als EBCDIC-Anwendungen erstellt wurden, somit ohne Eingriffe in den Source-Code als ASCII-Anwendungen produzieren.

2.6.2 C-Bibliotheksfunktionen, die ASCII-Codierung unterstützen

Zu jeder Bibliotheksfunktion im C-Laufzeitsystem, die mit Zeichen und/oder Zeichenketten arbeitet oder ein Zeichen bzw. eine Zeichenkette zurückliefert (z.B. `printf`), gibt es

– eine Implementierung der Funktion für die Bearbeitung von Zeichen und/oder Zeichenketten im EBCDIC-Format,

– eine Implementierung der Funktion für die Bearbeitung von Zeichen und/oder Zeichenketten im ASCII-Format,

– ein Makro-Define, das die Originalfunktion (EBCDIC-Format) auf die zugehörige ASCII-Funktion abbildet.

Prototyp einer ASCII-Funktion und zugehöriges Define sind in derselben Include-Datei abgelegt, in der auch die korrespondierende Originalfunktion deklariert ist. Dies hat den Vorteil, dass, außer ggf. <ascii_ebcdic.h> (siehe Seite 38), für die Nutzung der ASCII-Codierung von Zeichen und Zeichenketten keine zusätzlichen Include-Dateien benötigt werden.

Namen der ASCII-Funktionen

Die Namen der ASCII-Funktionen sind nach folgender Syntax aufgebaut:

`__`*originalfunktion*`_ascii()`

Dabei ist für *originalfunktion* der Name der Originalfunktion einzusetzen.

Die ASCII-Variante von `printf()` beispielsweise lautet `__printf_ascii()`.

C-Bibliotheksfunktionen, zu denen es eine ASCII-Funktion gibt

Zu folgenden C-Bibliotheksfunktionen gibt es jeweils eine ASCII-Variante:

asctime	assert	atof	atoi	atol
atoll	basename	bs2exit	bs2fstat	creat
creat64	ctime	ctime64	ecvt	fdopen
fgetc	fgets	fopen	fopen64	fprintf
fputc	fputs	fread	freopen	freopen64
fscanf	fwrite	gcvt	getc_unlocked	getenv
getpgmname	gets	gettsn	isalnum	isalpha
isascii	iscntrl	isdigit	isgraph	islower
isprint	ispunct	isspace	isupper	localeconv
mknod	mktemp	open	open64	perror
printf	remove	rename	scanf	setlocale
snprintf	sprintf	sscanf	strerror	strlower
strptime	strtod	strtol	strtoll	strtoul
strtoull	strupper	tmpnam	tolower	toupper
ungetc	vfprintf	vsnprintf	vsprintf	

2.6.3 Steuerung von Originalfunktionen auf die zugehörigen ASCII-Varianten

Mit dem Präprozessor-Define `_ASCII_SOURCE` legen Sie fest, ob die Original-Bibliotheksfunktionen (EBCDIC-Funktionen) für Zeichen-/Zeichenketten-Verarbeitung auf die zugehörigen ASCII-Varianten abgebildet werden. Die Prototypen der ASCII-Funktionen werden in jedem Fall generiert.

`_ASCII_SOURCE` kann folgende Werte annehmen:

_ASCII_SOURCE == 0

Die EBCDIC-Funktionen werden nicht auf die korrespondierenden ASCII-Varianten abgebildet. Die parallele Vewendung von EBCDIC- und ASCII-Funktionen ist somit möglich. Diese Einstellung gilt unabhängig von den Einstellungen des Compilers (Define `_ASCII`, siehe Seite 33).

_ASCII_SOURCE == 1

Die EBCDIC-Funktionen werden auf die korrespondierenden ASCII-Varianten abgebildet. Die parallele Verwendung von EBCDIC- und ASCII-Funktionen ist nicht möglich. Diese Einstellung gilt unabhängig von den Einstellungen des Compilers (Define `_LITERAL_ENCODING_ASCII`, siehe Seite 33).

Mit dem Präprozessor-Define `_MAP_NAME` können Sie wählen, ob die Abbildung der EBCDIC-Funktionen auf die ASCII-Funktionen via Namens-Define-Technik oder via Makro-Define-Technik erfolgen soll (siehe Seite 43).

Wenn Sie die ASCII-Funktionen via Präprozessor-Define nutzen wollen, müssen Sie die Funktionsdeklarationen der Standard-Include-Dateien verwenden, d.h. Sie müssen die Standard-Include-Dateien inkludieren.

_ASCII_SOURCE ist nicht definiert

In diesem Fall wird in Abhängigkeit von den Einstellungen des Compilers (Define `_LITERAL_ENCODING_ASCII`, siehe Seite 33) wie folgt verfahren:

LITERAL_ENCODING_ASCII == 0 oder
LITERAL_ENCODING_ASCII nicht definiert

Die Originalfunktionen werden nicht auf die korrespondierenden ASCII-Varianten abgebildet.

LITERAL_ENCODING_ASCII == 1

Die Originalfunktionen werden auf die korrespondierenden ASCII-Varianten abgebildet.

Die Steuerung der EBCDIC-Funktionen auf die zugehörigen ASCII-Funktionen setzt voraus, dass die Compiler-Option MODIFY-MODULE-PROPERTIES mit den folgenden Angaben spezifiziert wird:

```
MODIFY-MODULE-PROPERTIES          -
...
LOWER-CASE-NAMES=*YES,            -
SPECIAL-CHARACTERS=*KEEP,         -
...
```

Dadurch wird verhindert, dass

- die Namen der ASCII-Funktionen (siehe Seite 34) auf acht Zeichen gekürzt werden,
- in den Funktionsnamen Kleinbuchstaben in Großbuchstaben umgewandelt und die Zeichen „_“ durch „$“ ersetzt werden.

In POSIX spezifizieren Sie zu diesem Zweck:

```
-K llm_keep
-K llm_case_lower
```

2.6.4 Expliziter Wechsel zwischen EBCDIC- und ASCII-Codierung

Neben den in den vorhergehenden Abschnitten genannten Compiler- und Laufzeitsystem-Erweiterungen für die ASCII-Unterstützung gibt es Funktionen zur expliziten Konvertierung von Zeichen und Zeichenketten zwischen EBCDIC- und ASCII-Darstellung. Dies ermöglicht das Mischen von EBCDIC- und ASCII-Darstellung innerhalb eines Moduls. Die Konvertierungsfunktionen sind in der Include-Datei <ascii_ebcdic.h> deklariert.

Folgende Konvertierungsfunktionen und Daten stehen zur Verfügung:

```
char *_a2e(char *str);
char *_e2a(char *str);

char *_a2e_n(char *str, size_t n);
char *_e2a_n(char *str, size_t n);

char *_a2e_max(char *str, size_t n);
char *_e2a_max(char *str, size_t n);

char *_a2e_dup(const char *str);
char *_e2a_dup(const char *str);

char *_a2e_dup_n(const char *str, size_t n);
char *_e2a_dup_n(const char *str, size_t n);
```

Ausführlich beschrieben sind Konvertierungsfunktionen im alphabetischen Nachschlageteil (siehe Seite 119).

2.7 Funktionen, die IEEE und ASCII-Codierung unterstützen

Die Include-Dateien <stdio.h> und <stdlib.h> des C-Laufzeitsystems enthalten einige Funktionen, die sowohl IEEE-Gleitpunkt-Arithmetik, als auch ASCII-Codierung unterstützen.

Die Originalfunktionen (/390, EBCDIC) werden auf die korrespondierenden ASCII/IEEE-Funktionen abgebildet, wenn die Präprozessor-Defines `_IEEE_SOURCE` (siehe Seite 30) und `_ASCII_SOURCE` (siehe Seite 36) gleichzeitig auf den Wert 1 gesetzt sind.

Namen der ASCII/IEEE-Funktionen

Die Namen dieser ASCII/IEEE-Funktionen sind nach folgender Syntax aufgebaut:

`__`*originalfunktion*`_ascii_ieee()`

Dabei ist für *originalfunktion* der Name der Originalfunktion einzusetzen.

Die ASCII/IEEE-Variante von `printf()` beispielsweise lautet `__printf_ascii_ieee()`.

C-Bibliotheksfunktionen, zu denen es eine ASCII/IEEE-Funktion gibt

Zu folgenden C-Bibliotheksfunktionen gibt es jeweils eine ASCII/IEEE-Variante:

atof	ecvt	fcvt	fprintf	fscanf	gcvt
fprintf	fscanf	gcvt	printf	scanf	snprintf
sprintf	sscanf	srtod	vfprintf	vsnprintf	vsprintf

2.8 Langzeichen und Multibyte-Zeichen

Langzeichen und Multibyte-Zeichen wurden definiert, um das ursprüngliche „Zeichen“-Konzept der Computersprachen zu erweitern, das die Zuordnung eines Zeichens zu einem Byte Speicherplatz vorsah. Diese Zuordnung reichte jedoch für Sprachen wie zum Beispiel Japanisch nicht aus, da die Darstellung eines Zeichens in diesen Sprachen mehr als ein Byte Speicherplatz erfordert. Aus diesem Grunde wurde das Zeichen-Konzept um Multibyte-Zeichen und Langzeichen erweitert.

Multibyte-Zeichen stellen Zeichen des erweiterten Zeichensatzes in zwei, drei oder mehr Bytes dar.
Multibyte-Zeichenketten können „Shift Sequenzen“ enthalten, die die Bedeutung der nachfolgenden Multibyte-Codes verändern. Shift-Sequenzen können zum Beispiel umschalten zwischen verschiedenen Interpretationsmodi: Die ein-byte Shift-Sequenz `0200` kann festlegen, dass nachfolgende Byte-Paare als japanische Zeichen interpretiert werden, bzw. die Shift-Sequenz `0201`, dass nachfolgende Byte-Paare als Zeichen des ISO-Latin-1-Zeichensatzes interpretiert werden.

Programmier-Modell

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

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

Hinweise zu Langzeichen

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

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

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

Hinweise zu dieser Implementierung

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

2.9 Zeitfunktionen

Die in diesem Handbuch beschriebenen Zeitfunktionen verwenden für die Umrechnung in Sekunden standardmäßig den 1.1.1950 00:00:00 als Stichtag (Epoche).

Dies hat zur Folge, dass die Zeitfunktionen `ctime`, `difftime`, `ftime`, `gmtime`, `localtime`, `mktime` und `time` ab dem Jahr 2018 nicht mehr funktionieren. Das Jahr 2018 stellt somit für C-Anwendungen im BS2000 ein größeres Problem dar als der Jahrtausendwechsel.

Der Bindeschalter TIMESHIFT verlegt den Stichtag (Epoche) auf den 1.1.1970 00:00:00. Dadurch liefern die genannten Zeitfunktionen ohne Eingriffe ins Quellprogramm bis zum 19.1.2038 03:14:07 korrekte Ergebnisse, verhalten sich ansonsten aber exakt so wie ohne Einsatz des Bindeschalters.

Der Bindeschalter ist in den Bibliotheken
– SYSLNK.CRTE.TIMESHIFT,
– SKULNK.CRTE.TIMESHIFT bzw.
– SPULNK.CRTE.TIMESHIFT
enthalten.

Geben Sie beim Binden die folgende Anweisung an, um den Bindeschalter zu verwenden:

```
INCLUDE-MODULE LIB=<bibliothek>,ELEMENT=*ALL
```

Zeitfunktionen mit 64-bit-Zeitzähler

Die Funktionen `ctime64`, `difftime64`, `ftime64`, `gmtime64`, `localtime64`, `mktime64`, `time64` verwenden einen 64-bit-Zeitzähler. Sie liefern deshalb im Gegensatz zu den entsprechenden Funktionen mit 32-bit-Zeitzähler (`ctime`, `difftime`, `ftime`, `gmtime`, `localtime`, `mktime`, `time`) bis zum 18.3.4317 02:44:48 korrekte Ergebnisse. Bei den Zeitfunktionen mit 64-bit-Zeitzähler ist der Stichtag unabhängig vom Einsatz des TIMESHIFT-Bindeschalters immer der 1.1.1970 00:00:00.

2.10 Präprozessor-Define _STRICT_STDC

Die Standard-Includes enthalten die Prototyp-Deklarationen für alle C-Bibliotheksfunktionen, die das C-Laufzeitsystem zur Verfügung stellt. Ca. 50 dieser Bibliotheksfunktionen sind nicht im ANSI-Standard definiert, sondern stellen BS2000-spezifische Erweiterungen (z.B. `bs2fstat`, `_edt`) oder UNIX-spezifische Erweiterungen (z.B. `open`, `gamma`) dar.

Um die Programmierung ANSI-konformer Applikationen zu ermöglichen, wird das Define `_STRICT_STDC` zur Verfügung gestellt.
Dieses Define wird zum Übersetzungszeitpunkt mit folgender Option gesetzt:

– Mit dem C- und C++-Compiler V2.2

```
SOURCE-PROPERTIES = PAR(LANGUAGE-STANDARD = STRICT-ANSI)
```

– Mit dem C/C++-Compiler ab V3.0

```
MODIFY-SOURCE-PROPERTIES LANGUAGE=*C(MODE=*STRICT-ANSI)
```

Wenn das Define `_STRICT_STDC` gesetzt ist, werden die Prototyp-Deklarationen für alle nicht im ANSI-Standard definierten Funktionen in den Standard-Includes ausgeschaltet bzw. umgangen. Die Namen dieser Funktionen sind dann als benutzereigene Namen frei verfügbar.

Das Define `_STRICT_STDC` bezieht sich nur auf die Prototyp-Deklarationen innerhalb von ANSI-definierten Standard-Includes. Die BS2000-spezifischen Include-Header enthalten keine Abfrage dieses Defines.

Im Anhang (siehe Seite 537 ff) sind alle Funktionen aufgelistet, die das C-Laufzeitsystem zur Verfügung stellt. Bei jeder Funktion wird angegeben, ob sie im ANSI-Standard definiert ist oder eine Erweiterung darstellt.

2.11 Präprozessor-Define für Funktions-Prototypen gemäß XPG4

Die folgenden Funktionen sind im XPG4-Standard und im Amendment 1 unterschiedlich definiert:

`fputwc`, `putwc`, `putwchar`, `wcschr`, `wcsrchr`, `wcstok`

Mit Hilfe der Präprozessor-Defines `_XOPEN_SOURCE_EXTENDED` und `_XOPEN_SOURCE` können Sie steuern, ob der Prototyp der Funktion gemäß XPG4 oder Amendment 1 bereitgestellt wird.

Setzen Sie die Präprozessor-Defines `_XOPEN_SOURCE_EXTENDED` und `_XOPEN_SOURCE` nicht, wird der Prototyp gemäß Amendment 1 bereitgestellt.

Setzen Sie `_XOPEN_SOURCE_EXTENDED` oder `_XOPEN_SOURCE`, wird der Prototyp gemäß XPG4 bereitgestellt.

2.12 Präprozessor-Define _MAP_NAME

Bei der Nutzung der IEEE-Gleitpunkt-Arithmetik, der ASCII-Codierung sowie der 64-Bit-Schnittstellen für große Dateien (> 2 GB) können bestimmte C-Bibliotheksfunktionen durch die entsprechenden IEEE-, ASCII- bzw. 64-Bit-Varianten dieser Funktionen ersetzt werden.

Mithilfe des Präprozessor-Defines `_MAP_NAME` können Sie festlegen, ob die Ersetzung via Namens- oder via Makro-Define-Technik erfolgen soll:

– Wenn Sie `_MAP_NAME` definieren, wird die Namens-Define-Technik verwendet.
– Wenn Sie `_MAP_NAME` nicht definieren, wird die Makro-Define-Technik verwendet.

Die Namens-Define-Technik definiert via #define-Anweisung einen Makro ohne Argumente, während die Makro-Define-Technik via #define-Anweisung einen Makro mit Argumentenliste definiert.

Welche Lösung zu bevorzugen ist, hängt vom konkreten Anwenderprogramm ab. So werden bei der Makro-Define-Technik Zeiger auf Funktionen nicht erfasst, während bei Namens-Defines inkorrekterweise auch Variablen umbenannt werden.

3 Thematische Zusammenstellung der Funktionen

In diesem Kapitel finden Sie eine Zusammenstellung der Funktionen nach inhaltlichen Gesichtspunkten. Jede Funktion kommt genau einmal vor.

3.1 Dateiverarbeitung

Im Folgenden sind alle Ein-/Ausgabefunktionen als „elementar“ gekennzeichnet, die auf Basis von Dateikennzahlen arbeiten und nicht, wie die Standard-Ein-/Ausgabefunktionen, auf Basis von Dateizeigern. Außerdem finden Sie einen Hinweis, wenn die Funktion auch auf Dateien mit satzorientierter Ein-/Ausgabe (Satz-E/A) anwendbar ist.

Dateizugriffe (Öffnen, Schließen, Positionieren)

Name	Kurzbeschreibung
close	Datei schließen und Puffer bereinigen (elementar)
creat	neue Datei anlegen oder vorhandene überschreiben (elementar)
creat64	64-Bit-Variante von `creat` zur Bearbeitung von großen Dateien (> 2 GB)
fclose	Datei schließen und Puffer bereinigen (auch Satz-E/A)
fdelrec	Satz in indexsequenzieller Datei löschen (nur Satz-E/A)
fdopen	Dateizeiger einer Dateikennzahl zuweisen
fflush	Dateipuffer bereinigen
fgetpos	aktuelle Position des Lese-/Schreibzeigers ermitteln (auch Satz-E/A)
fgetpos64	64-Bit-Variante von `fgetpos` zur Bearbeitung von großen Dateien (> 2 GB)
flocate	indexsequenzielle Datei explizit positionieren (nur Satz-E/A)
fopen	Datei öffnen (auch Satz-E/A)
fopen64	64-Bit-Variante von `fopen` zur Bearbeitung von großen Dateien (> 2 GB)
freopen	Dateizeiger neu zuweisen (auch Satz-E/A)
freopen64	64-Bit-Variante von `freopen` zur Bearbeitung von großen Dateien (> 2 GB)
fseek	Lese-/Schreibzeiger positionieren (auch Satz-E/A)
fseek64	64-Bit-Variante von `fseek` zur Bearbeitung von großen Dateien (> 2 GB)
fsetpos	Lese-/Schreibzeiger positionieren (auch Satz-E/A)
fsetpos64	64-Bit-Variante von `fsetpos` zur Bearbeitung von großen Dateien (> 2 GB)
ftell	aktuelle Position des Lese-/Schreibzeigers ermitteln
ftell64	64-Bit-Variante von `ftell` zur Bearbeitung von großen Dateien (> 2 GB)
lseek	Lese-/Schreibzeiger positionieren (elementar)
lseek64	64-Bit-Variante von `lseek` zur Bearbeitung von großen Dateien (> 2 GB)
fwide	Orientierung einer Datei abfragen/festlegen
open	Datei öffnen (elementar)
open64	64-Bit-Variante von `open` zur Bearbeitung von großen Dateien (> 2 GB)
rewind	Lese-/Schreibzeiger auf Dateianfang positionieren (auch Satz-E/A)
setbuf	Ein-/Ausgabe-Puffer einrichten
setvbuf	Ein-/Ausgabe-Puffer einrichten
tell	aktuelle Position des Lese-/Schreibzeigers abfragen (elementar)

Dateiverwaltung

Name	Kurzbeschreibung
bs2fstat	Zugriff auf Dateinamen aus dem Katalog
mktemp	eindeutigen temporären Dateinamen erzeugen
remove	Datei löschen (auch Satz-E/A)
rename	Datei umbenennen (auch Satz-E/A)
tmpfile	temporäre Binärdatei öffnen
tmpfile64	64-Bit-Variante von tmpfile zu Bearbeitung großer Dateien (> 2 GB)
tmpnam	temporären Dateinamen erzeugen
unlink	Datei löschen (auch Satz-E/A)

Datei-/Fehlerinformation

Name	Kurzbeschreibung
clearerr	Dateiende- und Fehlerflag löschen (auch Satz-E/A)
feof	Test auf Dateiende (auch Satz-E/A)
ferror	Test auf Dateifehler (auch Satz-E/A)

Ein-/Ausgabe

Name	Kurzbeschreibung
fgetc	Zeichen aus einer Datei einlesen
fgets	Zeichenkette aus einer Datei einlesen
fgetwc	Langzeichen aus einer Datei einlesen
fgetws	Langzeichenkette aus einer Datei einlesen
fprintf	formatierte Ausgabe in eine Datei
fputc	Zeichen in eine Datei schreiben
fputs	Zeichenkette in eine Datei schreiben
fputwc	Langzeichen in eine Datei schreiben
fputws	Langzeichenkette in eine Datei schreiben
fread	blockweise aus einer Datei einlesen (auch Satz-E/A)
fscanf	formatierte Eingabe aus einer Datei
fwprintf	Zeichen formatiert (Langzeichen-Format) in Ausgabedatei schreiben
fwrite	blockweise in eine Datei schreiben (auch Satz-E/A)
fwscanf	formatierte Eingabe (Langzeichen-Format) aus einer Datei
getc	Zeichen aus einer Datei einlesen
getchar	Zeichen von Standardeingabe einlesen
gets	Zeichenkette von Standardeingabe einlesen
getw	wortweise aus einer Datei einlesen
getwc	Langzeichen aus einer Datei lesen
getwchar	Langzeichen von Standardeingabe lesen

Name	Kurzbeschreibung
printf	formatierte Ausgabe auf Standardausgabe
putc	Zeichen in eine Datei schreiben
putchar	Zeichen auf Standardausgabe ausgeben
puts	Zeichenkette auf Standardausgabe ausgeben
putw	wortweise in eine Datei schreiben
putwc	Langzeichen in eine Datei schreiben
putwchar	Langzeichen auf Standardausgabe schreiben
read	aus einer Datei einlesen (elementar)
scanf	formatierte Ausgabe (Langzeichen-Format) in eine Datei
snprintf	Zeichen formatiert (Langzeichen-Format) in eine Datei schreiben
sprintf	formatierte Ausgabe auf Standardausgabe
sscanf	formatierte Ausgabe in eine Zeichenkette
swprintf	formatierte Ausgabe in eine Zeichenkette
swscanf	Zeichen formatiert in Langzeichenkette schreiben
ungetc	Zeichen formatiert (Langzeichen-Format) auf Standardausgabe ausgeben
ungetwc	Zeichen formatiert (Langzeichen-Format) auf Standardausgabe ausgeben
vfprintf	in eine Datei schreiben (elementar)
vfwprintf	formatierte Eingabe (Langzeichen-Format) von Standardeingabe
vprintf	formatierte Eingabe von Standardeingabe
vsnprintf	formatierte Ausgabe in eine Zeichenkette
vsprintf	formatierte Ausgabe in eine Zeichenkette
vswprintf	formatierte Eingabe aus einer Zeichenkette
vwprintf	Zeichen formatiert in Langzeichenkette schreiben
wprintf	formatierte Eingabe aus einer Langzeichenkette
write	Zeichen in den Puffer zurückstellen
wscanf	Langzeichen in den Puffer zurückstellen

3.2 Kommunikation mit der Systemumgebung

Name	Kurzbeschreibung
cputime	verbrauchte CPU-Zeit der aktuellen Task
environ	externe Variable für die Umgebung
getenv	Wert einer Umgebungsvariablen ermitteln
getlogin	Benutzerkennung ermitteln
gettsn	TSN-Nummer ermitteln
putenv	Umgebungsvariable ändern oder hinzufügen
system	Systemkommando ausführen
_edt	Dateibearbeiter EDT aufrufen

3.3 Programminformationen und Programmablaufsteuerung

Programm-Informationen

Name	Kurzbeschreibung
getpgmname	Namen des Programms ermitteln
_ _DATE_ _	Ausgabe des Übersetzungsdatums (Makro)
_ _FILE_ _	Ausgabe des Quelldateinamens (Makro)
_ _LINE_ _	Ausgabe der aktuellen Quellprogramm-Zeilennummer (Makro)
_ _TIME_ _	Ausgabe der Übersetzungsuhrzeit (Makro)
_ _STDC_ _	ANSI-Sprachstandard (Makro)
_ _STDC_VERSION_ _	Amendment 1-konform?

Programmbeendigung

Name	Kurzbeschreibung
abort	abnormaler Programmabbruch
atexit	Beendigungsroutinen registrieren
bs2exit	Programmbeendigung (mit MONJV)
exit und _exit	Programmbeendigung

Behandlung von Ausnahmebedingungen, Ereignissteuerung

Name	Kurzbeschreibung
alarm	Alarmuhr stellen
cdisco	Abmelden einer Contingency-Routine
cenaco	Definition einer Contingency-Routine
cstxit und _cstxit	Definition einer STXIT-Routine
kill	Signal an eigenes Programm senden
raise	Signal an eigenes Programm senden
signal	Signalbearbeitung steuern
sleep	Programm für festgesetzte Zeitspanne anhalten

Nicht lokale Sprünge

Name	Kurzbeschreibung
setjmp	Marke für nicht lokale Sprünge setzen
longjmp	nicht lokaler Sprung

Programm-Diagnose

Name	Kurzbeschreibung
assert	Makro zur Fehlerdiagnose

3.4 Speicherverwaltung

Name	Kurzbeschreibung
calloc	Speicherplatz für einen Vektor reservieren
free	Speicherplatz freigeben
garbcoll	Speicherplatz an das System freigeben (Garbage Collection)
malloc	Speicherplatz reservieren
memalloc	Speicherplatz reservieren (größer 2 KByte)
memfree	Speicherplatz freigeben (größer 2 KByte)
realloc	Speicherplatz verändern

3.5 Zeichenbearbeitung

Zeichen prüfen

Name	Kurzbeschreibung
isalnum	alphanumerisches Zeichen?
isalpha	Buchstabe?
isascii	EBCDIC-Zeichen?
iscntrl	Kontrollzeichen?
isdigit	Ziffer?
isebcdic	EBCDIC-Zeichen?
isgraph	abdruckbares Zeichen exklusive Leerzeichen?
islower	Kleinbuchstabe?
isprint	abdruckbares Zeichen inklusive Leerzeichen?
ispunct	Sonderzeichen?
isspace	Zwischenraum?
isupper	Großbuchstabe?
isxdigit	sedezimales Zeichen?

Langzeichen prüfen

Name	Kurzbeschreibung
iswalnum	alphanumerisches Langzeichen?
iswcntrl	Kontroll-Langzeichen?
iswctype	Langzeichen in Zeichenklasse *chartype*?
iswdigit	Langzeichen-Ziffer?
iswgraph	abdruckbares Langzeichen exclusive Leerzeichen?
islower	Langzeichen-Kleinbuchstabe?
iswprint	abdruckbares Langzeichen inklusive Leerzeichen?
iswpunct	Sonderlangzeichen?
iswspace	Zwischenraum-Langzeichen?
iswupper	Langzeichen-Großbuchstabe?
iswxdigit	heaxadezimales Langzeichen?

Zeichen umwandeln

Name	Kurzbeschreibung
toascii	Umwandlung in EBCDIC-Zeichen
toebcdic	Umwandlung in EBCDIC-Zeichen
tolower	Umwandlung in Kleinbuchstaben
toupper	Umwandlung in Großbuchstaben

Langzeichen umwandeln

Name	Kurzbeschreibung
wcrtomb	Langzeichen in Multibyte-Zeichen umwandeln
wctob	Langzeichen in (ein-byte) Multibyte-Zeichen konvertieren
towlower	Langzeichen in Kleinbuchstaben umwandeln
towupper	Langzeichen in Großbuchstaben umwandeln

3.6 Zeichenketten und Zeichenvektoren bearbeiten

Zeichenketten

Name	Kurzbeschreibung
index	erstes Vorkommen eines Zeichens in einer Zeichenkette
rindex	letztes Vorkommen eines Zeichens in einer Zeichenkette
strcat	Verketten von zwei Zeichenketten
strchr	erstes Vorkommen eines Zeichens in einer Zeichenkette
strcmp	Vergleichen von zwei Zeichenketten
strcoll	Vergleichen von zwei Zeichenketten
strcpy	Kopieren einer Zeichenkette in eine andere
strcspn	Länge des Segments einer Zeichenkette berechnen, das kein Zeichen aus einer zweiten Zeichenkette enthält
strfill	Kopieren einer Zeichenkette in eine andere bis zur Länge n und ggf. mit Leerzeichen auffüllen
strlen	aktuelle Länge einer Zeichenkette berechnen
strlower	Kopieren einer Zeichenkette in eine andere mit Umwandlung der Groß- in Kleinbuchstaben
strncat	Verketten von zwei Zeichenketten bis zur Länge n
strncmp	Vergleichen von zwei Zeichenketten bis zur Länge n
strncpy	Kopieren einer Zeichenkette in eine andere bis zur Länge n
strpbrk	erstes Vorkommen eines Zeichens in einer Zeichenkette, das mit einem Zeichen aus einer zweiten Zeichenkette übereinstimmt
strrchr	letztes Vorkommen eines Zeichens in einer Zeichenkette
strspn	Länge des Segments einer Zeichenkette berechnen, das ausschließlich Zeichen aus einer zweiten Zeichenkette enthält
strstr	erstes Vorkommen einer Zeichenkette in einer anderen
strtok	Zeichenkette in mehrere Teilzeichenketten zerlegen
strtok_r	threadsichere Variante von `strtok`
strupper	Kopieren einer Zeichenkette in eine andere mit Umwandlung der Klein- in Großbuchstaben
strxfrm	Transformieren einer Zeichenkette

Zeichenvektoren (Speicherbereiche)

Name	Kurzbeschreibung
memchr	erstes Vorkommen eines Zeichens in einem Speicherbereich
memcmp	Vergleichen von zwei Speicherbereichen
memcpy	Kopieren eines Speicherbereiches in einen anderen
memmove	Kopieren eines Speicherbereiches in einen anderen
memset	Initialisieren eines Speicherbereiches mit einem Zeichen

Langzeichenketten

Name	Kurzbeschreibung
towctrans	Langzeichen abbilden
wcsrtombs	Langzeichenkette in Multibyte-Zeichenkette umwandeln
wcsstr	erstes Vorkommen einer Langzeichenkette in einer Langzeichenkette
wctrans	Abbildung zwischen Langzeichen definieren
wcscat	Langzeichenketten zusammenfügen
wcschr	Langzeichenkette nach Langzzeichen durchsuchen
wcscmp	zwei Langzeichenkette vergleichen
wcscoll	zwei Langzeichenketten gemäß LC_COLLATE vergleichen
wcscpy	Langzeichenkette kopieren
wcscspn	Länge einer komplementären Langzeichenteilkette ermitteln
wcsftime	Datum und Uhrzeit in Langzeichenkette umwandeln
wcslen	Länge einer Langzeichenkette ermitteln
wcsncat	zwei Langzeichenteilketten zusammenfügen
wcsncmp	zwei Langzeichenteilketten vergleichen
wcsncpy	Langzeichenteilkette kopieren
wcspbrk	erstes Vorkommen eines Langzeichens in LLangzeichenkette ermitteln
wcsrchr	letztes Vorkommen eines Langzeichens in LLangzeichenkette ermittel
wcsspn	Länge einer Langzeichenteilkette ermitteln
wcstod	Langzeichenkette in Gleitkommazahl (double) umwandeln
wcstok	Langzeichenkette in Langzeichenteilketten zerlegen
wcstol	Langzeichenkette in ganze Zahl (long) umwandeln
wcstoul	Langzeichenkette in ganze Zahl (unsigned long) umwandeln
wcsxfrm	Langzeichenkette transformieren
wctype	Langzeichenklasse definieren

Langzeichenvektoren (Speicherbereiche)

Name	Kurzbeschreibung
wmemchr	erstes Vorkommen eines Langzeichens in einem Speicherbereich
wmemcmp	Vergleichen von zwei Langzeichenketten (Speicherbereichen)
wmemcpy	Langzeichenkette kopieren (ohne Überlappung der Speicherbereiche)
wmemmove	Langzeichenkette in überlappenden Bereich kopieren
wmemset	erste n Langzeichen in Langzeichenkette setzen

Multibyte-Funktionen

Name	Kurzbeschreibung
btowc	Multibyte-Zeichen in Langzeichen umwandeln
mblen	Byteanzahl eines Multibyte-Zeichens ermitteln
mbrlen	Restlänge eines Multibyte-Zeichens ermitteln
mbsinit	auf „initial conversion“-Zustand überprüfen
mbrtowcs	Multibyte-Zeichen vervollständigen und in Langzeichen konvertieren
mbstowcs	Umwandlung einer Multibyte-Zeichenkette in Langzeichen
mbtowc	Umwandlung eines Multibyte-Zeichens in ein Langzeichen
wcstombs	Umwandlung einer Langzeichenkette in eine Multibyte-Zeichenkette
wctomb	Umwandlung eines Langzeichens in ein Multibyte-Zeichen

3.7 Fehlermeldungen

Name	Kurzbeschreibung
perror	Standardfehlermeldung ausgeben
strerror	Fehlermeldungstext ermitteln

3.8 Zeitfunktionen

Name	Kurzbeschreibung
asctime	Datum mit Uhrzeit in Englisch
asctime_r	threadsichere Varianten von asctime
clock	CPU-Zeitverbrauch seit Programmaufruf
ctime	Datum mit Uhrzeit(MEZ) in Englisch
ctime64	Datum mit Uhrzeit(MEZ) in Englisch (Variante mit 64-Bit-Zeitzähler)
difftime	Zeitdifferenz berechnen
difftime64	Zeitdifferenz berechnen (Variante mit 64-Bit-Zeitzähler)
ftime	aktuelle Zeit(GMT) als Struktur
gmtime	Datum und aktuelle Ortszeit(MEZ) als Struktur
gmtime64	Datum und aktuelle Ortszeit(MEZ) als Struktur (Variante mit 64-Bit-Zeitzähler)
localtime	Datum und aktuelle Ortszeit (MEZ) als Struktur
localtime64	Datum und aktuelle Ortszeit (MEZ) als Struktur (Variante mit 64-Bit-Zeitzähler)
mktime	Umwandlung von Datum und Uhrzeit (Kalenderfunktion)
mktime64	Umwandlung von Datum und Uhrzeit (Variante mit 64-Bit-Zeitzähler)
strftime	lokalitätsspezifische Formatierung von Datum und Zeit
time	aktuelle Zeit(GMT) in Sekunden
time64	aktuelle Zeit(GMT) in Sekunden (Variante mit 64-Bit-Zeitzähler)

3.9 Mathematische Funktionen

Integer-Arithmetik

Name	Kurzbeschreibung
abs	Absolutbetrag (integer)
div	Division (integer)
labs	Absolutbetrag (long integer)
llabs	Absolutbetrag (long long integer)
ldiv	Division (long integer)
lldiv	Division (long long integer)

Gleitkommazahlen

Name	Kurzbeschreibung
acos	Arcuscosinus
asin	Arcussinus
atan	Arcustangens x
atan2	Arcustangens x/y
cabs	Absolutbetrag einer komplexen Zahl
ceil	ganzzahlig aufrunden
cos	Cosinus
cosh	Cosinus hyperbolicus
erf	Fehlerfunktion
erfc	Komplement der Fehlerfunktion
exp	Exponentialfunktion
fabs	Absolutbetrag einer Gleitkommazahl
floor	ganzzahlig abrunden
fmod	Rest einer Division
frexp	normierte Darstellung zur Basis 2
gamma	logarithmische Gammafunktion
hypot	Euklidischer Abstand
j0, j1, jn	Besselfunktionen der ersten Art
ldexp	Wert im Zweiersystem berechnen
log	natürlicher Logarithmus
log10	Logarithmus zur Basis 10
modf	Aufspalten in ganzzahligen und gebrochenen Teil
pow	allgemeine Exponentialfunktion
sin	Sinus
sinh	Sinus hyperbolicus
sqrt	Quadratwurzel
tan	Tangens
tanh	Tangens hyperbolicus
y0, y1, yn	Besselfunktionen der zweiten Art

Rundungsfunktionen (unabhängig vom Rundungsmodus)

Name	Kurzbeschreibung
llrint	Rundet Typ double auf nächste Ganzzahl vom Typ long long int
llrintf	Rundet Typ float auf nächste Ganzzahl vom Typ long long int
llrintl	Rundet Typ long double auf nächste Ganzzahl vom long long int
lrint	Rundet Typ double auf nächste Ganzzahl vom Typ long int
lrintf	Rundet Typ float auf nächste Ganzzahl vom Typ long int
lrintl	Rundet Typ long double auf nächste Ganzzahl vom long int
rint	Rundet Typ double auf nächste Ganzzahl vom Typ double
rintf	Rundet Typ float auf nächste Ganzzahl vom Typ float
rintl	Rundet Typ long double auf nächste Ganzzahl vom long double

Rundungsfunktionen (abhängig vom Rundungsmodus)

Name	Kurzbeschreibung
llround	Rundet Typ double auf nächste Ganzzahl vom Typ long long int
llroundf	Rundet Typ float auf nächste Ganzzahl vom Typ long long int
llroundl	Rundet Typ long double auf nächste Ganzzahl vom long long int
lround	Rundet Typ double auf nächste Ganzzahl vom Typ long int
lroundf	Rundet Typ float auf nächste Ganzzahl vom Typ long int
lroundl	Rundet Typ long double auf nächste Ganzzahl vom long int
round	Rundet Typ double auf nächste Ganzzahl vom Typ double
roundf	Rundet Typ float auf nächste Ganzzahl vom Typ float
roundl	Rundet Typ long double auf nächste Ganzzahl vom long double

3.10 Konvertierung von Größen

Name	Kurzbeschreibung
atof	Zeichenkette in Gleitkommazahl
atoi	Zeichenkette in integer
atol	Zeichenkette in long integer
atoll	Zeichenkette in long long integer
ecvt	Gleitkommawert in Zeichenkette
fcvt	Gleitkommawert in Zeichenkette
gcvt	Gleitkommawert in Zeichenkette
mbstowcs	Multibyte-Zeichenkette in Langzeichenkette umwandeln
strtod	Zeichenkette in Gleitkommazahl
strtol	Zeichenkette in long integer
strtoll	Zeichenkette in long long integer
strtoul	Zeichenkette in unsigned long integer
strtoull	Zeichenkette in unsigned long long integer
strptime	Zeichenkette in Datum und Uhrzeit umwandeln
wcsrtombs	Langzeichenkette in Multibyte-Zeichenkette umwandeln
wcsftime	Datum und Uhrzeit in Langzeichenkette umwandeln
wcstod	Langzeichenkette in Gleitkommazahl (double) umwandeln
wcstol	Langzeichenkette in ganze Zahl (long) umwandeln
wcstoll	Langzeichenkette in ganze Zahl (long long) umwandeln
wcstoul	Langzeichenkette in ganze Zahl (unsigned long) umwandeln
wcstoull	Langzeichenkette in ganze Zahl (unsigned long long) umwandeln

3.11 Sonstige Funktionen

Suchen und Sortieren

Name	Kurzbeschreibung
bsearch	binärer Suchalgorithmus
qsort	Quicksort

Zufallsgenerator

Name	Kurzbeschreibung
rand	Zufallsgenerator
rand_r	threadsichere Variante von `rand`
srand	Zufallsgenerator initialisieren

Lokalitäten

Name	Kurzbeschreibung
localeconv	Abfrage der lokalitätsspezifischen Daten
setlocale	Lokalität auswählen

Variable Argumentenlisten

Name	Kurzbeschreibung
va_arg	Variable Argumentenliste abarbeiten (Makro)
va_end	Variable Argumentenliste abschließen (Makro)
va_start	Variable Argumentenliste initialisieren (Makro)

Offset einer Strukturkomponente

Name	Kurzbeschreibung
offsetof	Abstand einer Strukturkomponente zum Strukturbeginn in Bytes (Makro)

4 Dateiverarbeitung

Mit den Ein-/Ausgabe-Funktionen des C-Laufzeitsystems können folgende Dateiarten verarbeitet werden:

– die BS2000-Systemdateien SYSDTA, SYSOUT und SYSLST,

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

– temporäre PAM-Dateien (INCORE).

Im C-BS2000 wird einerseits zwischen Binär- und Textdateien unterschieden, andererseits zwischen strom- und satzorientierter Ein-/Ausgabe (siehe auch Abschnitt „Grundbegriffe“ auf Seite 62).

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

	Textdatei Strom-E/A	Binärdatei Strom-E/A	Binärdatei Satz-E/A
Systemdateien	X		
INCORE		X	
SAM	X	X	X
ISAM	X		X
PAM		X	X

Es können (inkl. `stdin`, `stdout` und `stderr`) maximal 2048 Dateien gleichzeitig geöffnet sein.

4.1 Grundbegriffe

Dieses Kapitel enthält die ausführliche Erklärung von Dateiverarbeitungsbegriffen, die bei der Beschreibung der C-Ein-/Ausgabefunktionen im Nachschlageteil (Kapitel 7) häufig benutzt werden. Die Begriffe sind in alphabetischer Reihenfolge aufgeführt.

Binärdatei

Eine Binärdatei ist eine geordnete Folge von Bytes. Die mit den C-Ausgabefunktionen geschriebenen Daten werden 1:1 in die Datei übernommen. Im Unterschied zu Textdateien werden Steuerzeichen für Zeilenvorschub und Tabulatoren nicht in ihre Wirkung umgesetzt (siehe „Textdatei“), sondern als entsprechende EBCDIC-Werte abgebildet.
Daten, die aus einer Binärdatei eingelesen werden, entsprechen daher genau den Daten, die ursprünglich in die Datei geschrieben wurden.

Binärdateien mit Strom-E/A sind:

– katalogisierte PAM-Dateien,
– temporäre PAM-Dateien (INCORE),
– katalogisierte SAM-Dateien, die mit `fopen`/`fopen64` bzw. `freopen`/`freopen64` im Binärmodus eröffnet wurden.

Binärdateien mit Satz-E/A sind:

– katalogisierte ISAM-Dateien,
– katalogisierte SAM-Dateien,
– katalogisierte PAM-Dateien,

die mit den Funktionen `fopen`/`fopen64` bzw. `freopen`/`freopen64` im Binärmodus und mit dem Zusatz
"type=record" eröffnet wurden.

Der Binärmodus kann nur mit den Funktionen `fopen`/`fopen64` bzw. `freopen`/`freopen64` angegeben werden.
Mit den elementaren Funktionen `open`/`open64` und `creat`/`creat64` werden SAM- und ISAM-Dateien stets als Textdateien eröffnet.

Beim Arbeiten mit den ASCII-Varianten der Ein-/Ausgabefunktionen und Binärdateien ist Folgendes zu beachten:

Da die Daten mit C-Eingabefunktionen 1:1 in eine Binärdatei geschrieben und mit C-Ausgabefunktionen in genau diesem Format wieder ausgelesen werden, sind bei Programmen, die mit Binärdateien arbeiten, ggf. Anpassungen erforderlich. Dies ist u.a. bei der Verarbeitung von Textbestandteilen der Fall. Wurde beispielsweise bei einer ISAM-Datei der Schlüssel als EBCDIC-Zeichenkette abgelegt, dann ist sicherzustellen, dass bei einem Zeichenkettenvergleich nicht EBCDIC-Code mit ASCII-Code verglichen wird.

Dateikennzahl

Eine Dateikennzahl ist eine positive ganze Zahl. Sie dient dazu, eine Datei bei der Verarbeitung mit elementaren Zugriffsfunktionen zu identifizieren. Beim Öffnen (mit `open`/`open64`, `creat`/`creat64`) wird einer Datei eine Dateikennzahl zugewiesen. Bei allen weiteren Zugriffen (`read`, `write`, `close`, `tell` etc.) wird die Dateikennzahl als Dateiargument benutzt.

Bei Programmstart sind die Standard-Ein-/Ausgabedateien automatisch mit folgenden Dateikennzahlen geöffnet:

0 Standardeingabe

1 Standardausgabe

2 Standard-Fehlerausgabe

Dateizeiger

Ein Dateizeiger ist ein Zeiger auf eine Struktur vom Typ FILE. Er dient dazu, eine Datei mit den Standard-Zugriffsfunktionen (siehe <stdio.h>) zu verarbeiten. Beim Öffnen (mit `fopen`/`fopen64`, `fdopen`, `freopen`/`freopen64`) wird einer Datei ein Dateizeiger zugewiesen. Bei weiteren Zugriffen (`fprintf`, `fwprintf`, `fscanf`, `fclose` etc.) wird der Dateizeiger als Dateiargument benutzt.

Bei Programmstart sind die Standard-Ein-/Ausgabedateien automatisch mit folgenden Dateizeigern geöffnet:

`stdin` (Standardeingabe)

`stdout` (Standardausgabe)

`stderr` (Standard-Fehlerausgabe)

elementar

Als „elementar“ werden alle Funktionen bezeichnet, die eine Datei auf der Basis von Dateikennzahlen verarbeiten. Im Unterschied dazu gibt es die Standard-Ein-/Ausgabefunktionen, die alle auf der Basis von Dateizeigern arbeiten. Außerdem lassen sich mit den elementaren Funktionen SAM-Dateien nur als Textdateien und nicht, wie mit den Standardfunktionen, auch als Binärdateien verarbeiten.

In vielen anderen Implementierungen (z.B. UNIX, SINIX) sind die elementaren Funktionen als Systemaufrufe realisiert und unterscheiden sich von den Standardfunktionen durch größere Systemnähe und bessere Performance. Diesen Unterschied zwischen Systemaufruf und Funktion gibt es im BS2000 nicht.

FILE-Struktur

Einer Datei, die mit `fopen`/`fopen64`, `fdopen` oder `freopen`/`freopen64` geöffnet wird, ist automatisch ab diesem Zeitpunkt eine bestimmte Struktur vom Typ FILE zugeordnet. Diese Struktur ist in <stdio.h> definiert.

Sie enthält u.a. folgende Informationen über die Datei:

– Zeiger auf den Ein-/Ausgabepuffer,
– Puffergröße,
– Position des Lese-/Schreibzeigers,
– Größe der Datei.

Der von `fopen`/`fopen64`, `fdopen` oder `freopen`/`freopen64` gelieferte Dateizeiger verweist auf diese FILE-Struktur.

Lese-/Schreibzeiger

Der Lese-/Schreibzeiger enthält Informationen über die aktuelle Position einer Datei. Daten werden jeweils ab dieser aktuellen Position gelesen bzw. geschrieben.
Die Information im Lese-/Schreibzeiger ist je nach Dateiart unterschiedlich aufgebaut:

– Bei Binärdateien mit Strom-E/A entspricht sie der Anzahl Bytes vom Dateianfang gerechnet.

– Bei Textdateien enthält sie Informationen über den aktuellen Satz und die Position innerhalb des Satzes. Der Aufbau ist für SAM- und ISAM-Dateien unterschiedlich. Die Information wird vom Laufzeitsystem intern verwendet.

– Bei Binärdateien mit Satz-E/A entspricht sie der Position hinter dem zuletzt gelesenen, geschriebenen oder gelöschten Satz bzw. der Position, die durch ein unmittelbar vorangegangenes Positionieren erreicht wurde.

Bei ISAM-Dateien mit Schlüsselverdoppelung ist der Lese-/Schreibzeiger hinter dem letzten Satz einer Gruppe mit gleichen Schlüsseln positioniert, wenn einer dieser Sätze zuvor gelesen, geschrieben oder gelöscht wurde.

Pufferung

Bei allen Ausgabefunktionen, die Daten in Textdateien und Binärdateien mit Strom-E/A schreiben (`printf`, `putc`, `fwrite` etc.), werden die Daten in einem C-internen Puffer zwischengespeichert und erst in die externe Datei geschrieben, wenn ein bestimmtes Ereignis eintritt. Dieses unterscheidet sich bei Text- und Binärdateien.

Textdatei:

a) Ein Neue-Zeile-Zeichen (\n) wird erkannt,

b) die maximale Satzlänge einer Plattendatei ist erreicht,

c) bei Datensichtstationen: nach einer Ausgabe auf die Datensichtstation folgt eine Eingabe von der Datensichtstation,

d) die Positionier-Funktionen `fseek`/`fseek64`, `fsetpos`/`fsetpos64`, `rewind` oder `lseek`/`lseek64` werden aufgerufen,

e) die Funktion `fflush` wird aufgerufen; `fflush` wird intern automatisch ausgeführt, wenn eine Datei geschlossen wird (`fclose`, `close`) oder wenn ein Programm normal bzw. mit `exit` beendet wird.

f) die Datei wird geschlossen,

g) zusätzlich bei ANSI-Funktionalität: Wenn das Lesen aus einer beliebigen Textdatei eine Datenübertragung von der externen Datei in den C-internen Puffer notwendig macht, werden die noch in Puffern zwischengespeicherten Daten aller ISAM-Dateien automatisch in die Dateien hinausgeschrieben.

Auch wenn die Daten im Puffer nicht mit einem Neue-Zeile-Zeichen enden, bewirkt das Schreiben in die externe Datei einen Zeilenwechsel. Nachfolgende Daten werden in eine neue Zeile (bzw. in einen neuen Satz) geschrieben.

Ausnahme bei ANSI-Funktionalität:
Wenn die Daten einer ISAM-Datei im Puffer nicht mit einem Neue-Zeile-Zeichen enden, bewirkt das Schreiben in die externe Datei keinen Zeilenwechsel (bzw. Satzwechsel).

Nachfolgende Daten verlängern den Satz in der Datei. Beim Lesen einer ISAM-Datei werden daher nur Neue-Zeile-Zeichen eingelesen, die vom Programm explizit geschrieben wurden.

Binärdatei:

a) der Puffer ist voll,

b) die Positionier-Funktionen `fseek`/`fseek64`, `fsetpos`/`fsetpos64`, `rewind` oder `lseek`/`lseek64` werden aufgerufen,

c) die Funktion `fflush` wird aufgerufen (s.o. Textdatei)

d) die Datei wird geschlossen.

Die Zwischenpufferung entfällt bei INCORE-Dateien und generell bei Dateien mit Satz-E/A.

Satzorientierte-Ein-/Ausgabe

Satzorientierte Ein-/Ausgabe bedeutet, dass sich der Lese-/Schreibzeiger der Datei jeweils nur auf den Beginn eines Satzes (bzw. Blockes) positionieren lässt. Satzorientierte Ein-/Ausgabe ermöglicht eine der BS2000-Struktur angepasste performante Dateiverarbeitung. Die Einheit für einen Ein-/Ausgabe-Funktionsaufruf ist stets ein Satz (bzw. Block). Es stehen zusätzliche Funktionen zur Verfügung wie Löschen und Einfügen von Sätzen, Zugriff auf den Schlüssel in ISAM-Dateien.

Satzorientiert können katalogisierte SAM-, ISAM- und PAM-Dateien verarbeitet werden. Die Dateien müssen mit den Funktionen `fopen`/`fopen64` bzw. `freopen`/`freopen64` eröffnet werden, und zwar mit dem Zusatz "type=record" im *art*-Parameter und stets im Binärmodus.
U.a. sind Ein-/Ausgabefunktionen, die Zeichen oder Zeichenketten (bis \n) einlesen und ausgeben, auf Dateien mit Satz-E/A nicht anwendbar.

Folgende Funktionen dienen zur Verarbeitung von Dateien mit Satz-E/A:

`fopen`/`fopen64`, `freopen`/`freopen64`, `fclose`	Öffnen, Schließen
`fread`, `fwrite`	Lesen, Schreiben
`fsetpos`/`fsetpo64`, `fgetpos`/`fgetpos64`, `flocate`, `fseek`/`fseek64`, `rewind`	Positionieren
`fdelrec`	Löschen Satz

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

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

Im Gegensatz zur Strom-E/A werden bei Satz-E/A keine Daten im Puffer zwischengespeichert (siehe „Pufferung“).

Stromorientierte Ein-/Ausgabe

Stromorientierte Ein-/Ausgabe bedeutet, dass sich der Lese-/Schreibzeiger auf jedes einzelne Byte in der Datei positionieren lässt. Strom-E/A ist der herkömmliche Verarbeitungsmodus und standardmäßig eingestellt, d.h. ohne besondere Zusatzangaben bei den Eröffnungsfunktionen. Textdateien können ausschließlich in diesem E/A-Modus verarbeitet werden.
Im Gegensatz zur Satz-E/A werden bei der Ausgabe in Dateien mit Strom-E/A die Daten zunächst in einem C-internen Puffer zwischengespeichert und erst später in die externe Datei geschrieben (siehe „Pufferung“).

Textdatei

Textdateien gibt es nur für Strom-E/A.

Folgende Dateiarten werden als Textdateien behandelt:

– katalogisierte SAM-Dateien (kein Binärmodus beim Öffnen),
– katalogisierte ISAM-Dateien,
– Systemdateien (SYSDTA, SYSOUT, SYSLST, SYSTERM).

Eine Textdatei ist eine geordnete Folge von Bytes, die zu Zeilen (bzw. Sätzen) zusammengefasst sind. Im Unterschied zu Binärdateien werden die Steuerzeichen für Zwischenraum je nach Art der Textdatei in ihre entsprechende Wirkung umgesetzt (siehe „Zwischenraum“). Daten, die aus einer Textdatei eingelesen werden, entsprechen daher nicht genau den Daten, die ursprünglich in die Datei geschrieben wurden. Für einen geschriebenen Tabulator (\t) wird eine entsprechende Anzahl von Leerzeichen gelesen.

Zusätzlich gibt es bei Textdateien noch Folgendes zu beachten:

– Es können Neue-Zeile-Zeichen eingelesen werden, die ursprünglich nicht in die Datei geschrieben wurden (siehe `fflush`, `fseek`/`fseek64`, `fsetpos`/`fsetpos64`, `lseek`/`lseek64`, `rewind`).

– Ausgabe auf SYSOUT und SYSTERM (zum Schreiben)
Jede Zeile wird mit einem Leerzeichen als Drucksteuerzeichen begonnen. Dies bewirkt einen Zeilenvorschub.

– Ausgabe auf SYSLST
Nur wenn keines der Steuerzeichen \f, \v, \r oder \b in einer Zeile angegeben wird, beginnt die Zeile mit einem Leerzeichen als Drucksteuerzeichen.

– Der Inhalt einer Textdatei wird stets als eine Folge von EBCDIC-Zeichen interpretiert. Bei der Verarbeitung von Textdateien mit der ASCII-Variante einer E/A-Funktion (siehe Seite 34) wird deshalb intern wie folgt konvertiert:

  – Beim Schreiben in die Datei: Konverierung von ASCII nach EBCDIC
  – Beim Lesen aus der Datei: Konvertierung von EBCDIC nach ASCII

Zwischenraum

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

In der folgenden Tabelle bedeutet

X Steuerzeichen wird in die entsprechende Wirkung umgesetzt

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

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

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

  In SAM- und ISAM-Dateien wird das Tabulatorzeichen bei KR-Funktionalität standardmäßig in Leerzeichen umgesetzt (KR-Funktionalität ist nur in C/C++ Versionen kleiner V3.0 vorhanden).
  bei ANSI-Funktionalität dagegen nicht (siehe Zusatzangabe "tabexp" bei den Funktionen `fopen`/`fopen64`, `freopen`/`freopen64`).

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

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

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

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

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

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

4.2 Unterstützung von DVS- und UFS-Dateien > 2 GB

Für die Bearbeitung von Dateisystemen, die Dateien > 2 Gigabyte (GB) enthalten, gibt es jeweils eine 64-Bit-Variante zu den nachfolgend aufgelisteten 32-Bit-C-Bibliotheksfunktionen. Die 64-Bit-Funktionen unterscheiden sich von den korrespondierenden 32-Bit-Funktionen durch das Suffix „64“ im Namen.

creat:	creat64
fgetpos:	fgetpos64
fopen:	fopen64
freopen:	freopen64
fseek:	fseek64
fseeko:	fseeko64
fsetpos:	fsetpos64
ftell:	ftell64
ftello:	ftello64
lseek:	lseek64
open:	open64
tmpfile:	tmpfile64

32-Bit- und 64-Bit-C/C++-Bibliotheksfunktionen

Zwischen der 32-Bit-Variante einer Funktion und der zugehörigen 64-Bit-Variante besteht kein funktionaler Unterschied. Abweichungen gibt es nur hinsichtlich der Datentypen für Parameter und Rückgabewerte, falls diese einen Offset oder eine Dateiposition spezifizieren, da für die Bearbeitung von Dateien > 2 GB auch Offset- und Rückgabewerte > 2 GB möglich sein müssen. So gibt es z.B. neben dem 32-Bit-Datentyp `off_t` den 64-Bit-Datentyp `off64_t`.

Die Übersetzungsumgebung stellt alle expliziten 64-Bit-Funktionen und -Typen zusätzlich zu den 32-Bit-Funktionen und Typen zur Verfügung. Damit kann ein Programm, je nach Bedarf, beide Schnittstellen verwenden.

- Die 64-Bit-Funktionen stehen nur in ANSI-Funtionalität zur Verfügung.
- Da die meisten Namen der 64-Bit-Funktionen, auf 8 Zeichen gekürzt, CRTE-weit nicht mehr eindeutig sind, müssen Sourcen, die 64-Bit-Funktionen nutzen wollen, als LLMs generiert werden.

64-Bitschnittstelle nutzen

Für die Nutzung der 64-Bit-Schnittstelle stehen Ihnen zwei Alternativen zur Verfügung, zwischen denen Sie mithilfe des Defines `_FILE_OFFSET_BITS` wählen können:

– 64-Bit-Funktionen transparent nutzen (`_FILE_OFFSET_BITS 64`)
– 64-Bit-Funktionen explizit aufrufen (`_FILE_OFFSET_BITS 32`)

- Das Define `_FILE_OFFSET_BITS` muss vor dem ersten Include auf eine Include-Datei gesetzt werden.
- Die automatische Ersetzung durch die 64-Bit-Funktionen können Sie wahlweise via Namens-Defines oder Makro-Defines durchführen lassen (siehe Abschnitt „Präprozessor-Define _MAP_NAME“ auf Seite 43).

64-Bit-Funktionen transparent nutzen (`_FILE_OFFSET_BITS 64`)

Das Define `_FILE_OFFSET_BITS 64` ermöglicht die transparente Nutzung der 64-Bit-Schnittstelle, da die im Source-Code notierten 32-Bit-Funktionen bei der Übersetzung automatisch durch die zugehörigen 64-Bit-Varianten ersetzt werden (Ausnahme `fseek` und `ftell`, siehe unten). Außerdem stellt die Übersetzungsumgebung Datentypen in der passenden Größe bereit. So ist beispielsweise der Datentyp `off_t` als `long long` deklariert.

Mit dem Präprozessor-Define `_MAP_NAME` wählen Sie, ob die Abbildung auf die 64-Bit-Funktionen via Namens-Define-Technik oder via Makro-Define-Technik erfolgen soll (siehe Seite 43).

Ein Programm kann sowohl Dateien > 2 GB als auch Dateien ≤ 2 GB bearbeiten. Die transparente Nutzung der 64-Bit-Funktionen ermöglicht es, dass Programme, die bisher ausschließlich für Dateien ≤ 2 GB vorgesehen waren, ohne Anpassungen im Source-Code auch Dateien > 2 GB bearbeiten können.

Bei den Funktionen `fseek` und `ftell` ist die automatische Ersetzung durch `fseek64` bzw. `ftell64` nicht möglich. Verwenden Sie deshalb bitte die Funktionen `fseeko` bzw. `ftello`, wenn Sie die automatische Ersetzung wünschen.

64-Bit-Funktionen explizit aufrufen

Wenn das Define `_FILE_OFFSET_BITS 32` gesetzt ist oder wenn `_FILE_OFFSET_BITS` nicht definiert ist, müssen Sie für die Bearbeitung von Dateien > 2 GB die 64-Bit-Varianten der oben genannten Dateibearbeitungsfunktionen verwenden:

– Falls Sie versuchen, eine Datei > 2 GB mit einer 32-Bit-Variante zu bearbeiten, führt dies zum Abbruch.

– Mit den 64-Bit-Varianten können Sie jedoch auch Dateien ≤ 2 GB bearbeiten.

Die 64-Bit-Funktionen können explizit nur dann genutzt werden, wenn vorher das Define `_LARGEFILE64_SOURCE 1` gesetzt wird (Prototyp-Generierung und weitere Defines).

4.3 Systemdateien (SYSDTA, SYSOUT, SYSLST)

SYSDTA

Ein C-Programm kann SYSDTA folgendermaßen verwenden:

– Mit einer Eröffnungsfunktion (`fopen`/`fopen64`, `freopen`/`freopen64`, `open`/`open64`) wird eine Datei mit dem Namen "(SYSDTA)" oder "(SYSTERM)" zum Lesen geöffnet. Der von der Eröffnungsfunktion gelieferte Dateizeiger dient dann als Argument einer anschließenden Eingabefunktion.

  Beispiel

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

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

  Beispiele

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

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

Soll die Eingabe nicht von der Datensichtstation, sondern aus einer katalogisierten Datei erfolgen, kann dies auf zweierlei Weise geschehen:

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

   Diese Umweisung wirkt sich **nicht** auf Dateien aus, die mit dem Namen "(SYSDTA)" bzw. "(SYSTERM)" eröffnet werden. Die Eingabe aus Dateien dieses Namens wird nach wie vor von der Datensichtstation erwartet.

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

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

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

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

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

Hinweis

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

SYSOUT

Ein C-Programm kann SYSOUT folgendermaßen verwenden:

– Mit einer Eröffnungsfunktion (`fopen`/`fopen64`, `freopen`/`freopen64`, `open`/`open64`) wird eine Datei mit dem Namen "(SYSOUT)" oder "(SYSTERM)" zum Schreiben geöffnet. Der von der Eröffnungsfunktion gelieferte Dateizeiger dient dann als Argument einer anschließenden Ausgabefunktion.

Beispiel

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

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

Beispiele

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

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

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

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

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

SYSLST

Ein C-Programm kann SYSLST folgendermaßen verwenden:

– Mit einer Eröffnungsfunktion (`fopen`/`fopen64`, `freopen`/`freopen64`, `open`/`open64`) wird eine Datei mit dem Namen "(SYSLST)" zum Schreiben geöffnet. Der von der Eröffnungsfunktion gelieferte Dateizeiger dient als Argument einer anschließenden Ausgabefunktion.

  Beispiel

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

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

  Diese Umweisung wirkt sich **nicht** auf Dateien aus, die mit dem Namen "(SYSOUT)" eröffnet wurden.

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

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

4.4 Katalogisierte Plattendateien (SAM, ISAM, PAM)

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

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

Beim Neuanlegen einer Datei gelten je nach C-Dateiart (Binärdatei, Textdatei, stromorientierte oder satzorientierte Ein-/Ausgabe) Standardwerte des C-Laufzeitsystems. Diese Werte können mit einem ADD-FILE-LINK-Kommando vor Aufruf des Programms geändert werden. Dazu muss bei den Öffnungsfunktionen (`open`/`open64`, `creat`/`creat64`, `fopen`/`fopen64`, `freopen`/`freopen64`) ein Linkname angegeben ("link=*linkname*") und dieser Linkname im ADD-FILE-LINK-Kommando mit dem Namen der katalogisierten Datei verknüpft werden.

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

Im Folgenden erhalten Sie Informationen

– zu den Standardwerten sowie den möglichen Modifikationen der Dateiattribute,

– zum K- und NK-Blockformat,

– zur strom- und satzorientierten Verarbeitung von Plattendateien,

– zum Last Byte Pointer (LBP).

4.4.1 Standardwerte und zulässige Modifikationen der Dateiattribute

Mit den Ein-/Ausgabe-Funktionen des C-Laufzeitsystems können Plattendateien mit den in den folgenden Tabellen 1 bis 3 aufgeführten Dateiattributen verarbeitet werden. Die Standardattribute, die das Laufzeitsystem einsetzt, wenn der Benutzer keine Angaben im ADD-FILE-LINK-Kommando bzw. bei den Eröffnungsfunktionen macht, sind jeweils unterstrichen.

Erläuterungen zu den Tabellen 1 bis 3

– Die maximale Anzahl Datenbyte in den Tabellen gibt die Anzahl der Zeichen an, die vom C-Programm aus in einen Satz bzw. Block abgelegt werden (feste Satzlänge) oder maximal abgelegt werden können (variable Satzlänge).

– Die Größe des logischen Blocks (BLKSIZE) ist abhängig von Art und Format des Datenträgers (siehe auch Seite 81).
K- und NK2-Platten: Standardblock (2048 Bytes) oder ein ganzzahliges Vielfaches eines Standardblocks (maximal 16 Standarblöcke).
NK4-Platten: Mindestens zwei Standardblöcke (4096 Bytes) oder ein ganzzahliges Vielfaches davon (2, 4, 6, 8 Standardblöcke).

– Zum Blockformat (BLKCTRL) und zur maximalen Anzahl Datenbyte beachten Sie bitte auch den Abschnitt „K- und NK-Blockformat“ auf Seite 81.
Insbesondere finden Sie dort Hinweise, wie bei NK-ISAM-Dateien Überlaufblöcke vermieden werden können, die dann entstehen, wenn beim Schreiben der Sätze die volle Länge einer Übertragungseinheit (RECSIZE = BLKSIZE) ausgenutzt wird.

– Bei Dateien mit variabler Satzlänge (RECFORM=V) zählt in C generell das 4 Byte lange Satzlängenfeld nicht zu den Satzdaten. Die maximale Anzahl Datenbyte reduziert sich deshalb um 4 Bytes.

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

Tabelle 1: Dateiattribute von Textdateien bei Stromorientierter Ein-/Ausgabe

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

1) Bei KR-Funktionalität werden standardmäßig SAM-Dateien erstellt (KR-Funktionalität ist nur in C/C++ Versionen kleiner V3.0 vorhanden).
Bei ANSI-Funktionalität werden standardmäßig ISAM-Dateien erstellt.

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

Tabelle 2: Dateiattribute von Binärdateien bei Stromorientierter Ein-/Ausgabe

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

Tabelle 3: Dateiattribute von Binärdateien bei Satzorientierter Ein-/Ausgabe

FCB-TYPE	REC-FORM	BLKCTRL	BLKSIZE (STD,n)	RECSIZE (r Byte)	Max. Anzahl Datenbyte
SAM	V	PAMKEY	1≤ n ≤16	4≤ r ≤n*2048-4	RECSIZE - 4
		DATA(2K)	1≤ n ≤16	4≤ r ≤n*2048-16	RECSIZE - 4
		DATA(4K)	2≤ n ≤16		
	F	PAMKEY	1≤ n ≤16	1≤ r ≤n*2048	RECSIZE
		DATA(2K)	1≤ n ≤16	1≤ r ≤n*2048-16	RECSIZE
		DATA(4K)	2≤ n ≤16		
	U	PAMKEY	1≤ n ≤16		BLKSIZE
		DATA(2K)	1≤ n ≤16		BLKSIZE - 16
		DATA(4K)	2≤ n ≤16		
PAM		PAMKEY	1≤ n ≤16		BLKSIZE
		DATA(2K)	1≤ n ≤16		BLKSIZE - 12
		DATA(4K)	2≤ n ≤16		
		NO(2K)	1≤ n ≤16		BLKSIZE
		NO(4K)	2≤ n ≤16		
ISAM [1)]	V	PAMKEY	1≤ n ≤16	5≤ r ≤n*2048	RECSIZE - 4
		DATA(2K)	1≤ n ≤16	5≤ r ≤n*2048	RECSIZE - 4
		DATA(4K)	2≤ n ≤16		
	F	PAMKEY	1≤ n ≤16	1≤ r ≤n*2048-4	RECSIZE
		DATA(2K)	1≤ n ≤16	1≤ r ≤n*2048-4	RECSIZE
		DATA(4K)	2≤ n ≤16		

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

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

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

4.4.2 K- und NK-Blockformat

Das BS2000 unterstützt Datenträger, die unterschiedlich formatiert sind:

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

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

Zusätzlich werden NK-Datenträger nach der Mindestgröße der Übertragungseinheit (Transfer Unit) unterschieden. NK2-Datenträger haben eine Transfer Unit von 2KByte. NK4-Datenträger haben eine Transfer Unit von 4KByte.

Das Blockformat wird durch den Operanden BLOCK-CONTROL-INFO des ADD-FILE-LINK-Kommandos gesteuert. Die ausführliche Beschreibung des BLOCK-CONTROL-INFO-Operanden, der verschiedenen Datei- und Datenträgerstrukturen sowie der Umstellung von K-Dateiformat auf NK-Dateiformat findet sich im Handbuch „DVS Einführung“.

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

Datei-organisation	CLASS2-OPTION BLKCTRL = NONKEY			
	nicht angegeben		angegeben	
	K-Platte	NK-Platte	K-Platte	NK-Platte
SAM	PAMKEY	DATA	DATA	DATA
ISAM	PAMKEY	DATA	DATA	DATA
PAM	PAMKEY	NO	NO	NO

K- und NK-ISAM-Dateien

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

Die Bildung von Überlaufblöcken bringt folgende Probleme mit sich:

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

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

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

Für Datensätze nutzbarer Bereich bei NK-ISAM-Dateien

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

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

Zur Erläuterung der Formeln:

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

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

Dateivereinbarung:

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

maximale Satzlänge nach der Formel:

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

4.4.3 Unterstützung der Zugriffsmethode DIV

Die Zugriffsart DIV (DATA IN VIRTUAL) eignet sich insbesondere für die Bearbeitung von unstrukturierten Datenströmen, wie sie in C-Programmen (u.a. aus UNIX portierten) häufig vorkommen.

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

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

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

Das C-Laufzeitsystem führt die stromorientierte Ein-/Ausgabe auf NK-PAM-Dateien ohne Datenverwaltungsinformationen generell mit der Zugriffsart DIV durch. Bei NK-PAM-Dateien, die für satzorientierte Ein-/Ausgabe geöffnet werden, ist die Verwendung von DIV nicht möglich.

Hinweise zur stromorientierten Ein-/Ausgabe

Binärdateien (SAM)

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

Wird mit variabler Satzlänge gearbeitet (V oder U), kann das Weiterschreiben bytebezogen erfolgen. Allerdings sind durch die variablen Satzlängen beim Positionieren (z.B. mit `fseek`/`fseek64`, `ftell`/`ftell64`) Performanceverluste in Kauf zu nehmen.

Binärdateien (PAM)

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

Textdateien (SAM, ISAM)

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

Hinweise zur satzorientierten Ein-/Ausgabe

Satzorientierte Ein-/Ausgabe, die bei SAM-, ISAM- und PAM-Dateien möglich ist, ist immer eine binäre Ein-/Ausgabe. Daher werden bei satzorientierter Ein-/Ausgabe mit den ASCII-Varianten der Ein-/Ausgabefunktionen (siehe Seite 34) Daten weder beim Schreiben, noch beim Lesen konvertiert.

Mit den Funktionen `fopen`/`fopen64` bzw. `freopen`/`freopen64` muss die Datei stets im Binärmodus und mit dem Zusatz "type=record" eröffnet werden.

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

Verfügbare Ein-/Ausgabefunktionen

Folgende Funktionen stehen zur Verarbeitung von Dateien mit Strom-E/A zur Verfügung:

`fopen`/`fopen64`, `freopen`/`freopen64`	Öffnen
`fclose`	Schließen
`fread`	Lesen
`fwrite`	Schreiben
`fsetpos`/`fsetpos64`	Positionieren im Datenstrom
`fgetpos`/`fgetpos64`	Position im Datenstrom
`fseek`/`fseek64`	Positionieren auf Dateianfang/Dateiende
`rewind`	Positionieren auf Dateianfang
`flocate`	explizit Positionieren in einer ISAM-Datei
`fdelrec`	Löschen eines Satzes in einer ISAM-Datei

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

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

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

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

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

FCBTYPE einer neu zu erstellenden Datei

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

– Angabe in einem ADD-FILE-LINK-Kommando und Verwendung des LINK-Namens bei den Funktionen `fopen`/`fopen64` bzw. `freopen`/`freopen64`

– Angabe des `forg`-Parameters bei den Funktionen `fopen`/`fopen64` bzw. `freopen`/`freopen64`, und zwar:

  „forg=seq“: Es wird eine SAM-Datei erstellt
  „forg=key“: Es wird eine ISAM-Datei erstellt.

Es gibt folgende Einschränkungen für den FCBTYPE einer Datei und die Angaben bei `fopen`/`fopen64` bzw. `freopen`/`freopen64`:

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

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

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

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

Mehrfachschlüssel bei ISAM-Dateien

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

Beispiel für die satzorientierte Verarbeitung einer ISAM-Datei

Folgendes Programm erstellt und verarbeitet eine ISAM-Datei mit satzorientierter Ein-/Ausgabe.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

main()
{
  FILE * isamfp;
  size_t ret=0;
  int    i,intret;
  char   puffer[200];
  char   puffer2[200];
  static char * texts[3] = {"1  Ritchie***, 9999, ZZ",
                            "2  Kernighan*, 8765",
                            "3  Stroustrup, 1234, C++"};

  static char isamlink[] = "ADD-FILE-LINK LINK=ISAMFILE,F-NAME=DATEI.ISAM,"
                           "ACCESS-METHOD=ISAM(KEY-LEN=10,KEY-POS=4),"
                           "REC-FORM=FIXED(REC-SIZE=50)";

  static int maxtext = 3;
  fpos_t isampos;

  ret = system(isamlink);
  if (ret != 0)
  {
     printf("system(isamlink) fehlerhaft\n");
     exit(1);
  }

  isamfp = fopen("link=isamfile", "wb+,type=record,forg=key");
  if (isamfp == 0)
  {
     printf("Das Oeffnen von isamfp ist nicht gelungen\n");
     exit(2);
  }

  /* 3 Saetze in ISAM-Datei schreiben */

  for (i=0; i<maxtext; i++)
  {
     ret = fwrite(texts[i], 1, strlen(texts[i]), isamfp);
     if (ret == 0)
     {
```

```
            printf("Fehler beim ISAM-Schreiben\n");
            exit(3);
         }
       }

      /* Saetze ab Dateibeginn lesen und auf Standardausgabe schreiben */

      rewind(isamfp);
      for (i=0; i<maxtext; i++)
      {
         ret = fread(puffer, 1, 100, isamfp);
         fwrite(puffer, 1, ret, stdout);
      }

      /* Explizit auf Grund des Schluesselwerts positionieren und verarbeiten
    */

      flocate(isamfp, "Ritch", strlen("Ritch"), _KEY_GE);

      ret = fread(puffer, 1, 100, isamfp);     /*"Ritchie" ist gelesen         */
      *(puffer+ret) = '\0';                    /* EOS am Ende des Satzes       */
      printf("\nGelesener Satz: %s\n", puffer);
      fgetpos(isamfp, &isampos);               /* Position hinter gelesenem    */
                                               /* Satz merken                  */

      ret = fread(puffer, 1, 100, isamfp);     /*"Stroustrup" ist gelesen      */
      *(puffer+ret) = '\0';                    /* EOS am Ende des Satzes       */
      printf("Gelesener Satz: %s\n", puffer);

      fsetpos(isamfp, &isampos);               /* Zurueckpositionieren         */
      ret = fread(puffer2, 1, 100, isamfp);    /*"Stroustrup"ist erneut gelesen*/
      *(puffer2+ret) = '\0';                   /* EOS am Ende des Satzes       */
      printf("Gelesener Satz: %s\n", puffer2);

      intret = fdelrec(isamfp, "Kernighan*");  /* einen Satz loeschen          */
      if (intret == 0)
         printf("Kernighan geloescht\n");

      intret = fdelrec(isamfp, "Kernighan*");  /* Versuch, bereits geloeschten */
      if (intret > 0)                          /* Satz zu loeschen             */
         printf("OK, dieser Satz existiert nicht mehr\n");
      else
         printf("Fehler, \"Kernighan*\" koennte man zweimal loeschen\n");

      printf("******* PROGRAMM ENDE *******\n");
    }
```

4.4.4 Last Byte Pointer (LBP)

Im BS2000 ist die Länge einer PAM-Datei unabhängig von ihrem Inhalt immer ein ganzzahliges Vielfaches eines PAM-Blocks. Ab BS2000 OSD/BC V10.0 enthält der Katalogeintrag für PAM-Dateien den Eintrag Last Byte Pointer (LBP), in dem die echte Länge der Datei in Bytes hinterlegt werden kann. Dadurch können insbesondere auch Dateien, die auf einem Netzwerkserver (NAS) abgelegt sind, von allen darauf zugreifenden Systemen (auch UNIX) bytegenau gelesen und geschrieben werden.

Bisher wurde die Länge einer PAM-Datei mit einer Hilfskonstruktion ermittelt, indem das eigentliche Ende der Datei durch einen speziellen Marker gekennzeichnet wurde. Auf diese Hilfskonstruktion kann ab BS2000 OSD/BC V10.0 verzichtet werden.

Von dieser Schnittstelle sind alle C-Laufzeitfunktionen betroffen, die PAM-Dateien öffnen.

Die Funktionen `fopen`, `fopen64`, `freopen`, `freopen64`, `open`, `open64` und `creat`, `creat64` werden daher um den *lbp*-Schalter erweitert. Näheres finden Sie in den Beschreibungen der entsprechenden Funktionen.

Beim Öffnen oder Lesen bestehender Dateien verhalten sich diese Funktionen unabhängig vom *lbp*-Schalter folgendermaßen:

– Ist der LBP der Datei ungleich 0, wird er ausgewertet. Ein eventuell vorhandener Marker wird ignoriert.

– Ist der LBP = 0, wird nach einem Marker gesucht und die Dateilänge daraus ermittelt. Falls kein Marker gefunden wird, wird das Ende des letzten vollständigen Blocks als Dateiende betrachtet.

Beim Schließen von Dateien, die verändert oder neu erstellt wurden, hängt das Verhalten vom *lbp*-Schalter beim Öffnen der Datei bzw. von der Umgebungsvariablen LAST_BYTE_POINTER ab.

Umgebungsvariable LAST_BYTE_POINTER

Die Umgebungsvariable LAST_BYTE_POINTER hat den Zweck, dass bestehende Programme den LBP nutzen können, ohne dass in sie eingegriffen werden muss. Festgebundene Programme müssen dann lediglich mit dem aktuellen CRTE neu gebunden werden. Für Programme, die mit PARTIAL-BIND oder CRTE-BASYS gebunden sind, genügt es, wenn das aktuelle CRTE bzw. CRTE-BASYS installiert ist.

Falls eine der betroffenen Funktionen ohne *lbp*-Schalter aufgerufen wird, hängt ihr Verhalten vom Inhalt der Umgebungsvariablen LAST_BYTE_POINTER ab:

`LAST_BYTE_POINTER=YES`

Die Funktionen `fopen`, `fopen64` und `freopen`, `freopen64` verhalten sich so, als ob im Parameter `art lbp=yes` angegeben wäre.

Die Funktionen `open`, `open64` und `creat`, `creat64` verhalten sich so, als ob im Parameter `modus O_LBP` angegeben wäre.

Beim Schließen einer Datei, die verändert oder neu erstellt wurde, wird kein Marker geschrieben (auch wenn einer vorhanden war) und ein gültiger LBP gesetzt. Auf diese Weise können Dateien mit Marker auf LBP ohne Marker umgestellt werden.

`LAST_BYTE_POINTER=NO`

Die Funktionen `fopen`, `fopen64` und `freopen`, `freopen64` verhalten sich so, als ob im Parameter `art lbp=no` angegeben wäre.

Die Funktionen `open`, `open64` und `creat`, `creat64` verhalten sich so, als ob im Parameter `modus O_NOLBP` angegeben wäre.

Beim Schließen einer Datei, die verändert oder neu erstellt wurde, wird der LBP auf Null gesetzt. Für eine neu erstellte Datei wird immer ein Marker geschrieben, für eine veränderte nur dann, wenn vorher bereits ein Marker vorhanden war. War kein Marker vorhanden, wird auch keiner geschrieben und die Datei endet mit dem vollständigen letzten Block.

Ist die Umgebungsvariable nicht gesetzt, verhalten sich die Funktionen so, als ob sie den Wert `NO` hätte.

Näheres zur Verwendung von Umgebungsvariablen im BS2000 finden Sie im Abschnitt „Umgebungsvariablen“ auf Seite 118).

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

Wird mit den Funktionen `fopen`/`fopen64`, `freopen`/`freopen64` oder `open`/`open64` der Dateiname "(INCORE)" angegeben, wird eine temporäre PAM-Datei im virtuellen Speicher angelegt. Diese Datei „lebt“ nur für die Dauer eines Programmablaufs.

INCORE-Dateien müssen zuerst zum Schreiben eröffnet werden, bevor auf sie lesend zugegriffen werden kann (vgl. `fopen`/`fopen64`, `freopen`/`freopen64`, `open`/`open64`).

INCORE-Dateien werden als Binärdateien verarbeitet.

4.6 Standard-Ein-/Ausgabedateien stdin, stdout, stderr

Im Unterschied zu anderen Implementierungen sind die Makros `stdin`, `stdout` und `stderr` keine konstanten Ausdrücke und können daher nicht in statischen Initialisierungen verwendet werden.

Beispiel

Folgende Konstruktion ist nicht erlaubt:

```
FILE *fp = stdin;
int main(void)
{
  .
  .
}
```

5 Contingency- und STXIT-Routinen

Dieses Kapitel gibt Hinweise, wie in C Contingency- bzw. STXIT-Routinen realisiert werden können.

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

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

Achtung

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

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

- Speicherverwaltungsroutinen: `malloc`, `calloc`, `realloc`, `free`
- Dateizugriffsfunktionen zum Öffnen und Schließen von Dateien: `fopen`/`fopen64`, `freopen`/`freopen64`, `open`/`open64`, `creat`/`creat64`, `fclose`, `close`
- Alle Dateizugriffs-, Dateiverwaltungs- und Ein-/Ausgabe-Funktionen, die auf die gleiche Datei angewendet werden
- Zufallsgeneratorfunktionen: `rand`, `srand`
- Zeitfunktionen: `localtime/localtime_r`, `gmtime/gmtime_r`
- Funktionen zum An- und Abmelden von Contingency-Routinen: `cenaco`, `cdisco`
- `atexit`
- `strtok`
- `setlocale`

Zu der Gruppe der „kritischen“ Funktionen gehören außerdem die Ein-/Ausgabefunktionen aus der C++-Standardbibliothek.

5.1 C-Bibliotheksfunktionen (alarm, raise, signal)

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

`alarm`	Signal SIGALRM senden (STXIT-Ereignis RTIMER)
`raise`	Signale senden (simulierte STXIT-Ereignisse und benutzerdefinierte Ereignisse)
`signal`	Signalbearbeitungs-Routinen zuordnen

STXIT-Contingency-Routinen

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

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

Die Ereignisklasse SVC-Unterbrechung wird derzeit nicht unterstützt.

Ereignisgesteuerte Routinen

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

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

5.2 Freie Verwendung von Contingency-Routinen

Bei speziellen Anforderungen, die durch die Funktionen `signal` und `raise` nicht abgedeckt sind (siehe Abschnitt „C-Bibliotheksfunktionen (alarm, raise, signal)“ auf Seite 94), können die entsprechenden BS2000-Funktionen für Ereignissteuerung frei programmiert werden. Solche Anforderungen sind z.B. eine größere Anzahl von Ereignissen (mit `raise` und `signal` lassen sich nur zwei Ereignisse selbst definieren) oder Inter-Task-Kommunikation (mit `raise` und `signal` ist Ereignissteuerung nur innerhalb eines Tasks möglich).

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

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

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

Contingency-Routine in C

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

```
struct contp
{
   int     comess;         /* contingency message */
   evcode  indicat;        /* information indicator */
   char    filler[2];      /* reserved for int. use */
   evcode  switchc;        /* event switch */
   int     pcode;          /* post code */
   int     __reg4;         /* Register 4 */
   int     __reg5;         /* Register 5 */
   int     __reg6;         /* Register 6 */
   int     __reg7;         /* Register 7 */
   int     __reg8;         /* Register 8 */
};
```

```
#define evcode     char
#define _normal    0       /* evceventnormal */
#define _abnorm1   4       /* evceventabnormal */
#define _nmnpc     0       /* evcnocomessnopostcode */
#define _mnpc      4       /* evccomessnopostcode */
#define _nmpc      8       /* evcnocomesspostcode */
#define _mpc       12      /* evccomesspostcode */
#define _etnm      0       /* evcelapsedtimenocomess */
#define _etm       4       /* evcelapsedtimecomess */
#define _disnm     16      /* evceventdisablednocomess */
#define _dism      20      /* evceventdisabledcomess */
```

Aufbau der Contingency-Routine:

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

```
#include <cont.h>

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

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

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

Contingency-Routine in Assembler

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

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

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

In der Contingency-Routine darf der RETCO-Makro nicht aufgerufen werden!

Die Rückkehr muss mit dem Makro @EXIT erfolgen.

5.3 Freie Verwendung von STXIT-Contingency-Routinen

Bei speziellen Anforderungen, die durch die Funktion `signal` nicht abgedeckt sind (siehe Abschnitt „C-Bibliotheksfunktionen (alarm, raise, signal)“ auf Seite 94), können STXIT-Contingency-Routinen in C frei programmiert werden. Solche Anforderungen sind z.B. umfangreichere Informationsübergaben oder mehr Fortsetzungs-Steuermöglickeiten nach Ablauf der STXIT-Contingency-Routine.

Die Definition einer frei programmierten STXIT-Contingency-Routine muss durch Aufruf der C-Bibliotheksfunktion `cstxit` bzw. `_cstxit` erfolgen.

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

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

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

Aufbau der STXIT-Contingency-Routine:

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

```
#include <stxit.h>

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

Diese Routine kann auf drei verschiedene Arten beendet werden:

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

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

6 Lokalität

6.1 Konzept der Lokalität

Hinter dem Begriff „Lokalität“ steht das Konzept, das Verhalten von C-Programmen an landesübliche Konventionen, Normen und Sprache anzupassen.

Die Lokalität betrifft zum einen direkt den Ablauf einiger C-Bibliotheksfunktionen. Außerdem werden mit der Funktion `localeconv` lokalitätsspezifische Informationen in einer Struktur zur Verfügung gestellt, die bei der formatierten Ausgabe (`printf`, `fprintf` etc.) verwendet werden können.

Die Lokalität umfasst folgende Kategorien:

LC_COLLATE	Die Sortierreihenfolge des Zeichenvorrats beeinflusst das Verhalten der Funktionen `strcoll`, `strxfrm`, `wcscoll` und `wcsxfrm`.
LC_CTYPE	Die Klassifizierung der Zeichen beeinflusst das Verhalten der Makros zur Zeichenbearbeitung `is`... (nicht `isdigit`, `iswdigit`, `isxdigit` und `iswxdigit`) sowie die Funktionen `tolower`, `toupper`, `towctrans`, `towlower`, `towupper`, `strlower`, `strupper` und `wctrans`.
LC_MONETARY	Die Konventionen zur Darstellung von monetären Werten (z.B. Währungsdarstellung) beinflussen die von `localeconv` gelieferten Werte.
LC_NUMERIC	Die Konventionen zur Darstellung von nicht-monetären numerischen Werten (z.B. Dezimalpunkt, Vorzeichen) beeinflussen die Art des Dezimalpunktes bei formatierter Ein-/Ausgabe und bei der Umwandlung von Zeichenketten (`atof`, `strtod`, `wcstod`), sowie die von `localeconv` gelieferten Werte.
LC_TIME	Die Konventionen zur Darstellung von Datum und Uhrzeit beeinflussen das Verhalten von `strftime` und `wcsftime`.

Das C-Laufzeitsystem stellt einige vordefinierte Lokalitäten zur Verfügung (siehe Abschnitt „Vordefinierte Lokalität C“ auf Seite 100). Der Benutzer kann aber auch eine eigene Lokalität definieren (siehe Abschnitt „Kompatible Lokalitäten V1CTYPE und V2CTYPE“ auf Seite 103).

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

Die gewünschte Lokalität, unter der das C-Programm ablaufen soll, wird mit der Funktion `setlocale` ausgewählt.
Die ausführliche Beschreibung der in diesem Abschnitt erwähnten C-Bibliotheksfunktionen finden Sie im Nachschlageteil dieses Handbuchs.
Voreingestellt ist die Lokalität C, sofern nicht die `main`-Routine ein C-V1.0-Objekt ist; in diesem Falle wird bei Programmstart automatisch die Lokalität "V1CTYPE" bzw. LC_C_V1CTYPE eingestellt.

6.2 Vordefinierte Lokalität C

Das C-Laufzeitsystem stellt mehrere vordefinierte Lokalitäten zur Verfügung. Bei Programmstart ist die "C"-Lokalität eingestellt.

Standard-Lokalität

Diese Lokalität wird mit "" oder LC_C_DEFAULT bezeichnet. Sie entspricht in dieser Version stets der C-Lokalität.

C-Lokalität

Diese Lokalität wird mit "C" oder LC_C_C bezeichnet. Sie ist bei Programmstart voreingestellt (Ausnahme: die `main`-Routine ist ein C-V1.0-Objekt, dann gilt "V1CTYPE", siehe Seite 103).

Die C-Lokalität hat für die einzelnen Kategorien folgende Auswirkungen:

LC_COLLATE
: Die Sortierreihenfolge der Zeichen richtet sich nach der Definition im XPG4-Standard, in der die Folge dem ASCII-Wert der Zeichen entspricht (7-bit-Code, siehe Tabelle auf Seite 102). Für alle anderen Kategorien entspricht die Sortierreihenfolge dem EBCDIC-Wert der Zeichen, wie er in der Tabelle auf Seite 105 dargestellt ist).

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

LC_NUMERIC

Die in `localeconv` definierten Informationen haben folgende Werte:

decimal_point	'.'
thousands_sep	""
grouping	""

LC_MONETARY

Die in `localeconv` definierten Informationen haben folgende Werte:

int_curr_symbol	""
currency_symbol	""
mon_decimal_point	""
mon_thousands_sep	""
mon_grouping	""
positive_sign	""
negative_sign	""
int_frac_digits	CHAR_MAX (= 255)
frac_digits	CHAR_MAX
p_cs_precedes	CHAR_MAX
n_cs_precedes	CHAR_MAX
p_sep_by_space	CHAR_MAX
n_sep_by_space	CHAR_MAX
p_sign_pos	CHAR_MAX
n_sign_pos	CHAR_MAX

LC_TIME

Für Wochentags- und Monatsnamen wird die englische Sprache verwendet. Die Darstellung von Datum und Uhrzeit entspricht den im englischen Sprachraum üblichen Konventionen.

LC_COLLATE

Die Kategorie LC_COLLATE beeinflusst die Sortierreihenfolge bei den Funktionen `strcoll`, `strxfrm`, `wcscoll` und `wcsxfrm`. In der voreingestellten Lokalität "C" entspricht die Sortierreihenfolge der in der folgenden Tabelle abgebildeten Definiton gemäß dem XPG4-Standard.

Sortierreihenfolge gemäß dem XPG4-Standard (ASCII)

\0	/	D	Y	n
\t	0	E	Z	o
\n	1	F	[	p
\v	2	G	\	q
\f	3	H	]	r
\r	4	I	^	s
␣	5	J	_	t
!	6	K	`	u
"	7	L	a	v
#	8	M	b	w
$	9	N	c	x
%	:	O	d	y
&	;	P	e	z
'	<	Q	f	{
(	=	R	g	|
)	>	S	h	}
*	?	T	i	~
+	@	U	j	
,	A	V	k	
-	B	W	l	
.	C	X	m	

6.3 Kompatible Lokalitäten V1CTYPE und V2CTYPE

V1CTYPE

Diese Lokalität wird mit "V1CTYPE" oder LC_C_V1CTYPE bezeichnet. Sie wird bei Programmstart automatisch eingestellt, wenn die `main`-Routine ein C-V1.0-Objekt ist.

Abweichungen von der C-Lokalität:

LC_CTYPE
: Die Zeichen X'8B', X'8C', X'8D' gelten als Kleinbuchstaben, die Zeichen X'AB', X'AC', X'AD' als Großbuchstaben und die Zeichen X'C0' und X'D0' als Sonderzeichen.
In der Lokalität "C" gelten all diese Zeichen als Kontrollzeichen.

LC-COLLATE
: Die Sortierreihenfolge entspricht dem EBCDIC-Wert der Zeichen (siehe Tabelle auf Seite 105).

V2CTYPE

Diese Lokalität wird mit "V2CTYPE" oder LC_C_V2CTYPE bezeichnet. Sie ist kompatibel zu der Lokalität "C" in den Versionen 2.0 und 2.1 des C-Laufzeitsystems.

Abweichung von der C-Lokalität:

LC-COLLATE
: Die Sortierreihenfolge entspricht dem EBCDIC-Wert der Zeichen (siehe Tabelle auf Seite 105).

6.4 Länderspezifische Lokalität GERMANY

Für den deutschen Sprachraum steht eine länderspezifische Lokalität zur Verfügung. Diese Lokalität wird mit "GERMANY" oder LC_C_GERMANY bezeichnet.

Abweichungen von der C-Lokalität:

LC_CTYPE
Die Zeichen X'FB', X'4F', X'FD' und X'FF' sind als Kleinbuchstaben klassifiziert (ä, ö, ü, ß).
Die Zeichen X'BB', X'BC' und X'BD' sind als Großbuchstaben klassifiziert (Ä, Ö, Ü).

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

LC_MONETARY
Internationales Währungssymbol (int_curr_symbol): "DEM "

Lokales Währungssymbol (currency_symbol): "DM"

Dezimalpunkt (mon_decimal_point): ","

LC_TIME
Für Wochentags- und Monatsnamen wird die deutsche Sprache verwendet.
Die Datumsdarstellung entspricht den im deutschen Sprachraum üblichen Konventionen:

```
<Wochentagsname>, <Tag des Monats>.<Monatsname> <Jahreszahl>
```

z.B. `Donnerstag, 25.Juli 1991`

LC_COLLATE
Die Kategorie LC_COLLATE beeinflusst die Sortierreihenfolge bei den Funktionen `strcoll`, `strxfrm`, `wcscoll` und `wcsxfrm`. In der Lokalität "GERMANY" entspricht die Sortierreihenfolge dem EBCDIC-Wert der Zeichen, wie er aus der folgenden Tabelle ersichtlich ist.

Sortierreihenfolge gemäß dem EBCDIC-Wert

\0	^	j	B	W
\t	,	k	C	X
\v	%	l	D	Y
\f	_	m	E	Z
\r	>	n	F	0
\n	?	o	G	1
␣	:	p	H	2
`	#	q	I	3
.	@	r	J	4
<	'	s	K	5
(	=	t	L	6
+	"	u	M	7
|	a	v	N	8
&	b	w	O	9
!	c	x	P	{
$	d	y	Q	}
*	e	z	R	~
)	f	[	S	
;	g	\	T	
-	h	]	U	
/	i	A	V	

6.5 Die Lokalitäten De.EDF04F und De.EDF04F@euro

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

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

Die beiden Lokalitäten unterscheiden sich nur in der Kategorie LC_MONETARY.

LC_CTYPE

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

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

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

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

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

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

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

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

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

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

Die weiteren Klassen sind definiert wie folgt:

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

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

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

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

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

LC_COLLATE

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

LC_NUMERIC

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

LC_TIME

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

am_pm: "AM", "PM"

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

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

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

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

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

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

```
full_day:"%a %e.%b"
ar_date:"%b %d %H:%M %Y"
last_date:"%a %e.%b %H:%M"
ls_date: "%h %e %H:%M"
ls_date2:"%h %e %Y"
ps_date:"%d.%b"
su_date: "%d.%m %H:%M"
tar_date:"%e.%b %H:%M %Y"
diff_date:"%a %e.%b.%Y, %T"
```

LC_MESSAGES

yesstring	"ja"
nostr	"nein"
quitstr	"abbrechen"
noexpr	"^[nN]"
yesexpr	"^[jJ]"
quitexpr	"^[aA]"

LC_MONETARY

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

6.6 Benutzerspezifische Lokalitäten

Der Benutzer kann sich seine eigenen Lokalitäten definieren.

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

Die Quellprogramme legen die Daten für die einzelnen Lokalitätskategorien fest und sind mit den Daten der C-Lokalität vorbelegt. Der Aufbau dieser Daten ist weiter unten beschrieben. Diese Daten können auf die gewünschten Werte geändert werden.

Außerdem ist in den Quellprogrammen folgende Änderung vorzunehmen:

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

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

Der vom Benutzer modifizierte Name wird beim Aufruf der Bibliotheksfunktion `setlocale` zur Kennzeichnung der benutzerspezifischen Lokalität verwendet (als Zeichenkette im zweiten Parameter).

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

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

Aufbau der Daten für die einzelnen Lokalitätskategorien

LC_COLLATE

Die Sortierreihenfolge wird durch eine Tabelle (COLL/uscol) festgelegt, die den Sortierrang jedes Zeichens durch ein Gewicht definiert. Die Initialwerte sind die eigenen sedezimalen Werte der Zeichen, d.h. die Sortierreihenfolge entspricht der EBCDIC-Reihenfolge.

LC_CTYPE

Es gibt 3 Tabellen, die für alle EBCDIC-Zeichen die Klassifizierung und die Umwandlung von Groß- in Kleinbuchstaben bzw. von Klein- in Großbuchstaben festlegen.

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

	Assembler-Programm	C-Programm
Großbuchstabe	X'01'	_U
Kleinbuchstabe	X'02'	_L
Dezimalziffer	X'04'	_N
Zwischenraum	X'08'	_S
Sonderzeichen	X'10'	_P
Kontrollzeichen	X'20'	_C
Sedezimalzeichen	X'40'	_X

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

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

LC_NUMERIC, LC_MONETARY

Für alle Informationen vom Typ char * ist eine Zeichenkette von maximal 8 Zeichen vorgesehen. Diese Zeichenketten müssen immer mit dem Nullbyte abgeschlossen sein.

LC_TIME

Für die Wochentags- und Monatsnamen sind Zeichenketten von maximal 12 Zeichen vorgesehen.

6.7 Umgebungsvariablen

Umgebungsvariablen beeinflussen das Verhalten von Funktionen. Beim Programmstart wird ein Vektor von Zeichenketten zur Verfügung gestellt, der **Umgebung** genannt wird. Auf diesen Vektor wird durch die folgende externe Variable gezeigt:

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

Entsprechend dem XPG4-Standard haben diese Zeichenketten die Form "*name*=*value*", z.B. "LAST_BYTE_POINTER=YES".

Umgebungsvariablen im BS2000

Im BS2000 können Sie Umgebungsvariablen folgendermaßen vorbesetzen:

– Definieren Sie die S-Variable SYSPOSIX als Struktur:

```
/DECLARE-VARIABLE VARIABLE-NAME=SYSPOSIX(TYPE=*STRUCTURE)
```

– Belegen Sie die Umgebungsvariable *name* mit dem Wert *value*:

```
/SYSPOSIX.name='value'
```

Beachten Sie dabei:
Im BS2000 dürfen Variablennamen nur Großbuchstaben und keine Unterstriche (_) enthalten. Um eine Umgebungsvariable mit Unterstrich im Namen zu definieren, müssen Sie den Unterstrich durch einen Bindestrich (-) ersetzen. Die Umgebungsvariable LAST_BYTE_POINTER belegen Sie beispielsweise mit:

```
/SYSPOSIX.LAST-BYTE-POINTER='YES'
```

Beim Start eines Programms wird neben den Voreinstellungen für die Umgebung auch die S-Variable SYSPOSIX als Umgebungsdefinition ausgewertet. Für jede Variable vom Typ String, die in der Struktur SYSPOSIX enthalten ist, wird die Zeichenkette "*name*=*value*" in den globalen Datenbereich des Programms geschrieben, wobei *name* der Name der Umgebungsvariablen und *value* ihr Wert ist. Hierbei werden Bindestriche im Namen wiederum in Unterstriche umgewandelt, z.B. wird aus der S-Variablen SYSPOSIX.LAST-BYTE-POINTER die Umgebungsvariable LAST_BYTE_POINTER.

Nach dem Auslesen der S-Variable werden die Umgebungsvariablen ausschließlich im Programm verwaltet. Insbesondere haben dann Änderungen der S-Variablen keinen Einfluss mehr auf den Programmablauf. Während des Programmlaufs können Umgebungsvariablen nur mit der Funktion nur putenv geändert und mit getenv gelesen werden.

7 Nachschlageteil

Sie finden hier in alphabetischer Reihenfolge die Beschreibungen aller C-Funktionen und Makros, die das C-Laufzeitsystem zur Verfügung stellt.

Darstellung der Funktionsbeschreibungen

Die Funktionsbeschreibungen folgen alle demselben Prinzip, das im Folgenden erläutert werden soll.

Die Beschreibung einer Funktion ist in folgende Informationsabschnitte aufgeteilt:

- Funktionsname und deutsche Kurzbezeichnung in der Überschrift
- Definition und allgemeine Funktionsbeschreibung
- Parameter
- Returnwert
- Hinweise
- Satz-E/A
- Beispiel
- Siehe auch

Einige der o.g. Abschnitte können fehlen, wenn sie für die Funktion ohne Bedeutung sind bzw. wenn die Information (wie z.B. Datentyp der Parameter) bereits aus der Syntax des Funktionsaufrufes hervorgeht.

Definition und allgemeine Funktionsbeschreibung

Die Funktionsdefinition enthält folgende Informationen:

– den Namen der Include-Datei, die man für die Funktion benötigt,

– den Funktionskopf (Datentyp und Name der Funktion, Liste der formalen Parameter).

Nach der Definition (Include-Dateien/Funktionskopf) finden Sie die allgemeine Beschreibung der Arbeitsweise einer Funktion.

Parameter

Im Anschluss an die Beschreibung finden Sie bei komplexeren Funktionen eine Beschreibung der Parameter: Bedeutung, mögliche Werte, davon abhängige Wirkungsweise etc.

Es wird unterschieden zwischen Eingabe- und Ergebnisparametern. Im Unterschied zu Eingabeparametern wird bei Ergebnisparametern der Inhalt der beim Aufruf übergebenen Variablen durch die Funktion verändert. Man spricht hier auch von „impliziten“ Funktionsergebnissen. Ergebnisparameter sind als Zeiger auf ein Objekt ohne den Zusatz „const“ definiert. Für Ergebnisparameter müssen Sie beim Funktionsaufruf stets die Variablenadresse, d.h. ein Zeigerargument, angeben. Für Vektoren, Zeichenkettenvariablen und Strukturen ist außerdem genügend Speicherplatz bereitzustellen.

Returnwert

Hier werden die möglichen Funktionsergebnisse aufgelistet. Zeigt der Returnwert einen Fehler an, finden Sie als zusätzlichen Hinweis, ob und ggf. welcher Fehlercode in der Variablen `errno` abgespeichert wird.

Hinweise

In diesem Informationsabschnitt finden Sie folgende Informationen:

– mögliche Fehlerquellen (immer als erste Information),
– Programmier- und Anwendungstips,
– Zusammenhang mit anderen Funktionen,
– technische Details zur Arbeitsweise der Funktion,
– BS2000-Besonderheiten.

Satz-E/A

Diesen Informationsabschnitt finden Sie bei allen Ein-/Ausgabefunktionen, die auch auf Dateien mit satzorientierter Ein-/Ausgabe anwendbar sind. Er ergänzt die allgemeinen „Hinweise“ (vornehmlich für stromorientierte Ein-/Ausgabe formuliert) mit Besonderheiten, die bei Satz-E/A zu beachten sind.
Vgl. Abschnitt „Binärdatei“ auf Seite 62“, „Stromorientierte Ein-/Ausgabe“ auf Seite 67, „Satzorientierte-Ein-/Ausgabe“ auf Seite 66.

Beispiel

Kurzes Beispiel, das die Anwendung der beschriebenen Funktion illustriert.

Siehe auch

Verweise auf die Namen verwandter Funktionen.

_a2e, _e2a - von ASCII nach EBCDIC und EBCDIC nach ASCII konvertieren

Definition

```
#include <ascii_ebcdic.h>

char*_a2e (char* z);

char*_e2a (char* z);
```

Die Funktionen `_a2e`, `_e2a` konvertieren die als Parameter übergebene Zeichekette *z* (NULL-terminiert) von ASCII nach EBCDIC bzw. umgekehrt. Die Konvertierung erfolgt am Ort mithilfe von Umsetztabellen. Die entsprechenden Datenbereiche müssen also beschreibbar sein.

Die Umsetztabellen sind wie folgt deklariert:

```
unsigned char _a2e_tab[256];
unsigned char _e2a_tab[256];
```

Parameter

char* z
: Zeichenkette in ASCII- bzw. EBCDIC-Codierung, die konvertiert werden soll.

Returnwert Die als Parameter übergebene Zeichenkette *z* nach ihrer Konvertierung in EBCDIC- bzw. ASCII-Code

Siehe auch _a2e_n, _e2a_n, _a2e_max, _e2a_max, _a2e_dup, _e2a_dup, _a2e_dup_n, _e2a_dup_n

_a2e_dup, _e2a_dup - von ASCII nach EBCDIC und EBCDIC nach ASCII konvertieren

Definition

```
#include <ascii_ebcdic.h>

char*_a2e_dup (const char* z);

char*_e2a_dup (const char* z);
```

Die Funktionen `_a2e_dup`, `_e2a_dup` erzeugen eine neue Zeichenkette, die aus der als Parameter übergebenen Zeichenkette *z* hervorgeht durch Konvertierung von ASCII nach EBCDIC bzw. umgekehrt. Der Speicherplatz für die neue Zeichenkette wird mit `malloc()` allokiert, seine Freigabe liegt in der Verantwortung des Anwenders. Reicht der verfügbare Speicherplatz nicht aus, so wird NULL als Ergebnis geliefert, ansonsten die neue Zeichenkette.

Die Umsetztabellen sind wie folgt deklariert:

```
unsigned char _a2e_tab[256];
unsigned char _e2a_tab[256];
```

Parameter

char* z
: Zeichenkette in ASCII- bzw. EBCDIC-Codierung, die konvertiert werden soll

Returnwert

neue EBCDIC- bzw. ASCII-Zeichenkette (bei Erfolg)

NULL, falls Speicherplatz nicht ausreicht

Siehe auch _a2e, _e2a, _a2e_n, _e2a_n, _a2e_max, _e2a_max, _a2e_dup_n, _e2a_dup_n

_a2e_dup_n, _e2a_dup_n - von ASCII nach EBCDIC und EBCDIC nach ASCII konvertieren

Definition

```
#include <ascii_ebcdic.h>

char*_a2e_dup_n (const char* z, size_t n);

char*_e2a_dup_n (const char* z, size_t n);
```

Die Funktionen `_a2e_dup_n`, `_e2a_dup_n` erzeugen eine neue Zeichenkette, die aus *z* hervorgeht durch Umwandlung von genau *n* Zeichen von ASCII nach EBCDIC bzw. umgekehrt. Der Speicherplatz für die neue Zeichenkette wird mit `malloc()` allokiert, seine Freigabe liegt in der Verantwortung des Anwenders. Reicht der verfügbare Speicherplatz nicht aus, wird NULL als Ergebnis geliefert, ansonsten die neue, NULL-terminierte Zeichenkette.

Die Umsetztabellen sind wie folgt deklariert:

```
unsigned char _a2e_tab[256];
unsigned char _e2a_tab[256];
```

Parameter

const char* z
: Zeichenkette in ASCII- bzw. EBCDIC-Codierung, die konvertiert werden soll

size_t n
: Anzahl der zu konvertierenden Zeichen in der Zeichenkette *z*

Returnwert

Neue EBCDIC- bzw. ASCII-Zeichenkette (bei Erfolg)

NULL, falls Speicherplatz nicht ausreicht

Siehe auch

_a2e, _e2a, _a2e_max, _e2a_max, _a2e_n, _e2a_n, _a2e_dup; _e2a_dup

_a2e_max, _e2a_max - von ASCII nach EBCDIC und EBCDIC nach ASCII konvertieren

Definition

```
#include <ascii_ebcdic.h>

char*_a2e_max (char* z, size_t n);

char*_e2a_max (char* z, size_t n);
```

Die Funktionen `_a2e_max`, `_e2a_max` konvertieren die als Parameter übergebene Zeichenkette *z* maximal in der Länge *n* von ASCII nach EBCDIC bzw. umgekehrt. Enthält *z* an einer Position < *n* ein NULL-Zeichen, so wird die Konvertierung beendet. Die Konvertierung erfolgt am Ort mithilfe von Umsetztabellen. Die entsprechenden Datenbereiche müssen also beschreibbar sein.

Die Umsetztabellen sind wie folgt deklariert:

```
unsigned char _a2e_tab[256];
unsigned char _e2a_tab[256];
```

Parameter

char* z
: Zeichenkette in ASCII- bzw. EBCDIC-Codierung, die konvertiert werden soll

size_t n
: maximale Anzahl von Zeichen (linksbündig), die in *z* konvertiert werden sollen

Returnwert Die als Parameter übergebene Zeichenkette *z* nach ihrer Konvertierung in EBCDIC- bzw. ASCII-Code

Siehe auch _a2e, _e2a, _a2e_n, _e2a_n, _a2e_dup, _e2a_dup, _a2e_dup_n, _e2a_dup_n

_a2e_n, _e2a_n - von ASCII nach EBCDIC und EBCDIC nach ASCII konvertieren

Definition

```
#include <ascii_ebcdic.h>

char*_a2e_n (char* z, size_t n);

char*_e2a_n (char* z, size_t n);
```

Die Funktionen `_a2e_n`, `_e2a_n` konvertieren die als Parameter übergebene Zeichenkette *z* (NULL-terminiert) der Länge *n* von ASCII nach EBCDIC bzw. umgekehrt. Die Konvertierung erfolgt am Ort mithilfe von Umsetztabellen. Die entsprechenden Datenbereiche müssen also beschreibbar sein.

Die Umsetztabellen sind wie folgt deklariert:

```
unsigned char _a2e_tab[256];
unsigned char _e2a_tab[256];
```

Parameter

char* z
: Zeichenkette in ASCII- bzw. EBCDIC-Codierung, die konvertiert werden soll

size_t n
: Anzahl der zu konvertierenden Zeichen in der Zeichenkette *z*

Returnwert Die als Parameter übergebene Zeichenkette *z* nach ihrer Konvertierung nach EBCDIC bzw. ASCII

Siehe auch _a2e, _e2a, _a2e_max, _e2a_max, _a2e_dup, _e2a_dup, _a2e_dup_n, _e2a_dup_n

abort - Abnormaler Programmabbruch

Definition

```
#include <stdlib.h>

void abort(void);
```

`abort` löst das Signal SIGABRT aus. Wenn das Programm keine Routine zur Signalbehandlung vorsieht, oder wenn eine solche Routine zur unterbrochenen Stelle zurückspringt, wird das Programm mit `_exit(-1)` abgebrochen.
Ggf. mit `atexit` registrierte Beendigungsroutinen werden nicht aufgerufen und geöffnete Dateien nicht geschlossen.

Siehe auch atexit, exit, _exit, raise, signal

abs - Absolutbetrag einer ganzen Zahl

Definition

```
#include <stdlib.h>

int abs(int i);
```

`abs` berechnet den Absolutbetrag einer ganzen Zahl *i*.

Returnwert `|i|` für einen ganzzahligen Wert *i*.

Hinweis Der Absolutbetrag der betragsmäßig größten darstellbaren negativen Zahl ist nicht darstellbar. Wenn als Argument *i* die betragsmäßig größte negative Zahl (-2^{31}) angegeben wird, wird das Programm mit Fehler beendet.

Beispiel Folgendes Programm gibt zu einem eingelesenen Integer-Wert den entsprechenden Absolutbetrag aus.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int i;
  printf("Eine int-Zahl bitte: \n");
  if (scanf("%d", &i) == 1)          /* Überprüft Anzahl der Eingaben */
     printf("i = %d; ?i? = %d\n", i, abs(i));
  else
     printf("input error! \n");
  return 0;
}
```

Siehe auch cabs, fabs, labs, llabs

acos - Arcuscosinus

Definition #include <math.h>

double acos(double x);

`acos` (Arcuscosinus) ist die Umkehrfunktion zu `cos` und berechnet zu einer Zahl x aus dem Intervall [-1.0, +1.0] den entsprechenden Winkel im Bogenmaß.

Returnwert

`acos(x)`	eine Gleitkommazahl vom Typ `double` aus [0, pi] für Werte x aus dem Intervall [-1.0, +1.0].
0	für Werte außerhalb [-1.0, +1.0]. Zusätzlich wird `errno` auf EDOM gesetzt (domain error, d.h. Argument zu groß).

Beispiel Folgendes Programm druckt für Werte aus dem Intervall [0.0, 1.0] die entsprechenden Arcuscosinus-Werte:

```
#include <math.h>
#include <stdio.h>

int main(void)
{
  double x;
  for(x = 0.0; x < 1.0; x = x + 0.1)
  printf("x = %g : acos(%g) = %g\n", x, x, acos(x));
  return 0;
}
```

Siehe auch cos, sin, tan, asin, atan, atan2

alarm - Alarmuhr stellen

Definition #include <signal.h>

unsigned int alarm(unsigned int sek);

`alarm` löst nach einer als Argument übergebenen Zeitspanne *sek* das Signal SIGALRM beim aufrufenden Programm aus. SIGALRM entspricht der STXIT-Ereignisklasse RTIMER (Intervallzeitgeber Realzeit). Falls das Signal nicht abgefangen wird (siehe auch `signal`), wird das Programm mit `exit(-1)` beendet.
`alarm`-Aufrufe mit dem Wert 0 - `alarm(0)` - lösen keinen Alarm aus, sondern setzen die Alarmuhr auf 0 und löschen noch nicht erledigte Alarmanfragen.

Returnwert Restzeit in der Alarmuhr vor Ausführung des Alarmaufrufes.

Hinweise Mehrere aufeinander folgende `alarm`-Aufrufe setzen die Alarmuhr jedes Mal neu.

Da die Alarmuhr einen 1-Sekunden-Takt hat, kann es beim Auslösen des Signals zu Verschiebungen bis zu einer Sekunde kommen.

Wird das Signal abgefangen (siehe `signal`), kann sich das Wiederaufsetzen des unterbrochenen Programmes (bzw. Basisprozesses) aus Prioritätsgründen verzögern.

Mit der Zuweisung `i = alarm(0)` stellen Sie die Alarmuhr ab und können zudem feststellen, wieviel Zeit seit der letzten Alarmanforderung noch übrig gewesen wäre.

Beispiel Folgendes Programm schickt ca. alle zwei Sekunden einen Stern zur Standardausgabe.

```
#include <stdio.h>
#include <signal.h>
void f(int sig)        /* Signalbehandlung für SIGALRM */
{
  printf("*\n");
  alarm(2);            /* Neu stellen der Alarmuhr; alle weiteren Sterne */
}
int main(void)
{
  signal(SIGALRM + SIG_PS, f);
  alarm(2);            /* Erster Stern */
  for(;;)
    ;
  return 0;
}
```

Siehe auch signal, sleep

asctime - Datum mit Uhrzeit in englischer Darstellung

Definition #include <time.h>

char *asctime(const struct tm *tm_zg);

`asctime` wandelt eine gemäß der Struktur `tm` (s.u.) aufgeschlüsselte Zeitangabe in eine Zeichenkette um. Dabei wird nicht überprüft, ob es sich um eine sinnvolle Zeitangabe handelt, d.h. ob z.B. die angegebene Tageszahl zum angegebenen Monat passt. Ein Fehler liegt nur dann vor, wenn sich die eingegebenen Daten nicht im Zielformat darstellen lassen. So sind z.B. die kleinste darstellbare Jahreszahl -999 und die größte darstellbare Jahreszahl 9999.

Parameter const struct tm *tm_zg Struktur gemäß der include-Datei <time.h>:

```
struct tm
{
       int   tm_sec;        /* Sekunden (0-59) */
       int   tm_min;        /* Minuten (0-59) */
       int   tm_hour;       /* Stunden (0-23) */
       int   tm_mday;       /* Tag des Monats (1-31) */
       int   tm_mon;        /* Monate ab Jahresbeginn (0-11) */
       int   tm_year;       /* Jahre seit 1900 */
       int   tm_wday;       /* Wochentag (0-6, Sonntag=0) */
       int   tm_yday;       /* Tage seit dem 1. Januar (0-365) */
       int   tm_isdst;      /* Sommerzeitanzeige */
};
```

Returnwert Zeiger auf die erzeugte Zeichenkette,
Die Ergebniszeichenkette hat die Länge 26 (einschließlich des abschließenden Null-bytes \0) und das Format einer Datumsangabe mit Uhrzeit in Englisch:
Wochentag Monat Tag Std:Min:Sek Jahr,
z.B. `Fri Apr 29 12:01:20 2011\n\0`

NULL im Fehlerfall

Hinweise Die Funktionen `asctime`, `ctime`, `ctime64`, `gmtime`, `gmtime64`, `localtime` und `localtime64` schreiben ihre Ergebnisse in denselben C-internen Datenbereich, so dass der Aufruf einer dieser Funktionen das vorherige Ergebnis einer der anderen Funktionen überschreibt.

Eine Struktur vom Typ `tm` wird von den Funktionen `gmtime`, `gmtime64`, `localtime` und `localtime64` geliefert.

Die Aufrufe `asctime(localtime(sek_zg))` und `ctime(sek_zg)` sind äquivalent. Ebenso sind die Aufrufe `asctime(localtime64(sek_zg))` und `ctime64(sek_zg)` äquivalent.

Beispiel Ausgabe der aktuellen Zeit

```
#include <time.h>
#include <stdio.h>

struct tm *t;
char *s;
time_t clk;

int main(void)
{
   clk = time((time_t *) 0);
   t = gmtime(&clk);
   printf("Jahr: %d\n", t->tm_year + 1900);
   printf("Uhrzeit in Stunden: %d\n", t->tm_hour);
   printf("Jahrestag: %d\n", t->tm_yday);

   s = asctime(t);
   printf("%s", s);
   return 0;
}
```

Siehe auch ctime, ctime64, gmtime, gmtime64, localtime, localtime64, mktime, mktime64, time, time64

asin - Arcussinus

Definition #include <math.h>

double asin(double x);

`asin` ist die Umkehrfunktion zu `sin` und berechnet zu einer Zahl x aus dem Intervall [-1.0, +1.0] den entsprechenden Winkel im Bogenmaß.

Returnwert

arcussinus(x)	eine Gleitkommazahl vom Typ `double` aus [-pi/2, +pi/2] für Werte x aus dem Intervall [-1.0, +1.0].
0	für Werte außerhalb [-1.0, +1.0]. Zusätzlich wird `errno` auf EDOM gesetzt (domain error, d.h. Argument zu groß).

Beispiel Folgendes Programm berechnet für die Werte 0.0, 0.1,..., 1.0 die entsprechenden Arcussinus-Werte und gibt sie aus.

```
#include <math.h>
#include <stdio.h>

int main(void)
{
   double x;
   for(x = 0.0; x < 1.0; x = x + 0.1)
   printf("x = %g : asin(%g) = %g\n", x, x, asin(x));
   return 0;
}
```

Siehe auch sin, cos, acos, tan, atan, atan2

assert - Makro zur Fehlerdiagnose

Definition

```
#include <assert.h>

void assert(int ausdruck);
```

Dieses Makro stellt fest, ob ein Ausdruck *ausdruck* an einer bestimmten Programmstelle den Wert falsch (Null) hat. In diesem Fall wird das Programm mit `abort` beendet und folgender Kommentar auf die Standardfehlerausgabe (`stderr`) geschrieben:

`"CCM0009 Assertion failed: file` *xyz*`, line` *nnn*`"`

xyz ist der Name der Quelldatei, *nnn* ist die Zeilennummer, in der der `assert`-Aufruf steht.

Hinweis `assert`-Aufrufe werden aus dem Programm gestrichen (d.h. nicht ausgeführt), wenn Sie es mit folgender Compiler-Option übersetzen:

```
SOURCE-PROPERTIES = PARAMETERS(DEFINE = NDEBUG)
```

Siehe auch abort

atan - Arcustangens

Definition #include <math.h>

double atan(double x);

`atan` (Arcustangens) ist die Umkehrfunktion zu `tan` und berechnet zu einer Gleitkommazahl x den entsprechenden Winkel im Bogenmaß.

Returnwert `atan(x)` eine Gleitkommazahl vom Typ `double` aus dem Intervall [-pi/2, +pi/2].

Beispiel Folgendes Programm berechnet zu einem eingelesenen Wert den entsprechenden Arcustangens und gibt das Ergebnis aus.

```
#include <math.h>
#include <stdio.h>

int main(void)
{
  double x;
  printf("Von welcher Zahl wollen Sie den ATAN berechnen: \n");
  if( scanf("%lf", &x) == 1)   /* Überprüft die Eingabe einer Zahl */
    printf("x = %g : atan(%g) = %g\n", x, x, atan(x));
  return 0;
}
```

Siehe auch atan2, tan, sin, asin, cos, acos

atan2 - Arcustangens von x/y

Definition

```
#include <math.h>

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

`atan2` berechnet den Arcustangens von x/y. Die Vorzeichen der beiden Argumente bestimmen den Ergebnisquadranten.

Returnwert

`atan2(x/y)`	eine Gleitkommazahl vom Typ `double` aus dem Intervall [-pi/2, +pi/2]. Ist der Divisor y gleich 0, liefert `atan2` -pi/2 bzw. +pi/2, abhängig vom Vorzeichen des Dividenden.
0	falls der Dividend x gleich 0 ist.l
pi/2	falls beide Argumente gleich 0 sind. `errno` wird auf EDOM (domain error) gesetzt.

Beispiel Folgendes Programm liest die Argumente x und y ein und gibt den berechneten Arcustangens von x/y aus.

```
#include <math.h>
#include <stdio.h>

int main(void)
{
  double x, y;
  printf("Beispiel fuer ATAN2(x/y)\n");
  printf("bitte x und y eingeben:\n");
  if (scanf("%lf %lf", &x, &y) == 2)
       printf("ATAN2 (%g / %g) = %g\n", x, y, atan2(x, y));
  return 0;
}
```

Siehe auch atan, tan, sin, asin, cos, acos

atexit - Beendigungsroutinen registrieren

Definition

```
#include <stdlib.h>

int atexit(void (*funkt) (void));
```

Mit `atexit` wird eine Funktion *funkt* registriert, die bei normaler Programmbeendigung ausgeführt werden soll. Sie hat keine Parameter.

Returnwert

0	bei erfolgreicher Registrierung der Funktion.
≠ 0	bei Fehler.

Hinweise

Es können bis zu 40 Funktionen registriert werden. Die Funktionen werden in der umgekehrten Reihenfolge ihrer Registrierung aufgerufen. Wird eine Funktion mehrmals registriert, wird sie auch mehrmals aufgerufen.

Die mit `atexit` registrierten Funktionen werden nur aufgerufen, wenn das Programm auf eine der folgenden Arten „normal“ beendet wird:

- bei explizitem Aufruf der Funktion `exit`,
- bei Beendigung der `main`-Funktion ohne expliziten `exit`-Aufruf,
- bei Beendigung des Programms durch das C-Laufzeitsystem mit `exit(-1)`, das heißt bei Auftritt eines `raise`-Signals (nicht SIGABRT), das entweder nicht oder durch die `signal`-Standardfunktion SIG_DFL behandelt wird (siehe `signal`).

Erst nachdem alle Beendigungsroutinen abgearbeitet sind, werden ggf. noch geöffnete Dateien automatisch geschlossen.

Beispiel Die mit `atexit` registrierten Beendigungsroutinen *end1* und *end2* werden nach Beendigung der `main`-Funktion in der Reihenfolge *end2*, *end1* ausgeführt.

```
#include <stdlib.h>
#include <stdio.h>

void end1(void);
void end2(void);

int main(void)
{
   atexit(end1);
   atexit(end2);
   printf("main-Funktion\n");
   return 0;
}

void end1(void)
{
   printf("end1-Routine\n");
}

void end2(void)
{
   printf("end2-Routine\n");
}
```

Siehe auch exit, raise, signal

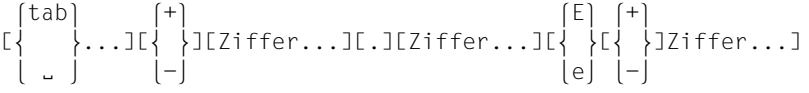
atof - Zeichenkette in Gleitkommazahl umwandeln (double)

Definition

```
#include <stdlib.h>

double atof(const char *s);
```

`atof` wandelt die Zeichenkette, auf die *s* zeigt, in eine Gleitkommazahl vom Typ `double` um. Die umzuwandelnde Zeichenkette kann wie folgt aufgebaut sein:



Für *tab* sind alle Steuerzeichen für „Zwischenraum“ zulässig (siehe Definition bei `isspace`).

Returnwert

Gleitkommazahl vom Typ `double`	für Zeichenketten, die eine wie oben beschriebene Struktur haben und einen Zahlenwert darstellen, der im zulässigen Gleitkommabereich liegt.
0	für Zeichenketten, die nicht der oben beschriebenen Syntax entsprechen.
HUGE_VAL	für Zeichenketten, deren Zahlenwert außerhalb des zulässigen Gleitkommabereichs liegt. `errno` wird auf ERANGE gesetzt (Resultat zu groß).

Hinweise

Das Dezimalpunktzeichen (Punkt oder Komma) in der umzuwandelnden Zeichenkette wird durch die Lokalität (Kategorie LC_NUMERIC) beeinflusst. Voreingestellt ist der Punkt.

`atof` erkennt auch Zeichenketten, die mit Ziffern beginnen, dann aber mit beliebigen Zeichen enden. atof schneidet den Ziffernteil ab, wandelt ihn gemäß obiger Beschreibung um und ignoriert den Rest.

Beispiel

Folgendes Programm wandelt eine beim Aufruf (Enter Options) übergebene Zeichenkette in die entsprechende Gleitkommazahl um.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
              /* Zahlen werden als Zeichenketten!!übergeben. Eine Umwandlung
                 ist erforderlich, falls der Zahlenwert benötigt wird */
{
  printf("floating : %f\n", atof(argv[1]));
  return 0; }
```

Siehe auch atoi, atol, strtod, strtol, strtoul

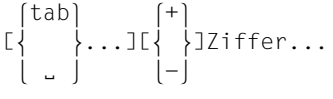
atoi - Zeichenkette in ganze Zahl umwandeln (int)

Definition

```
#include <stdlib.h>

int atoi(const char *s);
```

`atoi` wandelt eine Zeichenkette, auf die *s* zeigt, in eine ganze Zahl vom Typ `int` um. Die umzuwandelnde Zeichenkette kann wie folgt aufgebaut sein:



Für *tab* sind alle Steuerzeichen für „Zwischenraum“ zulässig (siehe Definition bei `isspace`).

Returnwert

Ganzzahliger Wert vom Typ `integer`
: für Zeichenketten, die eine wie oben beschriebene Struktur haben und einen Zahlenwert darstellen, der im zulässigen Integerbereich liegt.

0
: für Zeichenketten, die nicht der oben beschriebenen Syntax entsprechen.

INT_MAX bzw. INT_MIN
: bei Überlauf, abhängig vom Vorzeichen.

Hinweis

`atoi` erkennt auch Zeichenketten, die mit Ziffern beginnen, dann aber mit beliebigen Zeichen enden. `atoi` schneidet den Ziffernteil ab, wandelt ihn gemäß obiger Beschreibung um und ignoriert den Rest.

Beispiel

Folgendes Programm wandelt eine beim Aufruf (Enter Options) übergebene Zeichenkette in den entsprechenden ganzzahligen Wert um.

```
#include <stdio.h>
#include <stdlib.h>

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

                /* Zahlen werden als Zeichenkette!! übergeben.
                   Eine Umwandlung ist erforderlich,
                   falls der Zahlenwert benötigt wird. */
{
   printf("integer : %d\n", atoi(argv[1]));
   return 0;
}
```

Siehe auch

atof, atol, strtod, strtol, strtoul

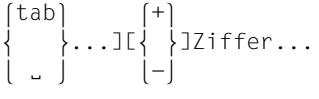
atol - Zeichenkette in ganze Zahl umwandeln (long)

Definition

```
#include <stdlib.h>

long int atol(const char *s);
```

`atol` wandelt eine Zeichenkette, auf die *s* zeigt, in eine ganze Zahl vom Typ `long int` um. Die umzuwandelnde Zeichenkette kann wie folgt aufgebaut sein:



Für *tab* sind alle Steuerzeichen für „Zwischenraum“ zulässig (siehe Definition bei `isspace`).

Returnwert

Ganzzahliger Wert vom Typ `long int`
: für Zeichenketten, die eine wie oben beschriebene Struktur haben und einen Zahlenwert darstellen.

0
: für Zeichenketten, die nicht der oben beschriebenen Syntax entsprechen.

LONG_MAX bzw. LONG_MIN
: bei Überlauf, abhängig vom Vorzeichen.

Hinweis

`atol` erkennt auch Zeichenketten, die mit Ziffern beginnen, dann aber mit beliebigen Zeichen enden. `atol` schneidet den Ziffernteil ab, wandelt ihn gemäß obiger Beschreibung um und ignoriert den Rest.

Beispiel

Folgendes Programm wandelt eine beim Aufruf (Enter Options) übergebene Zeichenkette in den entsprechenden ganzzahligen Wert um.

```
#include <stdio.h>
#include <stdlib.h>

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

                /* Zahlen werden als Zeichenkette!! übergeben.
                   Eine Umwandlung ist erforderlich,
                   falls der Zahlenwert benötigt wird. */
{
   printf("long integer : %ld\n", atol(argv[1]));
   return 0;
}
```

Siehe auch atof, atoi, atoll, strtod, strtol, strtoll, strtoul, strtoull

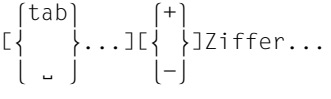
atoll - Zeichenkette in ganze Zahl umwandeln (long long int)

Definition

```
#include <stdlib.h>

long long int atoll(const char *s);
```

`atoll` wandelt eine Zeichenkette, auf die *s* zeigt, in eine ganze Zahl vom Typ `long long int` um. Die umzuwandelnde Zeichenkette kann wie folgt aufgebaut sein:



Für *tab* sind alle Steuerzeichen für „Zwischenraum“ zulässig (siehe Definition bei `isspace`).

Returnwert

Ganzzahliger Wert vom Typ `long long int`
: für Zeichenketten, die eine wie oben beschriebene Struktur haben und einen Zahlenwert darstellen.

0
: für Zeichenketten, die nicht der oben beschriebenen Syntax entsprechen.

LLONG_MAX bzw. LLONG_MIN
: bei Überlauf, abhängig vom Vorzeichen.

Hinweise

`atoll` erkennt auch Zeichenketten, die mit Ziffern beginnen, dann aber mit beliebigen Zeichen enden. `atoll` schneidet den Ziffernteil ab, wandelt ihn gemäß obiger Beschreibung um und ignoriert den Rest.

Ist *zg* ein NULL-Zeiger und *base* gleich 10, unterscheidet sich `atoll` von der Funktion `strtoll` nur durch die Fehlerbehandlung.
`atoll(s)` entspricht `strtoll(s, (char **)NULL, 10)`.

Der C-Compiler, der den Datentyp `long long` unterstützt, erzeugt nur Objekte im LLM-Format. Aus diesem Grunde werden auch die `long long`-Bibliotheksfunktionen nur als LLM's zur Verfügung gestellt und sind nicht in den Großmodulen enthalten. Sie müssen wie Datenmodule entweder fest eingebunden oder aus der Bibliothek nachgeladen werden.

Siehe auch

atof, atoi, atol, strtod, strtol, stroll, strtoul, stroull

bs2cmd - BS2000-Kommandos via CMD-Makro ausführen

Definition

```
#include <bs2cmd.h>

int bs2cmd(const char *cmd, bs2cmd_rc *rc, int maxoutput, int flag
        [, int *outbuflen, char *outbuf [, int *errbuflen, char *errbuf]]);
```

Mit `bs2cmd` kann ein BS2000-Kommando via CMD-Makro des BS2000 ausgeführt werden. Dabei können nur Kommandos verwendet werden, für die der CMD-Makro zugelassen ist. Insbesondere macht es keinen Sinn, Kommandos auszuführen, die zum Entladen des aufrufenden Programms führen, da die Schnittstelle keine Vorkehrungen enthält, mit denen dies verhindert werden kann.

Die Kommando-Ausgaben können optional gepuffert werden. In diesem Fall ist die Schnittstelle auch von einer rlogin-Task ohne SYSFILE-Umgebung nutzbar.

Parameter

const char *cmd

Dieser Parameter enthält das auszuführende Kommando oder eine Kommandoliste, in der die einzelnen Kommandos durch Semikolon getrennt angegeben werden. Bis auf Zeichenfolgen, die in Apostrophe eingeschlossen sind, werden alle Zeichen in cmd vor dem Aufruf in Großbuchstaben umgewandelt.

bs2cmd_rc *rc

rc ist ein Zeiger auf eine Struktur `bs2cmd_rc`, die Rückkehr-Informationen enthält.

Die Struktur `bs2cmd_rc` ist wie folgt aufgebaut:

```
typedef struct bs2cmd rc {
    unsigned char  subcode2;
    unsigned char  subcode1;
    unsigned short maincode;
    unsigned short progrc;
    char cmdmsg[8];
  } bs2cmd rc;
```

Falls beim Aufruf von `bs2cmd` an *rc* der NULL-Zeiger übergeben wird, werden keine Rückkehr-Informationen bereitgestellt.

int maxoutput

Dieser Parameter spezifiziert die Größe des anzulegenden Puffers für die Kommando-Ausgabe in Bytes. Bei der Wahl der Puffergröße ist zu beachten, dass zusätzlich zur eigentlichen Kommando-Ausgabe auch Verwaltungsinformationen ausgegeben werden.

Folgende Konstanten können angegeben werden:

BS2CMD_DEFAULT

Es wird ein Standard-Puffer von 256 K verwendet.

BS2CMD_NOBUFFER
Die Ausgaben werden nicht gepuffert. Unter rlogin-Tasks können mit dieser Einstellung Kommandos, die Ausgaben erzeugen, nur dann ausgeführt werden, wenn der Anwender einen Puffer bereitstellt (Angabe BS2CMD_FLAG_USER_BUFFER im Parameter *flag*).

Wenn der Puffer zu klein für die anstehenden Ausgaben gewählt wird, dann bricht die Kommando-Ausführung ab.

int flag
Dieser Parameter spezifiziert die Konfigurationsflags für die Schnittstelle. Derzeit können Sie die folgenden Flags oder Kombinationen von Flags (mit "|" verknüpft) angeben:

BS2CMD_FLAG_STRIP
Die Druckersteuerzeichen in der Kommando-Ausgabe werden vor der Ausgabe entfernt.

BS2CMD_FLAG_SPLIT
Die Kommando-Ausgaben werden auf stdout und stderr aufgeteilt. Meldungen werden nach stderr ausgegeben.

BS2CMD_FLAG_TRACE
Internes Debug-Flag zur Ausgabe der internen Puffer.

BS2CMD_FLAG_USER_BUFFER
`bs2cmd` wird mit einer variablen Parameterliste aufgerufen. Es werden die Parameter der variablen Parameterliste ausgewertet. Diese Parameter müssen vollständig angegeben werden, andernfalls ist das Verhalten der Funktion `bs2cmd` undefiniert.

Parameter der variablen Parameterliste:

Die folgenden Parameter ermöglichen Kommando-Ausgaben in einen vom Anwender bereitgestellten Speicherbereich, wenn im Parameter *flag* BS2CMD_FLAG_USER_BUFFER angegeben ist.

int *outbuflen
Länge des Speicherbereichs für stdout- Ausgaben. Nach Ausführung von `bs2cmd` steht in *outbuflen* die aktuelle Anzahl der nach `outbuf` geschriebenen Bytes oder -1, falls `outbuf` zu klein für die Ausgaben ist.

char *outbuf
Adresse des Speicherbereichs fuer stdout-Ausgaben.

int *errbuflen
: Länge des Speicherbereichs für stderr- Ausgaben. Nach Ausführung von `bs2cmd` steht in *errbuflen* die aktuelle Anzahl der nach `errbuf` geschriebenen Bytes oder -1, falls `errbuf` zu klein fuer die Ausgaben ist.

  **errbuflen* ist nur dann relevant, wenn im Parameter *flag* BS2CMD_FLAG_SPLIT angegeben ist.

char *errbuf
: spezifiziert die Adresse des Speicherbereichs fuer stderr- Ausgaben. *errbuf ist nur dann relevant, wenn BS2CMD_FLAG_SPLIT im Parameter *flag* gesetzt ist.

Hinweise

Die Meldungen werden, durch `\n` abgeschlossen, in den vom Anwender übergebenen Speicherbereich geschrieben. Abhängig von den Angaben im Parameter *flag* werden die Meldungen mit oder ohne Druckersteuerzeichen entweder nur nach `outbuf` geschrieben oder auf `outbuf` und `errbuf` verteilt.
Wenn die Größe der übergebenen Speicherbereiche dafür ausreicht, wird die Ausgabe durch \0 abgeschlossen.
Das \0 Byte wird bei der zurückgegebenen Länge nicht mitgerechnet.

Reicht die Größe der Speicherbereiche für die anfallenden Daten nicht aus, dann wird der Returnwert -1 zurückgeliefert und in `errno` der Wert EFBIG gesetzt. Um unterscheiden zu können, ob einer der benutzereigenen Speicherbereiche oder der interne Zwischenpuffer zu klein ist, wird in `outbuflen` bzw. `errbuflen` der Wert -1 eingetragen, wenn `outbuf` bzw. `errbuf` zu klein ist.

Wenn für *maxoutput* der Wert BS2CMD_NOBUFFER und gleichzeitig für *flag* der Wert BS2CMD_FLAG_USER_BUFFER angegeben ist, dann erfolgt keine interne Zwischenpufferung. Die Kommandoausgaben werden in diesem Fall direkt an den vom Anwender bereitgestellten Puffer `outbuf` geleitet. Der Aufbau der Ausgaben nach `outbuf` ist im Handbuch "Makroaufrufe an den Ablaufteil" beschrieben.

Achtung!

Im beschriebenen Fall muss die Adresse des Speicherbereichs auf Wortgrenze ausgerichtet sein. Bei Ausrichtungsfehler wird `errno` auf den Wert EFAULT gesetzt.

Wenn keine Zwischenpufferung erfolgt, können die flag-Werte BS2CMD_FLAG_STRIP und BS2CMD_FLAG_SPLIT nicht berücksichtigt werden. Eine Angabe dieser Werte wird ignoriert.

Returnwert `maincode` bei erfolgreicher Ausführung des Kommandos. `errno` wird nicht gesetzt.

-1 bei Fehler. `errno` wird auf einen der folgenden Werte gesetzt:

EINVAL
: Eines der Argumente hat einen unzulässigen Wert, z.B. ein leeres Kommando oder eine negative Puffergröße.

ENOMEM
: Der Speicherplatz für die anzulegenden Puffer reicht nicht aus.

EFAULT
: Nach der Kommando-Ausführung ist der Inhalt des Ausgabe-Puffers nicht interpretierbar oder Ausrichtungsfehler bei `outbuf`.

EFBIG
: Die Größe des Ausgabe-Puffers reicht nicht aus für die anfallenden Ausgaben.

Im Fehlerfall ist der Inhalt der Benutzerpuffer undefiniert.

bs2exit - Programmbeendigung mit MONJV

Definition

```
#include <stdlib.h>

void bs2exit(int status, const char *monjv_rcode);
```

`bs2exit` beendet das Programm.
Vorher werden alle vom Programm geöffneten Dateien geschlossen und folgende Meldungen auf `stderr` ausgegeben:

- „CCM0998 used CPU-time t seconds“, falls in der RUNTIME-Option CPU-TIME=YES gesetzt ist,
- „CCM0999 exit status“, falls *status* ≠ EXIT_SUCCESS (Wert 0) ist,
- „CCM0999 exit FAILURE“, falls *status* = EXIT_FAILURE (Wert 9990888) ist.

Die Zustandsanzeige der Monitor-Jobvariablen (1. - 3. Byte) wird wie bei der Funktion `exit` auf den Wert "$T " oder "$A " gesetzt, entsprechend dem ersten Parameter *status*.

Zusätzlich lässt sich mit dem Parameter *monjv_rcode* die Rückkehranzeige der MONJV (4. - 7. Byte) versorgen.

Parameter

int status
: siehe Funkfion `exit`.

const char *monjv_rcode:
: In diesem Parameter kann ein Zeiger auf eine 4 Byte lange Information (Rückkehrcode-Anzeige) angegeben werden, die bei Programmbeendigung in die MONJV aufgenommen wird.

Hinweise

Bei der Programmbeendigung mit `bs2exit` werden die mit `atexit` registrierten Beendigungsroutinen nicht aufgerufen (vgl. `exit`).

Um Monitor-Jobvariablen versorgen und abfragen zu können, müssen Sie das C-Programm mit folgendem Kommando starten:

`/START-PROG` *`programm`*`,MONJV=`*`monjvname`*

Der Inhalt der Jobvariablen lässt sich dann z.B. mit folgendem Kommando abfragen:

`/SHOW-JV JV-NAME(`*`monjvname`*`)`

Weitere Informationen zur Ablaufüberwachung mit MONJV finden Sie im Handbuch „Jobvariablen“.

Beispiel Programmbeendigung mit Setzen der Rückkehranzeige

```
#include <stdio.h>

int main(void)
{
  .
  .
  .
  if(fehler)
    bs2exit(-1, "ABCD");
}
```

Siehe auch exit, _exit

bs2fstat - Zugriff auf Dateinamen aus dem Katalog

Definition #include <stdlib.h>

int bs2fstat(const char *muster, void (*fkt)(const char *d_name, int len));

`bs2fstat` liefert

- den vollqualifizierten Dateinamen (:catid:$userid.dateiname) einer oder mehrerer Dateien, die das Auswahlkriterium *muster* erfüllen, sowie
- die Länge des jeweiligen Dateinamens inkl. dem abschließenden Nullbyte (\0).

Für jede gefundene Datei ruft `bs2fstat` eine vom Benutzer bereitzustellende Funktion *fkt* auf und übergibt an diese als aktuelle Argumente den jeweiligen Dateinamen *d_name* (Zeichenkette `char` *) und die Namenslänge *len* (ganze Zahl).

Wenn keine Datei dem Auswahlkriterium *muster* entspricht oder *muster* fehlerhaft ist, unterbleibt der Aufruf der Funktion *fkt*, und `bs2fstat` liefert als Returnwert eine DMS-Fehlermeldung.

Parameter const char *muster:

Zeichenkette, die das Auswahlkriterium für einen oder mehrere Dateinamen angibt. *muster* ist ein voll- oder teilqualifizierter Dateiname mit Wildcard-Syntax.
Außerdem können aus Kompatibilitätsgründen weitere Parameter angegeben werden, die die Auswahl der Dateien beeinflussen, z.B:

- Datei- und Katalogeigenschaften (FCBTYPE, SHARE etc.)
- Erstellungs- und Zugriffsdatum (CRDATE, EXDATE etc.)

Diese Parameter müssen in der Syntax des ISP-Kommandos FSTAT angegeben werden. Beispielsweise liefert das Muster "*,crdate=today" die Namen aller Dateien, die am jeweils aktuellen Tag ("heute") erstellt bzw. verändert wurden.

void (*fkt)(const char *d_name, int len)

Eine vom Benutzer bereitzustellende Funktion mit den Parametern *d_name* (Dateiname) und *len* (Namenslänge).
Diese Parameter werden von `bs2fstat` bei jedem Funktionsaufruf mit aktuellen Werten versorgt.
Die Funktionsaufrufe erfolgen durch `bs2fstat` automatisch (in einer `while`-Schleife).

Returnwert 0 bei erfolgreichem Aufruf.

DMS-Fehlermeldungscode
bei nicht erfolgreichem Aufruf.

Hinweise Der DMS-Fehlermeldungscode kann nur außerhalb der benutzereigenen Funktion *fkt* abgefragt werden, da bei erfolgloser Suche die Funktion nicht aufgerufen wird (siehe auch Beispiel).

Beispiel Folgendes Programm macht alle Dateien, die dem vom Benutzer eingegebenen Namensmuster entsprechen, mit dem MODIFY-FILE-ATTRIBUTES-Kommando shareable.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void share(const char *, int);

int main(void)
{
  char name[54];
  int result;
  printf("Welche Dateien shareable machen?\n");
  gets(name);
  result = bs2fstat(name, share);
  if(result != 0)
     printf("Fehlercode: DMS%x\n", result);
  return 0;
}

void share(const char *nam, int len)
                        /* die formalen Parameter nam und len werden von */
                        /* bs2fstat als aktuelle Parameter geliefert */

{
  char cmd[200];
  strcpy(cmd, "/MODIFY-FILE-ATTRIBUTES ");
  strcat(cmd, nam);
  strcat(cmd, ",PROTECTION=PAR(USER-ACCESS=ALL-USERS)");
  system("/MODIFY-TERMINAL-OPTIONS OVERFLOW-CONTROL=NO-CONTROL");
  printf("%s\n", cmd);
  system(cmd);
}
```

Siehe auch system

bsearch - Binärer Suchalgorithmus

Definition

```
#include <stdlib.h>

void *bsearch(const void *such, const void *feld, size_t anz,
              size_t elgroesse, int (*vergl) (const void *, const void *));
```

Die Funktion `bsearch` ist eine binäre Suchfunktion. `bsearch` durchsucht *anz* Elemente eines Vektors *feld* nach dem Wert im Datenelement *such*. Jedes Vektorelement ist *elgroesse* Bytes lang. Die Vektorelemente müssen bereits aufsteigend sortiert vorliegen, und zwar analog zu der Vergleichsfunktion *vergl*.

vergl ist eine vom Benutzer bereitzustellende Vergleichsfunktion, die von bsearch jeweils mit zwei Argumenten aufgerufen wird, einem Zeiger auf *such* (Argument 1) und einem Zeiger auf ein Vektorelement (Argument 2). *vergl* liefert eine ganze Zahl als Ergebnis, die wie folgt gedeutet wird:

< 0	Argument1 ist kleiner als Argument2
= 0	Argument1 und Argument2 sind gleich
> 0	Argument1 ist größer als Argument2

Returnwert Zeiger auf das gefundene Vektorelement.
Ist das gesuchte Element mehrmals vorhanden, ist nicht festgelegt, auf welches Element der Zeiger zeigt.

NULL-Zeiger falls kein Element gefunden wurde.

Hinweis Wird für die Sortierung des Vektors z.B. die Funktion `qsort` verwendet, ist es sinnvoll, dieselbe Vergleichsfunktion *vergl* zu verwenden, die von `bsearch` benutzt wird. Die aktuellen Argumente von `qsort` sind dann Zeiger auf zwei zu vergleichende Vektorelemente.

Siehe auch qsort

btowc - (ein-byte) Multibyte-Zeichen in Langzeichen konvertieren

Definition
```
#include <stdio.h>
#include <wchar.h>

wint_t btowc(int c);
```

`btowc` konvertiert das Multibyte-Zeichen *c*, das aus einem Byte besteht und sich im „initial shift“-Zustand befinden muss, in ein Langzeichen.

Returnwert

Langzeichen	bei Erfolg.
WEOF	falls *c* den Wert EOF enthält oder `(unsigned char)c` kein gültiges (1-Byte) Multibyte-Zeichen im „initial shift“-Zustand darstellt.

Hinweis
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen bzw. Multibyte-Zeichen unterstützt.
Der Shift-Zustand des Multibyte-Zeichens wird ignoriert.

Siehe auch mblen, mbtowc, wcstombs, wctomb

cabs - Absolutbetrag einer komplexen Zahl

Definition

```
#include <math.h>

double cabs(_ _complex z);
```

`cabs` berechnet den Absolutbetrag einer komplexen Zahl |z mit Realteil x und Imaginärteil y.

_ _complex ist ein in <math.h> vordefinierter Typ:

```
#typdef struct{double x, y;} __complex
```

Returnwert `sqrt(z.x * z.x + z.y * z.y)`
das ist der Absolutbetrag der komplexen Zahl z.

Bei Überlauf bricht das Programm ab (Signal SIGFPE)!

Beispiel Folgendes Programm berechnet den Absolutbetrag einer komplexen Zahl.

```
#include <stdio.h>
#include <math.h>

int main(void)
{
 __complex z;
 if (scanf("%f %f", &z.x, &z.y) == 2)
     printf("%f : Absolutbetrag\n", cabs(z));
 return 0;
}
```

Siehe auch abs, fabs, labs, llabs, sqrt

calloc - Speicherplatz reservieren

Definition

```
#include <stdlib.h>

void *calloc(size_t anz, size_t elgroesse);
```

`calloc` beschafft zur Ausführungszeit zusammenhängenden Speicherplatz für einen Vektor mit *anz* Elementen, wobei jedes Element *elgroesse* Bytes beansprucht. `calloc` initialisiert jedes Element des neuen Vektors mit binären Nullen.

`calloc` ist Teil eines C-spezifischen Speicherverwaltungspaketes, das angeforderte und wieder freigegebene Speicherbereiche intern verwaltet. Neue Anforderungen werden zuerst aus bereits verwalteten Bereichen zu erfüllen versucht, dann erst vom Betriebssystem (vgl. Funktion `garbcoll`).

Returnwert

Zeiger auf den neuen Speicherplatz
falls genügend Speicherplatz vorhanden ist.

NULL-Zeiger falls der Speicherplatz für die Anforderung nicht ausreicht.

Hinweise

Der neue Datenbereich beginnt auf Doppelwortgrenze.

Um sicherzugehen, dass Sie die richtige Größe für ein Vektorelement anfordern, sollten Sie für die Berechnung von *elgroesse* den Operator `sizeof` verwenden.

Wird die Länge des zur Verfügung gestellten Speicherbereiches beim Schreiben überschritten, führt dies zu schwerwiegenden Fehlern im Arbeitsspeicher.

Falls *anz* oder *elgroesse* den Wert 0 besitzt, liefert `calloc` eine eindeutige Adresse, die auch an `free` übergeben werden kann.

Beispiel

Folgender Programmausschnitt fordert Speicherplatz für 20 Vektorelemente vom Typ `long integer` an.

```
#include <stdlib.h>

long *long_array;
    .
    .
long_array = (long *)calloc(20, sizeof(long));
```

Siehe auch

malloc, realloc, free, garbcoll

cdisco - Abmelden einer Contingency-Routine

Definition #include <cont.h>

void cdisco(struct enacop *enacopar);

`cdisco` meldet eine mit `cenaco` definierte Contingency-Routine (TU bzw. P1) ab.

Ausführliche Informationen zu Contingency-Routinen finden Sie im Kapitel 5 und im Handbuch „Makroaufrufe an den Ablaufteil“.

Parameter struct enacop *enacopar

Zeiger auf eine Struktur, die in <cont.h> wie folgt definiert ist:

```
struct enacop
{
  char resrv1 [7];                /* reserved for int. use */
  char coname [54];               /* name of cont. routine */
  char resrv2 [15];               /* reserved for int. use */
  char level;                     /* priority of cont.rout. */
  int  (*econt)(struct contp);    /* start adr of cont.rout. */
  int  comess;                    /* contingency message */
  int  coidret;                   /* contingency identifier */
  errcod secind;                  /* secondary indicator */
  char resrv3 [2];                /* reserved for int. use */
  errcod rcode1;                  /* return code */
};
#define errcod     char
#define _norm      0              /* normterm */
#define _abnorm    4              /* abnormend */
#define _enabled   4              /* codefenabled */
#define _preven    12             /* coprevenabled */
#define _parerr    16             /* coparerror */
#define _maxexc    24            /* comaxexceed */
```

`cdisco` wertet nur den Eintrag `coidret` (Kurzkennung des Contingency-Prozesses) in der Struktur aus.

Einträge, die von `cdisco` versorgt werden:

secind „Secondary Indicator“, wie er nach Ausführung des DISCO-Makros im höchstwertigen Byte des Register 15 abgelegt wird (Werte X’10’ oder oder X’16’).

rcode1 „Return Code“, wie er nach Ausführung des DISCO-Makros im niedrigstwertigen Byte des Register 15 abgelegt wird (Werte 0 oder 4).

Hinweis Der Assembler-Makro DISCO sperrt die Contingency-Routine lediglich für zukünftige Ereingnisanforderungen. Tritt jedoch nach DISCO noch ein bereits vorher angefordertes Ereignis ein, wird die Contingency-Routine auch nach DISCO aufgerufen.
Aufrufe der Contingency-Routine `econt` werden jedoch auch für bereits angeforderte Ereignisse unterdrückt.

Siehe auch cenaco

ceil - Aufrunden

Definition #include <math.h>

double ceil(double x);

`ceil` rundet die Gleitkommazahl x nach oben (ganzzahlig) auf.

Returnwert Kleinste ganze Zahl vom Typ `double`, die größer oder gleich x ist
bei Erfolg

HUGE_VAL bei Überlauf. Zusätzlich wird `errno` auf ERANGE gesetzt (Resultat zu groß).

Beispiel
```
#include <math.h>
#include <stdio.h>

int main(void)
{
  double x;
  printf("Bitte geben Sie die aufzurundende Gleitkommazahl ein:\n");
  if (scanf("%lf", &x) == 1)
      printf("Die Zahl %g wird aufgerundet zu %f\n", x, ceil(x));
  return 0;
}
```

Siehe auch abs, fabs, floor

cenaco - Definition einer Contingency-Routine

Definition #include <cont.h>

void cenaco(struct enacop *enacopar);

`cenaco` definiert eine Contingency-Routine (TU bzw. P1), d.h. eine vom Anwender geschriebene Routine wird damit als Contingency-Routine angemeldet.

Ausführliche Informationen zu Contingency-Routinen finden Sie im Kapitel 5 und im Handbuch „Makroaufrufe an den Ablaufteil“.

Parameter struct enacop *enacopar

Zeiger auf eine Struktur, die in <cont.h> wie folgt definiert ist:

```
struct enacop
{
  char resrv1 [7];                /* reserved for int. use */
  char coname [54];               /* name of cont. routine */
  char resrv2 [15];               /* reserved for int. use */
  char level;                     /* priority of cont.rout. */
  int  (*econt)(struct contp);    /* start adr of cont.rout. */
  int  comess;                    /* contingency message */
  int  coidret;                   /* contingency identifier */
  errcod secind;                  /* secondary indicator */
  char resrv3 [2];                /* reserved for int. use */
  errcod rcode1;                  /* return code */
};

#define errcod     char
#define _norm      0      /* normterm */
#define _abnorm    4      /* abnormend */
#define _enabled   4      /* codefenabled */
#define _preven    12     /* coprevenabled */
#define _parerr    16     /* coparerror */
#define _maxexc    24     /* comaxexceed */
```

Einige Einträge der Parameterstruktur müssen bzw. können Sie vor dem `cenaco`-Aufruf selbst versorgen, in anderen Einträgen legt `cenaco` während des Ablaufs Informationen ab.

Einträge, die vom Anwender versorgt werden:

coname
: Name des Contingency-Prozesses. Der Name ist max. 54 Byte lang (ohne Nullbyte), muss in Großbuchstaben geschrieben und mit mindestens einem Leerzeichen abgeschlossen werden (ein Nullbyte unmittelbar hinter dem eigentlichen Namen wird vom System nicht als Endekriterium erkannt). Für die Versorgung von *coname* eignet sich z.B. die Funktion `strfill` (siehe auch Beispiel).
Die Versorgung ist obligatorisch.

level
: Prioritätsstufe des Contingency-Prozesses. Die Versorgung ist obligatorisch. Es sind Werte von 1 - 126 zulässig.

econt
: Startadresse der Contingency-Routine. Die Versorgung ist obligatorisch.

comess
: Contingency-Message. Die Versorgung ist fakultativ. Der Wert wird als Parameter an die Contingency-Routine übergeben.

Einträge, die von `cenaco` versorgt werden:

coidret
: Kurzkennung des Contingency-Prozesses. Diese Kurzkennung muss in weiteren Makros (z.B. SOLSIG) zur Bezeichnung des Contingency-Prozesses verwendet werden.

secind
: „Secondary Indicator“, wie er nach Ausführung des ENACO-Makros im höchstwertigen Byte des Register 15 abgelegt wird (Werte 4, 12, 16 oder 24).

rcode1
: „Return Code“, wie er nach Ausführung des ENACO-Makros im niedrigstwertigen Byte des Register 15 abgelegt wird (Werte 0 oder 4).

Hinweis Es können maximal 255 Contingency-Routinen definiert werden.

Beispiel Programmausschnitt zur Definition einer Contingency-Routine.

```
#include <cont.h>

/* Contingency-Routine: controut */

int controut(struct contp contpar)
{
    .
    .
    .
  printf("Contingency-Message: %d\n", contpar.comess);
    .
    .
    .
}

/* main-Routine, in der die Routine controut als Contingency-Routine
   definiert wird. */

int main(void)
{
    .
    .
    .
  struct enacop enacopar;
    .
    .
    .
  enacopar.econt  = controut;
  enacopar.level  = 1;
  enacopar.comess = 100;
  strfill(enacopar.coname, "CONTPROC1 ", sizeof(enacopar.coname));
  cenaco(&enacopar);
    .
    .
    .
}
```

Siehe auch cdisco, cstxt, signal, alarm, raise, sleep

clearerr - Dateiende- und Fehlerflag löschen

Definition

```
#include <stdio.h>
void clearerr(FILE *dz);
```

`clearerr` löscht die Dateiende- und Fehlerinformationen der Datei mit Dateizeiger *dz*.

Hinweis `clearerr` ist sowohl als Makro als auch als Funktion realisiert (siehe Abschnitt „Funktionen und Makros“ auf Seite 19).

Satz-E/A `clearerr` ist auch auf Dateien mit Satz-E/A unverändert anwendbar.

Siehe auch feof, ferror

clock - CPU-Zeitverbrauch seit Programmaufruf

Definition

#include <time.h>

clock_t clock(void);

`clock` liefert die seit dem Programmaufruf vergangene CPU-Zeit.

Returnwert

Die seit dem Programmaufruf vergangene CPU-Zeit in Zehntausendstel Sekunden bei Erfolg.

`(clock_t)-1` wenn die Zeit nicht errechen- bzw. darstellbar ist.

Hinweise

`clock` ist sowohl als Makro als auch als Funktion realisiert (siehe Abschnitt „Funktionen und Makros“ auf Seite 19).

Um die Zeit in Sekunden zu erhalten, muss man das Ergebnis von `clock` durch den Wert des Makros CLOCKS_PER_SEC dividieren.

Beispiel

```
#include <time.h>
#include <stdio.h>

int main(void)
{
  clock_t result;
  result = clock();
  printf("used cputime %f seconds\n", ((float)result / CLOCKS_PER_SEC));
  return 0;
}
```

Siehe auch

cputime

close - Datei schließen und Puffer bereinigen (elementar)

Definition

```
#include <stdio.h>

int close(int dk);
```

`close` schließt eine Datei, die mit `open`/`open64` oder `creat`/`creat64` geöffnet wurde. Bevor die Datei geschlossen wird, ruft `close` die Funktion `fflush` (Puffer bereinigen) auf.

Returnwert

0	`close` hat die Datei mit der Dateikennzahl *dk* geschlossen.
-1	Die Dateikennzahl ist unbekannt bzw. für diese Dateikennzahl ist keine Datei geöffnet. Zusätzlich wird `errno` auf EBADF gesetzt (unzulässige Dateikennzahl).

Hinweise

Bei Beendigung eines Programms (normal oder mit `exit`) werden automatisch alle offenen Dateien geschlossen.

In einem Programm dürfen maximal _NFILE Dateien gleichzeitig geöffnet sein. _NFILE ist in <stdio.h> mit 2048 definiert. Programme, die mehr Dateien verarbeiten, müssen daher zwischenzeitlich nicht benötigte Dateien schließen.

Wurde die Datei mit der Standard-Ein-/Ausgabefunktion `fopen` bzw. `fopen64` eröffnet, muss sie statt mit `close` mit `fclose` geschlossen werden.

Beispiel

siehe Beispiel bei `lseek`/`lseek64`

Siehe auch

creat, creat64, fclose, fflush, open, open64, exit

cos - Cosinus

Definition #include <math.h>

double cos(double x);

`cos` berechnet für die Gleitkommazahl x die trigonometrische Funktion Cosinus.

Returnwert `cos(x)` Gleitkommazahl im Intervall [-1.0, +1.0].

Beispiel Folgendes Programm gibt für Werte aus [-pi, +pi] die entsprechenden Cosinus-Werte aus.

```
#include <math.h>
#include <stdio.h>
#define pi 3.14159265358979

int main(void)
{
   double x;
   for (x = -pi; x <= pi; x = x + pi/4.)
       printf("cos(%f) = %f\n", x, cos(x));
   return 0;
}
```

Siehe auch acos, cosh, sin, asin, sinh, tan, atan, atan2, tanh

cosh - Cosinus hyperbolicus

Definition

```
#include <math.h>

double cosh(double x);
```

`cosh` berechnet den Cosinus hyperbolicus für die Gleitkommazahl x.

Returnwert

`cosh(x)`	für die Gleitkommazahl x.
+HUGE_VAL	bei Überlauf. Zusätzlich wird `errno` auf ERANGE gesetzt (Resultat zu groß).

Beispiel

Folgendes Programm gibt für Werte aus [-1.0, +1.0] die entsprechenden Cosinus hyperbolicus-Werte aus.

```
#include <math.h>
#include <stdio.h>

int main(void)
{
  double x;
  for (x = -1.0; x < 1.0; x = x + 0.1)
    printf("cosh(%f) = %f\n", x, cosh(x));
  return 0;
}
```

Siehe auch acos, cos, sin, asin, sinh, tan, atan, atan2, tanh

cputime - CPU-Zeitverbrauch der aktuellen Task

Definition

```
#include <stdlib.h>
int cputime(void);
```

`cputime` liefert den CPU-Zeitverbrauch der momentanen Task (seit LOGON).

Returnwert Ganze Zahl, die den CPU-Zeitverbrauch in Zehntausendstel Sekunden angibt.

Beispiel

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  float time_f;
  time_f = (float)cputime() / 10000;
  printf("cputime since logon: %f seconds\n", time_f);
  return 0;
}
```

creat, creat64 - Datei neu anlegen (elementar)

Definition

```
#include <stdio.h>

int creat(const char *d_name, int modus);
int creat64(const char *d_name, int modus);
```

`creat` und `creat64` öffnen eine Datei zum Neuschreiben:

- Wenn die Datei noch nicht vorhanden ist, wird sie neu angelegt.
- Bereits existierende Dateien werden auf die Länge 0 verkürzt.

`creat` und `creat64` liefern eine Dateikennzahl für spätere elementare Zugriffsoperationen (`write`, `read`).

Es besteht kein funktionaler Unterschied zwischen `creat` und `creat64`, außer dass in der mit dem Filedescriptor verknüpften Dateibeschreibung das Kennzeichen für eine große Datei hinterlegt wird, dh. es wird das O_LARGEFILE Bit gesetzt. Es wird eine Dateikennzahl zurückgegeben, die dazu verwendet werden kann, die Datei über 2 GB hinaus zu vergrößern.

Für die Bearbeitung von Dateien > 2 GB verfahren Sie wie folgt:

- Falls das Define `_FILE_OFFSET_BITS 64` (siehe Seite 71) gesetzt ist, rufen Sie `creat` auf. Implizit wird dann `creat64` mit den passenden Parametern verwendet.
- Andernfalls müssen Sie `creat64` aufrufen.

Parameter

const char *d_name
: Zeichenkette, die den Namen der zu öffnenden Datei angibt. *d_name* kann sein:
  - jeder gültige BS2000-Dateiname
  - "link=*linkname*"
    linkname bezeichnet einen BS2000-Linknamen

int modus
: Im BS2000 wird hier nur der *lbp*-Schalter ausgewertet. Alle anderen Angaben in diesem Parameter werden ignoriert. Zur Erstellung von portierbaren Programmen sind sie jedoch notwendig, da sie im UNIX-Betriebssystem die Schutzbitvergabe regeln.

lbp-Schalter

Der *lbp*-Schalter steuert die Behandlung des Last Byte Pointers (LBP). Er ist nur für Binärdateien mit Zugriffsart PAM relevant und kann mit allen bei `open` zulässigen Angaben kombiniert werden. Er wirkt sich erst aus, wenn die Datei geschlossen wird.

Beim Öffnen und Lesen einer bestehenden Datei wird der LBP unabhängig vom *lbp*-Schalter immer berücksichtigt:

– Ist der LBP der Datei ungleich 0, wird er ausgewertet. Ein eventuell vorhandener Marker wird ignoriert.

– Ist der LBP = 0, wird nach einem Marker gesucht und die Dateilänge daraus ermittelt. Falls kein Marker gefunden wird, wird das Ende des letzten vollständigen Blocks als Dateiende betrachtet.

`O_LBP`
: Beim Schließen einer Datei, die verändert oder neu erstellt wurde, wird kein Marker geschrieben (auch wenn einer vorhanden war) und ein gültiger LBP gesetzt. Auf diese Weise können Dateien mit Marker auf LBP ohne Marker umgestellt werden.

`O_NOLBP`
: Beim Schließen einer Datei, die verändert oder neu erstellt wurde, wird der LBP auf Null gesetzt. Für eine neu erstellte Datei wird immer ein Marker geschrieben, für eine veränderte nur dann, wenn vorher bereits ein Marker vorhanden war. War kein Marker vorhanden, wird auch keiner geschrieben und die Datei endet mit dem vollständigen letzten Block.

Wird der *lbp*-Schalter in beiden Varianten angegeben (`O_LBP` und `O_NOLBP`), so schlägt die Funktion `creat, creat64` fehl und `errno` wird auf `EINVAL` gesetzt.

Wird der *lbp*-Schalter nicht angegeben, hängt das Verhalten von der Umgebungsvariablen LAST_BYTE_POINTER ab (siehe auch „Umgebungsvariable LAST_BYTE_POINTER“ auf Seite 90):

`LAST_BYTE_POINTER=YES`
: Die Funktion verhält sich so, als ob `O_LBP` angegeben wäre.

`LAST_BYTE_POINTER=NO`
: Die Funktion verhält sich so, als ob `O_NOLBP` angegeben wäre.

Returnwert

Dateikennzahl	positive Zahl, die später bei den elementaren Zugriffsoperationen (`write`, `read`) zum Bezeichnen der Datei benutzt wird.
-1	wenn die Datei nicht geöffnet werden konnte, z.B. weil zuviele Dateien geöffnet sind oder *d_name* kein gültiger Datei- bzw. Linkname ist.

Hinweise

Der BS2000-Dateiname bzw. -Linkname kann in Klein- und Großbuchstaben geschrieben werden, er wird automatisch in Großbuchstaben umgesetzt.

Wird eine nicht vorhandene Datei neu angelegt, wird standardmäßig folgende Datei erzeugt:
Bei KR-Funktionalität (nur bei C/C++ Versionen kleiner V3.0 vorhanden) eine SAM-Datei mit variabler Satzlänge und Standardblocklänge,
bei ANSI-Funktionalität eine ISAM-Datei mit variabler Satzlänge und Standardblocklänge.

Bei Verwendung eines Linknamens lassen sich mit dem ADD-FILE-LINK-Kommando folgende Dateiattribute ändern: Zugriffsmethode, Satzlänge, Satzformat, Blocklänge und Blockformat. Siehe auch Abschnitt „Systemdateien (SYSDTA, SYSOUT, SYSLST)“ auf Seite 73.

Wird eine bereits existierende Datei auf die Länge 0 verkürzt, bleiben die Katalogeigenschaften dieser Datei erhalten.

Es können maximal _NFILE Dateien gleichzeitig geöffnet sein. _NFILE ist in <stdio.h> mit 2048 definiert.

Beispiel Folgendes Programm schreibt den Inhalt einer Eingabedatei in eine Ausgabedatei. Die Ausgabedatei wird mit `creat` neu angelegt. Der Name dieser Datei sowie die Dateieigenschaften werden mit einem ADD-FILE-LINK-Kommando (Linkname=LINK) festgelegt. Mit folgendem Kommando würde z.B. eine ISAM-Datei mit dem Namen OUT.ISAM erstellt:

```
/ADD-FILE-LINK LINK-NAME=LINK,FILE-NAME=OUT.ISAM,ACCESS-METHOD=ISAM
```

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char name[50];
   char buf;
   int fin, fout;

   printf("Name der Eingabedatei?\n");
   gets(name);
   printf("Die Datei %s wird kopiert.\n", name);
   if ((fin = open(name,0)) == -1)
   {
     perror(name);
     exit(-1);
   }
   if ((fout = creat("link=link", 1)) == -1)
   {
     perror("link");
     exit(-1);
   }
   while(read(fin, &buf, 1) > 0)
   {
     putchar(buf);           /* Protokoll auf stdout */
     write(fout, &buf, 1);
   }
   close(fin); close(fout);
   return 0;
}
```

Siehe auch close, fdopen, open, open64, read, write, perror

cstxit, _cstxit - Definition einer STXIT-Routine

Definition

```
#include <stxit.h>

void cstxit(struct stxitp stxitpar);
void _cstxit(struct stxitp *stxitpar);
```

`cstxit` definiert eine STXIT-Routine, d.h. eine vom Anwender geschriebene Routine wird damit als STXIT-Routine angemeldet.
Ausführliche Informationen zur Programmierung von STXIT-Routinen finden Sie im Kapitel „Contingency- und STXIT-Routinen“ auf Seite 93 und im Handbuch „Makroaufrufe an den Ablaufteil“.

`_cstxit` definiert ebenfalls eine STXIT-Routine.
Im Unterschied zu `cstxit` wird jedoch nicht die Struktur `stxitpar` selbst, sondern ein Zeiger auf diese übergeben. Somit kann die Funktion `_cstxit` das Strukturelement `err_set` so setzen, dass es vom aufrufenden Programm ausgewertet werden kann.

Parameter

struct stxitp stxitpar

Struktur, in der die notwendigen Informationen für die Definition einer STXIT-Routine anzugeben sind. Die Struktur ist in <stxit.h> für ANSI-C wie folgt definiert (`struct cont` ist abhängig vom Übersetzungsmodus):

```
struct stxitp
{
  addr     bufadr;      /* Adresse der Mitteilung an das  */
                        /*  Programm (OPINT)*/
  err_set retcode;      /* Returncode */
  struct cont contp;    /* Adresse der STXIT-Routinen */
  struct nest nestp;    /* max. Schachtelungstiefe */
  struct stx  stxp;     /* Steuerung des cstxit-Aufrufs */
  struct diag diagp;    /* Diagnosesteuerung */
  struct type typep;    /* Parameterübergabe-Modus */
};

struct cont             /* Adresse der STXIT-Routine für */
{                       /* die jeweilige Ereignisklasse */
  void  (*prchk) (struct stxcontp stxcontpar);
  void  (*timer) (struct stxcontp stxcontpar);
  void  (*opint) (struct stxcontp stxcontpar);
  void  (*error) (struct stxcontp stxcontpar);
  void  (*runout) (struct stxcontp stxcontpar);
  void  (*brkpt) (struct stxcontp stxcontpar);
  void  (*abend) (struct stxcontp stxcontpar);
  void  (*pterm) (struct stxcontp stxcontpar);
  void  (*rtimer) (struct stxcontp stxcontpar);
```

```
};

struct nest             /* max. Schachtelungstiefe für */
{                       /* die jeweilige Ereignisklasse */
  char prchk;
  char timer;
  char opint;
  char error;
  char runout;
  char brkpt;
  char abend;
  char pterm;
  char rtimer;
  char filler;
};


struct stx              /* Steuerung des cstxit-Aufrufs für */
{                       /* die jeweilige Ereignisklasse */
  stx_set prchk;
  stx_set timer;
  stx_set opint;
  stx_set error;
  stx_set runout;
  stx_set brkpt;
  stx_set abend;
  stx_set pterm;
  stx_set rtimer;
  stx_set filler;
};

struct diag             /* Diagnosesteuerung für die */
{                       /* jeweilige Ereignisklasse */
  diag_set prchk;
  diag_set timer;
  diag_set opint;
  diag_set error;
  diag_set runout;
  diag_set brkpt;
  diag_set abend;
  diag_set pterm;
  diag_set rtimer;
  diag_set filler;
};

struct type             /* Parameterübergabe-Modus für */
{                       /* jeweilige Ereignisklasse */
  type_set prchk;
```

```
  type_set timer;
  type_set opint;
  type_set error;
  type_set runout;
  type_set brkpt;
  type_set abend;
  type_set pterm;
  type_set rtimer;
  type_set filler;
};
#define stx_set      char
#define old_stx      0
#define new_stx      4
#define del_stx      8

#define diag_set     char
#define ful_diag     0
#define min_diag     4
#define no_diag      8

#define err_set      char
#define no_err       0
#define par_err      4
#define stx_err      8
#define mem_err      12


#define type_set     char
#define par_opt      0
#define par_std      4
```

Steuerung des cstxit-Aufrufs:

Über diese Information wird der Ablauf des `cstxit`-Aufrufs gesteuert. Es wird festgelegt, welche Aktionen für die jeweilige Ereignisklasse durchgeführt werden.

old_stx Für die entsprechende Ereignisklasse ergibt sich keine Änderung. Eine vorher zugeordnete STXIT-Routine bleibt erhalten. Die restlichen Informationen für diese Ereignisklasse werden nicht ausgewertet.

new_stx Für die entsprechende Ereignisklasse wird eine neue STXIT-Routine zugeordnet. In diesem Fall werden die restlichen Informationen für diese Ereignisklasse ausgewertet. Insbesondere muss die Adresse der Routine im entsprechenden Eintrag von `contp` stehen.

del_stx Für die entsprechende Ereignisklasse wird die bisher zugeordnete STXIT-Routine gelöscht. Die restlichen Informationen für diese Ereignisklasse werden nicht ausgewertet.

Diagnosesteuerung:

ful_diag	Die Parameter zur Diagnosesteuerung werden aus Kompatibilitäts-
min_diag	gründen syntaktisch akzeptiert, jedoch wegen der Umstellung auf
no_diag	ILCS nicht mehr ausgewertet. Die angemeldete Routine wird ohne vorhergehende Diagnosemeldung aktiviert.

Parameterübergabe-Modus:

par_opt	Die Parameter werden in den Registern 1-4 übergeben.
par_std	Die Parameter werden in einer Parameterliste übergeben. Für C ist nur dieser Wert zulässig!

Returncode:

no_err	Die STXIT-Routine wurde ordnungsgemäß definiert.
par_err	Die Parameterstruktur stxitpar wurde falsch versorgt.
stx_err	Fehler beim Anmelden der STXIT-Routine.
mem_err	Fehler bei der Speicherplatzanforderung (beim Anmelden der STXIT-Routine).

Hinweise Die Parameterstruktur `stxitpar` müssen Sie selbst versorgen.

Für die standardmäßige Initialisierung steht ein in der Include-Datei <stxit.h> definierter Prototyp (`stxit_pr`) zur Verfügung. Diesen Prototyp können Sie auf eine selbst definierte Struktur vom Typ `stxitp` kopieren und brauchen dann nur die Felder für diejenigen Ereignisklassen zu versorgen, bei denen die Zuordnung einer STXIT-Routine geändert werden soll.

Bei der Ereignisklasse INTR ist die Adresse zu versorgen (`stxitpar.bufadr`), bei der die Mitteilung an das Programm bereitgestellt werden soll. Die STXIT-Contingency-Routine kann dann die Mitteilung von dieser Adresse abholen und auswerten.

Der Rückgabewert *retcode* ist derzeit nicht abfragbar. Bitte beachten Sie eventuelle Hinweise in der Freigabemitteilung zu `cstxit`.

Siehe auch alarm, cenaco, raise, signal, sleep

ctime, ctime64 - Datum mit Uhrzeit (MEZ) in Englisch

Definition

```
#include <time.h>

char *ctime(const time_t *sek_zg);
char *ctime64(const time64_t *sek_zg);
```

`ctime` und `ctime64` interpretieren die Zeitangabe, auf die *sek_zg* zeigt, (siehe Returnwerte von `mktime`, `mktime64` und `time`, `time64`) als Anzahl der seit dem Stichtag (Epoche) vergangenen Sekunden. Die Funktionen berechnen daraus die Ortszeit (MEZ) und wandeln das Ergebnis in eine Zeichenkette um. `ctime` und `ctime64` verhalten sich analog zu `localtime` und `localtime64`.

Negative Werte werden als Sekunden vor dem Stichtag interpretiert. Das kleinste darstellbare Datum ist der 01.01.1900 00:00:00 lokale Zeit.

Bei `ctime` hängt der Stichtag von der Verwendung des TIMESHIFT-Bindeschalters ab (siehe Abschnitt „Zeitfunktionen“ auf Seite 41):
– ohne TIMESHIFT-Bindeschalter (Standard): 1.1.1950 00:00:00.
– mit TIMESHIFT-Bindeschalter: 1.1.1970 00:00:00.
Bei `ctime64` ist der Stichtag immer der 1.1.1970 00:00:00.

Das größte darstellbare Datum ist bei `ctime` der 19.01.2018 03:14:07 (ohne TIMESHIFT-Bindeschalter) bzw. der 19.01.2038 03:14:07 (mit TIMESHIFT-Bindeschalter).

`ctime64` kann unabhängig von der Verwendung des TIMESHIFT-Bindeschalters Daten bis zum 18.3.4317 02:44:48 darstellen.

Returnwert

Zeiger auf die erzeugte Zeichenkette,
Die Ergebniszeichenkette hat die Länge 26 (einschließlich des abschließenden Null-bytes \0) und das Format einer Datumsangabe mit Uhrzeit in Englisch:
Wochentag Monat Tag Std:Min:Sek Jahr,
z.B. `Fri Apr 29 12:01:20 2011\n\0`

NULL — im Fehlerfall

Hinweise

Die Funktionen `asctime`, `ctime`, `ctime64`, `gmtime`, `gmtime64`, `localtime` und `localtime64` schreiben ihre Ergebnisse in denselben C-internen Datenbereich, so dass der Aufruf einer dieser Funktionen das vorherige Ergebnis einer der anderen Funktionen überschreibt.

Zeitangaben sind auf den 24-Stunden-Tag bezogen.

Beispiel Folgendes Programm wandelt einen Wert in Ortszeit um und gibt das Ergebnis in Form einer englischen Datumsangabe mit Uhrzeit aus.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void)
{  time_t sek;
   sek = time((time_t *)0);
   printf("%s",ctime(&sek));
   return 0;   }
```

Siehe auch asctime, gmtime, gmtime64, localtime, localtime64, mktime, mktime64, time, time64

_ _DATE_ _ - Ausgabe des Übersetzungsdatums (Makro)

Definition _ _DATE_ _

Dieses Makro generiert das Übersetzungsdatum einer Quelldatei als Zeichenkette in der Form:

"dd Mmm yyyy\0" .

Dabei bedeutet

dd Tag (bei Tagen < 10 ohne führende Null)

Mmm Monatsname in Englisch (Abkürzung wie bei `asctime`)

yyyy Jahr

Hinweise Das Makro muss in keiner Include-Datei definiert werden. Sein Name wird vom Compiler erkannt und ersetzt.

Beispiel

```
#include <stdio.h>

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

{
printf("Uebersetzung des Programms %s am %s um %s Uhr\n", argv[0], __DATE__,
__TIME__);
return 0;
}
```

Siehe auch asctime, _ _TIME_ _

difftime, difftime64 - Zeitdifferenz berechnen

Definition #include <time.h>

double difftime(time_t zeit2, time_t zeit1);
double difftime64(time64_t zeit2, time64_t zeit1);

`difftime` und `difftime64` berechnen die Differenz zwischen den Zeitwerten *zeit2* und *zeit1*. Die Zeitwerte müssen vom Typ `time_t` bzw. `time64_t` sein. Zeitwerte dieser Typen werden zum Beispiel von den Funktionen `mktime` und `time` bzw. `mktime64` und `time64` geliefert.

Returnwert Zeitdifferenz in Sekunden als Gleitkommazahl.

Siehe auch time, time64, mktime, mktime64, ftime, ftime64, localtime, localtime64, asctime, gmtime, gmtime64

div - Division mit ganzen Zahlen (int)

Definition #include <stdlib.h>

div_t div(int dividend, int divisor);

`div` berechnet den Quotienten und den Rest der Division *dividend* durch *divisor*.
Das Vorzeichen des Quotienten ist gleich dem Vorzeichen des algebraischen Quotienten. Die Größe des Quotienten ist die größte ganze Zahl kleiner oder gleich dem absoluten Wert des algebraischen Quotienten.

Der Rest ergibt sich aus der Gleichung Quotient * Divisor + Rest = Dividend

Returnwert Struktur vom Typ `div_t`,
die sowohl den Quotienten *quot* als auch den Rest *rem* als `integer`-Werte enthält.

Beispiel

```
div_t d;

d = div( 7, 3);        /*  d.quot =  2,  d.rem =  1  */
d = div(-7, 3);        /*  d.quot = -2,  d.rem = -1  */
d = div( 7,-3);        /*  d.quot = -2,  d.rem =  1  */
d = div(-7,-3);        /*   d.quot =  2,    d.rem  = -1  */
```

Siehe auch ldiv, lldiv

double2ieee - Gleitpunktzahl vom /390-Format in das IEEE-Format konvertieren

Definition #include <ieee_390.h>

extern double double2ieee (double num);

`double2ieee` konvertiert eine 8-byte-Gleitpunktzahl *num* des /390-Formats in das IEEE-Format und liefert sie als Ergebnis zurück. Dabei kann weder Overflow noch Underflow auftreten, es können aber bis zu drei Bit-Stellen verloren gehen.

Parameter double num
8-byte-Gleitpunktzahl im /390-Format

Returnwert 8-byte-Gleitpunktzahl im IEEE-Format (bei Erfolg)

Die globale Variable *float_exceptions_flag* enthält Informationen für den Fall einer nicht ordnungsgemäßen Konvertierung und ist wie wie folgt definiert:

```
extern int float_exception_flags;
enum {
   float_flag_inexact   =  1,
   float_flag_divbyzero =  2,
   float_flag_underflow =  4,
   float_flag_overflow  =  8,
   float_flag_invalid   = 16
};
```

Falls bei der Konvertierung Bit-Stellen verloren gehen und das Ergebnis dadurch ungenau wird, wird *float_flag_inexact* gesetzt.

Siehe auch ieee2double, float2ieee, ieee2float

ecvt - Gleitkommazahl in Zeichenkette umwandeln

Definition

```
#include <stdlib.h>

char *ecvt(double wert, int anz, int *dez_pkt, int *vorzeichen);
```

`ecvt` wandelt einen Gleitkommawert *wert* in eine Zeichenkette aus *anz* Ziffern um und liefert als Ergebnis einen Zeiger auf diese Zeichenkette.

Die Zeichenkette beginnt mit der ersten Ziffer ungleich 0 aus dem umzuwandelnden Gleitkommawert, d.h. führende Nullen werden nicht übernommen.

Dezimalpunkt und ein ggf. negatives Vorzeichen sind nicht Bestandteil der Zeichenkette. `ecvt` liefert jedoch die Position des Dezimalpunktes und das Vorzeichen in Ergebnisparametern zurück.

Parameter

double wert
: Gleitkommawert, der für die Ausgabe aufbereitet werden soll.

int anz
: Anzahl der Ziffern in der Ergebniszeichenkette (gerechnet ab der ersten Ziffer ungleich 0 aus dem umzuwandelnden Gleitkommawert).

  Ist *anz* kleiner als die Ziffernzahl von *wert*, wird die niedrigste Stelle gerundet.
  Ist *anz* größer, wird rechtsbündig mit Nullen aufgefüllt.

int *dez_pkt
: Zeiger auf eine ganze Zahl, die die Position des Dezimalpunktes in der Ergebniszeichenkette angibt.

  positive Zahl: Position relativ zum Beginn der Ergebniszeichenkette.
  negative Zahl bzw. 0: Dezimalpunkt steht links vor der ersten Ziffer.

int *vorzeichen
: Zeiger auf eine ganze Zahl, die das Vorzeichen der Ergebniszeichenkette angibt.

  0: das Vorzeichen ist positiv
  ungleich 0: das Vorzeichen ist negativ

Returnwert Zeiger auf die umgewandelte Zeichenkette
bei Erfolg. `ecvt` schließt die Zeichenkette mit dem Nullbyte (\0) ab.

Hinweise Falsche Parameter, etwa ein `integer`- statt `double`-Wert, führen zum Programmabbruch!

Beachten Sie, dass die Argumente *dez_pkt* und *vorzeichen* Zeiger sein müssen!

`ecvt` schreibt sein Ergebnis in einen C-internen Datenbereich, der bei jedem Aufruf überschrieben wird! Denselben Datenbereich benutzt auch die Funktion `fcvt`.

Beispiel Folgendes Programm liest einen Gleitkommawert *x* ein, wandelt ihn nach der Angabe in *n* um und gibt ihn als Zeichenkette wieder aus. Zusätzlich werden das berechnete Vorzeichen *sign* und die Position des Dezimalpunktes *dec_p* ausgegeben.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  double x;
  int n, dec_p, sign;
  char *s;
  printf("Bitte Gleitkommazahl eingeben:  \n");
  if (scanf("%lf", &x) == 1)
  {
    printf("Wieviel signifikante Stellen :  \n");
    if (scanf("%d", &n) == 1)
    {

      s = ecvt(x, n, &dec_p, &sign);
      printf("Die Zeichenkette lautet: %s\n", s);

      printf("Das Vorzeichen ist %s \n",
             (sign == 0 ? "positiv" : "negativ"));

      printf("Position des Dezimalpunktes ist %d \n", dec_p);
    }
  }
  return 0;
}
```

Siehe auch fcvt, gcvt

_edt - EDT-Aufruf

Definition

#include <stdlib.h>

void _edt(void);

`_edt` ruft den BS2000-Dateibearbeiter EDT auf. Nach ordnungsgemäßer Beendigung des Dateibearbeiters fährt das Programm mit der nächsten C-Anweisung nach Aufruf der Funktion `_edt` fort.

Hinweis

Programme, die die Funktion `_edt` aufrufen, benötigen beim Ablauf Module aus der Modulbibliothek EDTLIB (standardmäßig auf der $TSOS-Kennung). Beim Binden ist eine RESOLVE-Anweisung auf diese Bibliothek abzusetzen.

Beispiel

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   _edt();
   printf("Ruecksprung ins C-Programm\n");
   return 0;
}
```

environ - externe Variable für die Umgebung

Definition

extern char **environ;

`environ` ist eine externe Variable, die auf einen Zeichenkettenvektor mit Umgebungsvariablen zeigt. Dieser wird auch kurz Umgebung genannt. Eine Zeichenkette dieses Vektors hat die Form "*name*=*value*", wobei *name* die Umgebungsvariable und *value* deren aktuellen Wert bezeichnet. Durch Umgebungsvariablen können einer Anwendung Informationen über die Programmumgebung zur Verfügung gestellt werden (siehe auch Abschnitt „Umgebungsvariablen“ auf Seite 118).

Hinweis

Auf den `environ`-Vektor sollte nicht direkt von der Anwendung zugegriffen werden.

Siehe auch

getenv, putenv.

erf - Error-Funktion (mathematisch)

Definition #include <math.h>

double erf(double x);

`erf` berechnet für Gleitkommazahlen x die Error-Funktion, die wie folgt definiert ist:

$$\frac{2}{\sqrt{\Pi}} \int_0^x e^{-t^2} \, dt$$

`Returnwerterf(x)`

Siehe auch erfc

erfc - Komplementäre Error-Funktion (mathematisch)

Definition #include <math.h>

double erfc(double x);

`erfc` berechnet für Gleitkommazahlen x die komplementäre Error-Funktion:

`1.0 - erf(x).`

Returnwert `1.0 - erf(x)`

Hinweis Die Funktion `erfc` wird zur Verfügung gestellt, da die Berechnung der Error-Funktion mit der Funktion `erf` bei großen Werten x zu großen Ungenauigkeiten führt.

Siehe auch erf

exit, _exit - Programmbeendigung

Definition

```
#include <stdlib.h>

void exit(int status);

void _exit(int status);
```

`exit` beendet das Programm.

Zunächst werden die mit der Funktion `atexit` registrierten Beendigungsroutinen in der umgekehrten Reihenfolge ihrer Registrierung aufgerufen. Wurde eine Routine mehrmals registriert, wird sie auch mehrmals aufgerufen.
Anschließend werden alle vom Programm geöffneten Dateien geschlossen und folgende Meldungen auf `stderr` ausgegeben:

- „CCM0998 used CPU-time t seconds“, falls in der RUNTIME-Option CPU-TIME=YES gesetzt ist,
- „CCM0999 exit status“, falls *status* ≠ EXIT_SUCCESS (Wert 0) ist,
- „CCM0999 exit FAILURE“, falls *status* = EXIT_FAILURE (Wert 9990888) ist.

`_exit` beendet ebenfalls das Programm.

Im Unterschied zu `exit` werden die mit `atexit` registrierten Beendigungsroutinen nicht aufgerufen und geöffnete Dateien nicht geschlossen. Es wird lediglich die Meldung „CCM0999 exit status“ ausgegeben (falls *status* ≠ EXIT_SUCCESS ist).

Je nach Wert des Parameters *status* wird die Zustandsanzeige der Monitor-Jobvariablen MONJV (1. - 3. Byte) auf den Wert "$T " oder "$A " gesetzt.

Parameter

int status

Dieser Parameter kann folgende Werte enthalten:

- die in der Include-Datei <stdlib.h> definierten symbolischen Konstanten EXIT_SUCCESS und EXIT_FAILURE oder
- einen beliebigen `integer`-Wert.

EXIT_SUCCESS (Wert 0)
verursacht eine normale Programmbeendigung. Die Zustandsanzeige der MONJV bekommt den Wert „$T „ zugewiesen.

EXIT_FAILURE (Wert 9990888)
verursacht eine sog. Jobstep-Beendigung, d.h.

– das Programm wird beendet,
– in einer DO- oder CALL-Prozedur verzweigt das System zum nächsten Kommando ABEND, END-PROCEDURE, SET-JOB-STEP oder LOGOFF,
– es erfolgt die Systemmeldung „ABNORMAL PROGRAM TERMINATION“.

Die Zustandsanzeige der MONJV bekommt den Wert "$A " zugewiesen.

`integer`-Wert
ist dieser Wert ungleich den vordefinierten Werten EXIT_SUCCESS und EXIT_FAILURE (≠ 0 bzw. ≠9990888), wird eine Jobstep-Beendigung durchgeführt und die Zustandsanzeige der MONJV bekommt den Wert "$T " zugewiesen.

Enspricht dieser Wert den vordefinierten Werten EXIT_SUCCESS oder EXIT_FAILURE, werden die oben genannten Aktionen durchgeführt.

Hinweise Um Monitor-Jobvariablen versorgen und abfragen zu können, müssen Sie das C-Programm mit folgendem Kommando starten:

```
/START-PROG programm,MONJV=monjvname
```

Der Inhalt der Jobvariablen lässt sich dann z.B. mit folgendem Kommando abfragen:

```
/SHOW-JV JV-NAME(monjvname)
```

Weitere Informationen zur Ablaufüberwachung mit Monitor-Jobvariablen finden Sie im Handbuch „Jobvariablen“.

Siehe auch abort, atexit, bs2exit, signal

exp - Exponentialfunktion

Definition #include <math.h>

double exp(double x);

`exp` berechnet die Exponentialfunktion für zulässige Gleitkommazahlen x.

Returnwert

e^x	bei Erfolg.
HUGE_VAL	bei Überlauf. Zusätzlich wird `errno` auf ERANGE gesetzt (Resultat zu groß).

Beispiel Folgendes Programm berechnet e^x für einen eingelesenen Wert x.

```
#include <math.h>
#include <stdio.h>

int main(void)
{
  double x;
  printf("Bitte geben Sie eine Gleitkommazahl ein:\n");
  if (scanf("%lf", &x) == 1)
      printf("exp(%g) = %g\n", x, exp(x));
  return 0;
}
```

Siehe auch log, log10, pow

fabs - Absolutbetrag einer Gleitkommazahl

Definition #include <math.h>

double fabs(double x);

`fabs` berechnet den Absolutbetrag einer Gleitkommazahl x.

Returnwert Absolutbetrag des Arguments: `|x|`

Beispiel Folgendes Programm berechnet den Absolutbetrag einer eingelesenen Gleitkommazahl.

```
#include <math.h>
#include <stdio.h>

int main(void)
{
   double x;
   printf("Bitte geben Sie eine Gleitkommazahl ein:\n");
   if (scanf("%lf", &x) == 1)
       printf("?%g? = %g\n", x, fabs(x));
   return 0;
}
```

Siehe auch abs, cabs, ceil, floor, labs, llabs

fclose - Datei schließen und Puffer bereinigen

Definition

```
#include <stdio.h>

int fclose(FILE *dz);
```

`fclose` schließt die Datei, auf deren FILE-Struktur der Dateizeiger *dz* zeigt, und gibt *dz* frei. Speicherplatz, der für diese FILE-Struktur dynamisch (bei `fopen` bzw. `fopen64`) angelegt wurde, wird ebenfalls freigegeben. `fclose` ruft die Funktion `fflush` auf, bevor die Datei geschlossen wird.

Returnwert

0	Die Datei wurde geschlossen.
EOF	`fclose` war nicht erfolgreich, weil – *dz* keiner Datei zugeordnet ist (Datei bereits geschlossen) oder – beim Leeren des Puffers ein Fehler auftrat.

Hinweise

Wenn der Dateizeiger *dz* nicht auf eine FILE-Struktur zeigt, bricht das Programm ab!

Immer wenn ein Programm normal oder mit `exit` beendet wird, wird für jede offene Datei automatisch ein `fclose` ausgeführt. Sie brauchen `fclose` also nur dann explizit aufzurufen, wenn Sie vor Programmbeendigung eine Datei schließen wollen, z.B. um das Limit für geöffnete Dateien (=2048) nicht zu überschreiten.

Satz-E/A

Da bei Satz-E/A keine Daten gepuffert werden, entfällt der interne Aufruf der Funktion `fflush`.

Beispiel

Folgender Programmausschnitt schließt die Datei mit Dateizeiger *fp* bei Erreichen des Dateiendes.

```
FILE *fp;

if(feof(fp))
  fclose(fp);
```

Siehe auch

fflush, close, fdopen, fopen, fopen64, exit

fcvt - Gleitkommazahl in Zeichenkette umwandeln

Definition

```
#include <stdlib.h>

char *fcvt(double wert, int anz, int *dez_pkt, int *vorzeichen);
```

`fcvt` wandelt einen Gleitkommawert *wert* in eine Zeichenkette aus Ziffern um und liefert als Ergebnis einen Zeiger auf diese Zeichenkette. Das Ausgabeformat entspricht dem FORTRAN F-Format.

Die Zeichenkette beginnt mit der ersten Ziffer ungleich 0 aus dem umzuwandelnden Gleitkommawert und enthält *anz* Stellen nach dem Dezimalpunkt.

Dezimalpunkt und ein ggf. negatives Vorzeichen sind nicht Bestandteil der Zeichenkette. `fcvt` liefert jedoch die Position des Dezimalpunktes und das Vorzeichen in Ergebnisparametern zurück.

Parameter

double wert
: Gleitkommawert, der für die Ausgabe aufbereitet werden soll.

int anz
: Anzahl der Ziffern nach dem Dezimalpunkt.
Ist *anz* kleiner als die Ziffernzahl von *wert* nach dem Dezimalpunkt, wird die niedrigste Stelle (wie bei FORTRAN F-Format) gerundet.
Ist *anz* größer, wird rechtsbündig mit Nullen aufgefüllt.

int *dez_pkt
: Zeiger auf eine ganze Zahl, die die Position des Dezimalpunktes in der Ergebniszeichenkette angibt.
positive Zahl: Position relativ zum Beginn der Ergebniszeichenkette
negative Zahl bzw. 0: Dezimalpunkt steht links vor der ersten Ziffer.

int *vorzeichen
: Zeiger auf eine ganze Zahl, die das Vorzeichen der Ergebniszeichenkette angibt.
0: das Vorzeichen ist positiv
ungleich 0: das Vorzeichen ist negativ

Returnwert Zeiger auf die umgewandelte Zeichenkette
bei Erfolg. `fcvt` schließt die Zeichenkette mit dem Nullbyte (\0) ab.

Hinweise Falsche Parameter, etwa ein `integer`- statt `double`-Wert, führen zum Programmabbruch!

Beachten Sie, dass die Argumente *dez_pkt* und *vorzeichen* Zeiger sein müssen!

`fcvt` schreibt sein Ergebnis in einen C-internen Datenbereich, der bei jedem Aufruf überschrieben wird! Denselben Datenbereich benutzt auch die Funktion `ecvt`.

Beispiel Folgendes Programm liest einen Gleitkommawert *x* ein, wandelt ihn nach der Angabe in *n* gemäß dem FORTRAN F-Format um und gibt ihn als Zeichenkette wieder aus. Zusätzlich werden das berechnete Vorzeichen *sign* und die Position des Dezimalpunktes *dec_p* ausgegeben.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  double x;
  int n, dec_p, sign;
  printf("Bitte Gleitkommazahl eingeben:  \n");
  if (scanf("%lf", &x) == 1)
  {
    printf("Wieviel signifikante Stellen :  \n");
    if (scanf("%d", &n) == 1)
    {
      printf("Die Zahl lautet umgewandelt :  %s \n",
             fcvt(x, n, &dec_p, &sign));
      printf("Das Vorzeichen ist %s \n",
             (sign == 0 ? "positiv" : "negativ"));
      printf("Position des Dezimalpunktes: %d \n", dec_p);
    }
  }
  return 0;
}
```

Siehe auch ecvt, gcvt

fdelrec - Satz in ISAM-Datei löschen (Satz-E/A)

Definition

```
#include <stdio.h>

int fdelrec(FILE *dz, void *key);
```

`fdelrec` löscht aus einer ISAM-Datei mit Satz-E/A den Satz mit dem Schlüsselwert *key*.

Parameter

FILE *dz
: Dateizeiger einer ISAM-Datei, die im Modus "type=record,forg=key" eröffnet wurde (vgl. `fopen`/`fopen64`, `freopen`/`freopen64`).

void *key
: Zeiger auf einen Bereich, der den Schlüsselwert des zu löschenden Satzes in vollständiger Länge enthält oder NULL.
Ist *key* gleich NULL, wird der zuletzt gelesene Satz gelöscht. Der Satz muss unmittelbar vor dem `fdelrec`-Aufruf gelesen worden sein.

Returnwert

0	wenn der Satz mit dem angegebenen Schlüssel gelöscht wurde.
> 0	wenn der zu löschende Satz nicht existiert.
EOF	wenn ein Fehler aufgetreten ist.

Hinweise

Wenn der Aufruf fehlerfrei war (Returnwerte 0 bzw. > 0), wird das EOF-Flag der Datei zurückgesetzt.

Ist der angegebene Schlüsselwert nicht in der Datei vorhanden (Returnwert > 0), bleibt die aktuelle Position des Lese-/Schreibzeigers unverändert. Einzige Ausnahme: Wenn die Datei zum Zeitpunkt des `fdelrec`-Aufrufs auf den zweiten oder höheren Schlüssel einer Gruppe von Sätzen mit gleichen Schlüsseln positioniert ist, positioniert `fdelrec` die Datei auf den ersten Satz nach dieser Gruppe.

In ISAM-Dateien mit Schlüsselverdoppelung löscht `fdelrec` den ersten Satz mit dem angegebenen Schlüssel. Anschließend ist die Datei auf den nächsten Satz (mit gleichem bzw. nächst höherem) Schlüssel positioniert.

Siehe auch

flocate, fopen, fopen64, freopen, freopen64

fdopen - Dateizeiger einer Dateikennzahl zuweisen

Definition

```
#include <stdio.h>

FILE *fdopen(int dk, const char *art);
```

`fdopen` weist der bereits mit `open`/`open64` oder `creat`/`creat64` geöffneten Datei (mit Dateikennzahl *dk*) einen Dateizeiger zu.
Nach einem `fdopen`-Aufruf kann die Datei auch mit den Funktionen aus der Standard-Ein-/Ausgabebibliothek bearbeitet werden (`fread`, `fputc`, `fprintf` etc.).

Parameter

int dk
: Dateikennzahl, die durch einen `creat`/`creat64`- oder `open`/`open64`-Aufruf zugewiesen wurde.

const char *art
: Zeichenkette, die die Zugriffsart angibt (siehe bei `fopen`/`fopen64`). Der Parameter wird nicht ausgewertet. Die Datei behält die ursprüngliche Zugriffsart, die bei `open`/`open64` bzw. `creat`/`creat64` angegeben wurde, d.h. eine Änderung der Zugriffsart mit `fdopen` ist nicht möglich. Auch die optionalen Zusatzangaben im Parameter *art* werden nicht ausgewertet.

Returnwert

Dateizeiger auf die zugewiesene FILE-Struktur
bei Erfolg.

Hinweis

Treten Fehler auf, z.B. eine ungültige Dateikennzahl, liefert `fdopen` weder ein definiertes Ergebnis noch eine Fehlermeldung. Das Programm bricht auch nicht ab!

Beispiel Folgendes Program öffnet die Datei *dat* für elementare und für Standard-Ein-/Ausgabeoperationen.

```
#include <stdio.h>
#include <stdlib.h>

FILE *fp;
int fd;
char buf[10];
int c;

int main(void)
{
   int n;

               /* zuerst mit Dateikennzahl arbeiten */
   if((fd = open("dat",2)) < 0)
     {
        perror("open");
        exit(1);
     }

   if((n = read(fd,buf,10)) > 0)
      write(1,buf,n);

               /* Dateizeiger mit Dateikennzahl verbinden */
   fp = fdopen(fd,"w");
   while((c = getchar()) != EOF)
        putc(c,fp);
   fclose(fp);
   return 0;
 }
```

Siehe auch creat, creat64, fclose, fseek, fseek64, fopen, fopen64, freopen, freopen64, open, open64

feof - Test auf Dateiende

Definition

```
#include <stdio.h>
int feof(FILE *dz);
```

`feof` erkennt das Ende der Datei mit Dateizeiger *dz*.

Returnwert

≠ 0	Dateiende wurde erreicht.
0	sonst.

Hinweise

`feof` ist sowohl als Makro als auch als Funktion realisiert (siehe Abschnitt „Funktionen und Makros“ auf Seite 19).

`feof` wird üblicherweise nach Zugriffsfunktionen angewendet, die keine Dateiende-Meldung bringen (`fread`).

Wenn die Datei nach Erreichen des Dateiendes zurückpositioniert wird (z.B. mit `fseek`/`fseek64`, `fsetpos`/`fsetpos64`, `rewind`) oder wenn die Funktion `clearerr` aufgerufen wird, liefert `feof` den Wert 0.

Satz-E/A

`feof` ist auch auf Dateien mit Satz-E/A unverändert anwendbar.

Siehe auch

clearerr, ferror, fopen, fopen64, fseek, fseek64, fsetpos, fsetpos64

ferror - Test auf Dateifehler

Definition

```
#include <stdio.h>
int ferror(FILE *dz);
```

`ferror` überprüft, ob in der FILE-Struktur, auf die *dz* zeigt, das Fehlerflag gesetzt ist.

Returnwert

≠ 0	ein Fehlerflag ist gesetzt.
0	kein Fehlerflag ist gesetzt.

Hinweise

`ferror` ist sowohl als Makro als auch als Funktion realisiert (siehe Abschnitt „Funktionen und Makros“ auf Seite 19).

Das Fehlerflag bleibt bestehen, bis der zugehörige Dateizeiger freigegeben wird (z.B. durch `fclose` oder Programmbeendigung) oder bis die Funktion `clearerr` aufgerufen wird.

`ferror` sollten Sie immer verwenden, wenn Sie aus einer Datei lesen oder in eine Datei schreiben.

Satz-E/A

`ferror` ist auch auf Dateien mit Satz-E/A unverändert anwendbar.

Beispiel

Folgender Programmausschnitt überprüft vor jedem `fread`-Aufruf, ob ein Fehler für die FILE-Struktur angezeigt ist, auf die *fp* zeigt.

```
FILE *fp;
char buf[10];
char x[5];

while( !ferror(fp))
     fread(buf,sizeof(x),10,fp);
```

Siehe auch

clearerr, feof, fopen, fopen64

fflush - Dateipuffer bereinigen

Definition

```
#include <stdio.h>

int fflush(FILE *dz);
```

`fflush` leert den Puffer der Datei mit Dateizeiger *dz* und schreibt alle Daten, die in dem Puffer zwischengespeichert waren, in diese Datei. Ist *dz* ein NULL-Zeiger, führt `fflush` diese Tätigkeiten für alle geöffneten Dateien durch.

Returnwert

0	`fflush` hat den Puffer geleert, oder es brauchte kein Puffer geleert zu werden, weil – der Puffer noch nicht existiert (für die Datei ist noch keine Schreibfunktion ausgeführt) oder – die Datei eine Eingabe- oder INCORE-Datei ist.
EOF	`fflush` hat den Puffer nicht geleert, weil – der Zeiger *dz* keiner Datei zugeordnet ist (z.B. weil die Datei bereits geschlossen ist) oder – die zwischengepufferten Daten nicht übertragen werden konnten.

Hinweise

Bei allen Standard-Ausgabefunktionen, die Daten in eine Datei schreiben (`printf`, `putc`, `fwrite` etc.), werden die Daten in einem C-internen Puffer zwischengespeichert und erst in die Datei geschrieben, wenn eines der folgenden Ereignisse eintritt (siehe auch Abschnitt „Pufferung“ auf Seite 65). Die Zwischenpufferung entfällt bei Ausgaben in Zeichenketten (`sprintf`) und in INCORE-Dateien:

- Ein Neue-Zeile-Zeichen (\n) wird erkannt (nur bei Textdateien),
- die maximale Satzlänge einer Plattendatei ist erreicht,
- bei Datensichtstationen: nach einer Ausgabe auf die Datensichtstation folgt eine Eingabe von der Datensichtstation,
- die Funktionen `fseek`/`fseek64`, `fsetpos`/`fsetpos64`, `rewind` oder `fflush` werden aufgerufen,
- die Datei wird geschlossen.
- Zusätzlich nur bei ANSI-Funktionalität: Wenn das Lesen aus einer beliebigen Textdatei eine Datenübertragung von der externen Datei in den C-internen Puffer notwendig macht, werden die noch in Puffern zwischengespeicherten Daten aller ISAM-Dateien automatisch in die Dateien hinausgeschrieben.

Auch wenn die Daten im Puffer nicht mit einem Neue-Zeile-Zeichen enden, bewirkt `fflush` in einer Textdatei einen Zeilenwechsel. Nachfolgende Daten werden in eine neue Zeile (bzw. in einen neuen Satz) geschrieben.

Ausnahme bei ANSI-Funktionalität:
Wenn die Daten einer ISAM-Datei im Puffer nicht mit einem Neue-Zeile-Zeichen enden, bewirkt `fflush` keinen Zeilenwechsel (bzw. Satzwechsel). Nachfolgende Daten verlängern den Satz in der Datei. Beim Lesen einer ISAM-Datei werden daher nur Neue-Zeile-Zeichen eingelesen, die vom Programm explizit geschrieben wurden.

`fflush` wird intern automatisch ausgeführt, wenn eine Datei geschlossen wird (`fclose`, `close`) oder wenn ein Programm normal bzw. mit `exit` beendet wird.

`fflush` kann dazu benutzt werden, die Ausgabe von Daten während des Programmablaufs zu steuern, z.B. um diverse Eingaben zu einer einzigen Ausgabe zu verketten und zu einem selbst definierten Zeitpunkt auf einmal auszugeben (siehe auch Beispiel).

Satz-E/A Der Aufruf von `fflush` wird zwar nicht mit Fehler abgewiesen, bleibt jedoch ohne Wirkung. Bei Dateien mit Satz-E/A werden keine Daten zwischengepuffert.

Beispiel Folgendes Programm liest von `stdin` alphabetisch sortierte Namen ein und gibt sie in eine Datei aus. Namen mit demselben Anfangsbuchstaben sollen, jeweils durch ein Leerzeichen getrennt, in denselben Satz der Datei geschrieben werden. Das erwünschte Ergebnis erhält man bei „ANSI“-Funktionalität nur bei der Ausgabe in SAM-Dateien. Bei ISAM-Dateien werden alle Namen in einen Satz geschrieben, da `fflush` keinen Satzwechsel bewirkt.

```
#include <stdio.h>
int main(void)
{
  FILE *fp;
  char name[20];
  char prevname;
  prevname = '%';
  fp = fopen ("link=link", "w");
  while (gets(name))
  {
    if(prevname != name[0])
       fflush(fp);
    else
       fputc(' ', fp);
    fputs(name, fp);
    prevname = name[0];
  };
  fclose(fp);
  return 0;
}
```

Siehe auch exit, close, fclose

fgetc - Zeichen aus einer Datei einlesen

Definition

```
#include <stdio.h>
int fgetc(FILE *dz);
```

`fgetc` liest ein Zeichen aus der Datei mit Dateizeiger *dz* von der aktuellen Lese-/Schreibposition. `fgetc` verhält sich wie getc (als Funktion).

Returnwert

Ganzzahl	gelesenes Zeichen als positiver `integer`-Wert, bei Erfolg.
EOF	bei Fehler oder Dateiende.

Hinweise

Wenn Sie in Ihrem Programm einen Vergleich verwenden, wie etwa
`while((c = fgetc(dz)) != EOF)`
dann muss die Variable *c* immer als `integer`-Größe vereinbart werden. Wenn Sie *c* als `char` definieren, wird die Bedingung EOF aus folgendem Grund nie erfüllt: -1 wird nach `char 'OxFF'` (also +255) konvertiert, EOF ist jedoch als -1 definiert.

Wenn `fgetc` von der Standardeingabe `stdin` liest und EOF das Einlese-Endekriterium ist, erreichen Sie die EOF-Bedingung durch folgende Maßnahmen an der Datensichtstation: Taste K2 drücken, die Systemkommandos EOF und RESUME-PROGRAM eingeben.

Beispiel

Folgendes Programm liest aus maximal 10 beim Aufruf übergebenen Dateien nacheinander jeweils ein Zeichen und gibt es auf Standardausgabe aus.

```
#include <stdio.h>
FILE *fp[10], **app;
int main(int argc, char *argv[])
{
  int c, i;
  for (i = 1; i < argc && i <= 10; i++)
       fp[i-1] = fopen(argv[i], "r");
  app = fp;
  while(*app != NULL)
       {
         c = fgetc(*app++);
         putchar(c);
       }
  putchar('\n');
  return 0;
}
```

Siehe auch

getc, getchar, ungetc, fopen, fopen64

fgetpos, fgetpos64 - Aktuelle Position des Lese-/Schreibzeigers ermitteln

Definition

```
#include <stdio.h>

int fgetpos(FILE *dz, fpos_t *pos);
int fgetpos64 (FILE *dz, fpos64_t *pos);
```

`fgetpos` und `fgetpos64` liefern die aktuelle Position des Lese-/Schreibzeigers für die Datei mit Dateizeiger *dz* in dem Bereich, auf den *pos* zeigt. Die in *pos* gespeicherte Information kann zum Positionieren der Datei mit den Funktionen `fsetpos` bzw. `fsetpos64` verwendet werden, indem ihr **pos* als Argument übergeben wird.

Es besteht kein funktionaler Unterschied zwischen `fgetpos` und `fgetpos64`, außer dass `fgetpos64` den `fpos64_t`-Datentyp verwendet.

Für die Bearbeitung von Dateien > 2 GB verfahren Sie wie folgt:

– Falls das Define `_FILE_OFFSET_BITS 64` (siehe Seite 71) gesetzt ist, rufen Sie `fgetpos` auf. Implizit wird dann `fgetpos64` mit den passenden Parametern verwendet.

– Andernfalls müssen Sie `fgetpos64` aufrufen.

Returnwert

0	bei erfolgreicher Ausführung von `fgetpos` bzw. `fgetpos64`.
≠ 0	im Fehlerfall. Zusätzlich wird `errno` auf EBADF gesetzt.

Hinweise

`fgetpos`/`fgetpos64` lässt sich auf Binärdateien (SAM im Binärmodus, PAM, INCORE) und Textdateien (SAM im Textmodus, ISAM) anwenden.
`fgetpos`/`fgetpos64` ist nicht anwendbar für Systemdateien (SYSDTA, SYSLST, SYSOUT).

Für ISAM-Dateien ist das Funktionspaar `fgetpos`/`fsetpos` bzw. `fgetpos64`/`fsetpos64` wesentlich performanter als das vergleichbare Funktionspaar `ftell`/`fseek` bzw. `ftell64`/`fseek64`.

Satz-E/A

`fgetpos` bzw. `fgetpos64` liefert die Position hinter dem zuletzt gelesenen, geschriebenen oder gelöschten Satz bzw. die Position, die durch ein unmittelbar vorangegangenes Positionieren erreicht wurde.

Bei ISAM-Dateien mit Schlüsselverdoppelung liefert `fgetpos` und `fgetpos64` immer die Position hinter dem letzten Satz einer Gruppe mit gleichen Schlüsseln, wenn einer dieser Sätze zuvor gelesen, geschrieben oder gelöscht wurde.

Siehe auch fsetpos, fsetpos64, fseek, fseek64, ftell, ftell64

fgets - Zeichenkette aus einer Datei einlesen

Definition

```
#include <stdio.h>

char *fgets(char *s, int n, FILE *dz);
```

`fgets` liest aus der Datei mit Dateizeiger *dz* höchstens *n*-1 Zeichen oder bis einschließlich zum nächsten Zeilenende oder bis Dateiende. Die eingelesenen Zeichen trägt `fgets` in den Bereich ein, auf den *s* zeigt, und schließt die Zeichenkette mit dem Nullbyte ab.

Returnwert

Zeiger auf die Ergebniszeichenkette *s*.
bei Erfolg. `fgets` schließt die Zeichenkette mit dem Nullbyte (\0) ab.

NULL-Zeiger wenn `fgets` nichts eingelesen hat, z.B. weil das Dateiende sofort erreicht wird oder ein Lesefehler auftrat.

Hinweise

Den Bereich, in den `fgets` die gelesene Zeichenkette abspeichern soll, müssen Sie explizit bereitstellen!

Im Unterschied zu `gets` trägt `fgets` auch ein gelesenes Neue-Zeile-Zeichen in die Ergebniszeichenkette ein.

Beispiel

Siehe Beispiel bei `fputs`.

Siehe auch

gets, fopen, fopen64, puts, fputs

fgetwc - Langzeichen aus Datei lesen

Definition
```
#include <wchar.h>
#include <stdio.h>

wint_t fgetwc(FILE *dz);
```

`fgetwc` liest das nächste Zeichen aus der Datei mit dem Dateizeiger *dz*, wandelt es in den entsprechenden Langzeichenwert um und bewegt den Lese-/Schreibzeiger für die Datei weiter, falls ein Lese-/Schreibzeiger definiert ist.

Wenn ein Fehler auftritt, ist der Wert des Lese-/Schreibzeigers für die Datei nicht definiert.

Wenn `fgetwc` von der Standardeingabe `stdin` liest und WEOF das Einlese-Endekriterium ist, erreichen Sie die WEOF-Bedingung durch folgende Maßnahmen an der Datensichtstation: Taste K2 drücken, die Systemkommandos EOF und RESUME-PROGRAM eingeben.

Returnwert
Wert des gelesenen Langzeichens als `wint_t`–Wert
bei erfolgreicher Beendigung.

WEOF — wenn das Dateiende erreicht ist. Das Dateiendekennzeichen für die Datei wird gesetzt,
oder
wenn ein Lesefehler auftritt. Die Fehleranzeige für die Datei wird gesetzt.
`errno` wird auf EBADF gesetzt, wenn *dz* kein gültiger Dateizeiger ist.

Hinweise
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Um zwischen einer Fehlerbedingung und einer Dateiendebedingung zu unterscheiden, müssen `ferror` bzw. `feof` verwendet werden.

Siehe auch
feof, ferror, fgetc, fopen, fopen64

fgetws - Langzeichenkette aus einer Datei lesen

Definition

```
#include <wchar.h>
#include <stdio.h>

wchar_t * fgetws(wchar_t *ws, int n, FILE *dz);
```

`fgetws` liest Zeichen aus der Datei mit dem Dateizeiger *dz*, wandelt sie in die entsprechenden Langzeichenwerte um und legt sie im Vektor *ws* vom Typ `wchar_t` ab. `fgetws` liest maximal *n*-1 Zeichen bis einschließlich einem Zeilenendezeichen oder bis zum Dateiende. Die Langzeichenkette *ws* wird mit einem Nullbyte-Langzeichen abgeschlossen.

Wenn ein Fehler auftritt, ist der Wert des Lese-/Schreibzeigers für die Datei nicht definiert.

Returnwert

Zeiger auf die Ergebnis-Langzeichenkette *ws*
bei Erfolg.

NULL-Zeiger wenn das Dateiende erreicht wird. Das Dateiendekennzeichen für die Datei wird gesetzt,
oder
wenn ein Lesefehler auftritt. Das Fehlerkennzeichen für die Datei wird gesetzt. `errno` wird auf EBADF gesetzt, wenn *dz* kein gültiger Dateizeiger ist

Hinweis

In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Siehe auch

fgetwc, fopen, fopen64, fread

_ _FILE_ _ - Ausgabe des Quelldateinamens

Definition _ _FILE_ _

Dieses Makro generiert den Dateinamen des Quellprogramms als Zeichenkette in der Form:

```
"name\0"
```

Hinweis Das Makro muss in keiner Include-Datei definiert werden. Sein Name wird vom Compiler erkannt und ersetzt.

float2ieee - Gleitpunktzahl vom /390-Format in das IEEE-Format konvertieren

Definition

```
#include <ieee_390.h>

extern float float2ieee (float num);
```

`float2ieee` konvertiert eine 4-byte-Gleitpunktzahl des /390-Formats in das IEEE-Format und liefert sie als Ergebnis zurück. Dabei geht keine Genauigkeit verloren.

Parameter

float num
4-byte-Gleitpunktzahl im /390-Format

Returnwert

4-byte-Gleitpunktzahl im IEEE-Format (bei Erfolg)

`+/- Infinity`, falls die /390-Gleitpunktzahl betragsmäßig größer ist als die größte darstellbare IEEE-Gleitpunktzahl

`0.0`, falls die /390-Gleitpunktzahl betragsmäßig kleiner ist als die kleinste darstellbare Zahl des IEEE-Formats

Die globale Variable *float_exceptions_flag* enthält Informationen für den Fall einer nicht ordnungsgemäßen Konvertierung und ist wie folgt definiert:

```
extern int float_exception_flags;
enum {
   float_flag_inexact   =  1,
   float_flag_divbyzero =  2,
   float_flag_underflow =  4,
   float_flag_overflow  =  8,
   float_flag_invalid   = 16
};
```

Falls die /390-Gleitpunktzahl betragsmäßig größer ist als die größte darstellbare IEEE-Gleitpunktzahl, wird *float_flag_overflow* gesetzt.

Falls die /390-Gleitpunktzahl betragsmäßig kleiner ist als die kleinste darstellbare Zahl des IEEE-Formats, wird *float_flag_underflow* gesetzt.

Siehe auch

ieee2float, double2ieee, ieee2double

flocate - ISAM-Datei explizit positionieren (Satz-E/A)

Definition

```
#include <stdio.h>

int flocate(FILE *dz, void *key, size_t keylen, int option);
```

`flocate` dient zum expliziten Positionieren einer ISAM-Datei mit Satz-E/A. `flocate` ändert die aktuelle Position des Lese-/Schreibzeigers der Datei mit Dateizeiger *dz* entsprechend den Angaben:
Schlüsselwert *key*,
Schlüssellänge *keylen* und
Option *option* (_KEY_FIRST, _KEY_LAST, _KEY_EQ, _KEY_GE).

Parameter

FILE *dz
: Dateizeiger einer ISAM-Datei, die im Modus "type=record,forg=key" eröffnet wurde (vgl. `fopen`/`open64`, `freopen`/`freopen64`).

void *key
: Zeiger auf einen Bereich, der den Schlüsselwert enthält.

size_t keylen
: Länge des Schlüsselwertes. Der Wert muss ungleich Null sein.

  Ist *keylen* kleiner als die Schlüssellänge der Datei, füllt `flocate` den Schlüsselwert intern bis auf die Schlüssellänge der Datei mit binären Nullen auf und nimmt diesen generierten Schlüssel als Positioniergrundlage.
  Ist *keylen* größer als die Schlüssellänge der Datei, schneidet `flocate` den Schlüsselwert intern von rechts bis auf die Schlüssellänge der Datei ab und nimmt diesen verkürzten Schlüssel als Positioniergrundlage.

int option
: Dieser Parameter kann folgende in <stdio.h> definierte Werte enthalten:

_KEY_FIRST	Positioniert auf den Dateianfang. Die Parameter *key* und *keylen* werden ignoriert.. Das Positionieren ist auch in leeren Dateien erfolgreich.
_KEY_LAST	Positioniert auf das Dateiende. Die Parameter *key* und *keylen* werden ignoriert. Das Positionieren ist auch in leeren Dateien erfolgreich.
_KEY_EQ	Positioniert auf den ersten Satz mit dem angegebenen Schlüssel *key*.
_KEY_GE	Positioniert auf den ersten Satz mit dem Schlüsselwert größer oder gleich dem angegebenen Schlüssel *key*.

Returnwert	0	wenn der Satz mit dem angegebenen Schlüssel exisitiert.
	> 0	wenn der Satz nicht existiert.
	EOF	wenn ein Fehler aufgetreten ist.

Hinweise Wenn der Aufruf fehlerfrei war (Returnwerte 0 bzw. > 0), wird das EOF-Flag der Datei zurückgesetzt.

Ist der angegebene Schlüsselwert nicht in der Datei vorhanden (Returnwert > 0), bleibt die aktuelle Position des Lese-/Schreibzeigers unverändert. Einzige Ausnahme: Wenn die Datei zum Zeitpunkt des `flocate`-Aufrufs auf den zweiten oder höheren Schlüssel einer Gruppe von Sätzen mit gleichen Schlüsseln positioniert ist, positioniert `flocate` die Datei auf den ersten Satz nach dieser Gruppe.

In ISAM-Dateien mit Schlüsselverdoppelung kann mit `flocate` nicht auf den zweiten oder höheren Satz einer Gruppe mit gleichen Schlüsseln positioniert werden. Dies lässt sich nur durch sequenzielles Lesen bzw. Löschen erreichen.
Mit `flocate` kann nur auf den ersten Satz oder hinter den letzten Satz einer solchen Gruppe positioniert werden.

Siehe auch fdelrec, fgetpos, fgetpos64, fsetpos, fsetpos64, fopen, open64, freopen, freopen64

floor - Abrunden

Definition #include <math.h>

double floor(double x);

`floor` rundet eine Gleitkommazahl x nach unten ganzzahlig ab.

Returnwert Größte ganze Zahl vom Typ `double`, die kleiner oder gleich x ist bei Erfolg.

-HUGE_VAL bei Überlauf. Zusätzlich wird `errno` auf ERANGE gesetzt (Resultat betragsmäßig zu groß).

Beispiel

```
#include <math.h>
#include <stdio.h>

int main(void)
{
  double x;
  printf("Bitte die abzurundende Gleitkommazahl eingeben\n");
  if (scanf("%lf", &x) == 1)
    printf("Die Zahl %g wird abgerundet zu %f\n", x, floor(x));
  return 0;
}
```

Siehe auch ceil

fmod - Rest einer Division

Definition

```
#include <math.h>

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

`fmod` berechnet den Rest der Division x/y.

Der Rest hat das gleiche Vorzeichen wie der Dividend x und sein Absolutbetrag ist immer kleiner als der Divisor y.

Returnwert

Rest der Division x/y	als Gleitkommazahl vom Typ `double`, wenn y ≠ 0.
0	wenn y = 0.

fopen, fopen64 - Datei öffnen

Definition #include <stdio.h>

FILE *fopen(const char *d_name, const char *art);
FILE *fopen64(const char *d_name, const char *art);

`fopen` und `fopen64` öffnen die Datei *d_name* und weisen ihr eine FILE-Struktur und einen Dateizeiger zu. Der Dateizeiger zeigt auf die zugewiesene FILE-Struktur.
Die FILE-Struktur ist in der Datei <stdio.h> definiert. Sie enthält notwendige Information für die meisten Funktionen aus der Standard-Ein-/Ausgabebibliothek.

Es besteht kein funktionaler Unterschied zwischen `fopen` und `fopen64`, außer dass in der mit dem Filedescriptor verknüpften Dateibeschreibung das Kennzeichen für eine große Datei hinterlegt wird, dh. es wird das O_LARGEFILE Bit gesetzt. Es wird ein Dateizeiger zurückgegeben, der dazu verwendet werden kann, die Datei über 2 GB hinaus zu vergrößern.

Für die Bearbeitung von Dateien > 2 GB verfahren Sie wie folgt:

– Falls das Define `_FILE_OFFSET_BITS 64` (siehe Seite 71) gesetzt ist, rufen Sie `fopen` auf. Implizit wird dann `fopen64` mit den passenden Parametern verwendet.

– Andernfalls müssen Sie `fopen64` aufrufen.

Parameter const char *d_name
Zeichenkette, die die zu öffnende Datei angibt. *d_name* kann sein:

– jeder gültige BS2000-Dateiname

– "link=*linkname*"
linkname bezeichnet einen BS2000-Linknamen

– "(SYSDTA)", "(SYSOUT)", "(SYSLST)"
die entsprechende Systemdatei

– "(SYSTERM)"
Terminal-Ein-/Ausgabe

– "(INCORE)"
temporäre Binärdatei, die nur im virtuellen Speicher angelegt wird

const char *art
Zeichenkette, die die gewünschte Zugriffsart angibt. Mit optionalen Zusatzangaben können weitere Funktionen gesteuert werden:

Zusatzangabe	**Funktion**
tabexp=yes/no	Behandlung des Tabulatorzeichens (\t)
lbp=yes/no	Behandlung des Last Byte Pointers (LBP)

Zugriffsarten:

"r"	Öffnen Textdatei zum Lesen. Die Datei muss bereits vorhanden sein.
"w"	Öffnen Textdatei zum Neuschreiben. Ist die Datei vorhanden, wird der alte Inhalt gelöscht. Ist die Datei nicht vorhanden, wird sie neu erstellt.
"a"	Öffnen Textdatei zum Anfügen ans Ende der Datei. Ist die Datei vorhanden, wird auf das Dateiende positioniert, d.h. der alte Inhalt bleibt erhalten und die neuen Daten werden ans Ende der Datei angehängt. Ist die Datei nicht vorhanden, wird sie neu erstellt.
"rb"	Öffnen Binärdatei zum Lesen. Die Datei muss bereits vorhanden sein.
"wb"	Öffnen Binärdatei zum Neuschreiben. Ist die Datei vorhanden, wird der alte Inhalt gelöscht. Ist die Datei nicht vorhanden, wird sie neu erstellt.
"ab"	Öffnen Binärdatei zum Anfügen ans Ende der Datei. Ist die Datei vorhanden, wird auf das Dateiende positioniert, d.h. der alte Inhalt bleibt erhalten und die neuen Daten werden ans Ende der Datei angehängt. Ist die Datei nicht vorhanden, wird sie neu erstellt.
"r+w", "r+"	Öffnen Textdatei zum Lesen und Schreiben. Die Datei muss bereits vorhanden sein. Der alte Inhalt bleibt erhalten.
"w+r", "w+"	Öffnen Textdatei zum Neuschreiben und Lesen. Ist die Datei vorhanden, wird der alte Inhalt gelöscht. Ist die Datei nicht vorhanden, wird sie neu erstellt.
"a+r", "a+"	Öffnen Textdatei zum Anfügen ans Ende der Datei und zum Lesen. Ist die Datei vorhanden, bleibt der alte Inhalt erhalten und die neuen Daten werden ans Ende der Datei angehängt. Eine bereits vorhandene Datei ist nach dem Öffnen je nach KR- oder ANSI-Funktionalität unterschiedlich positioniert: bei KR-Funktionalität (nur bei C/C++ Versionen kleiner V3.0 vorhanden) auf das Dateiende, bei ANSI-Funktionalität auf den Dateianfang. Ist die Datei nicht vorhanden, wird sie neu erstellt.
"r+b", "rb+"	Öffnen Binärdatei zum Lesen und Schreiben. Die Datei muss bereits vorhanden sein. Der alte Inhalt bleibt erhalten.
"w+b", "wb+"	Öffnen Binärdatei zum Neuschreiben und Lesen. Ist die Datei vorhanden, wird der alte Inhalt gelöscht. Ist die Datei nicht vorhanden, wird sie neu erstellt.
"a+b", "ab+"	Öffnen Binärdatei zum Anfügen ans Ende der Datei und zum Lesen. Ist die Datei vorhanden, bleibt der alte Inhalt erhalten und die neuen Daten werden ans Ende der Datei angehängt. Eine bereits vorhandene Datei ist nach dem Öffnen je nach KR- oder ANSI-Funktionalität unterschiedlich positioniert: bei KR-Funktionalität (nur bei C/C++ Versionen kleiner V3.0 vorhanden) auf das Dateiende, bei ANSI-Funktionalität auf den Dateianfang. Ist die Datei nicht vorhanden, wird sie neu erstellt.

Tabulatorzeichen (\t)

Im Parameter *art* kann zusätzlich zur Zugriffsart eine Angabe zur Behandlung des Tabulatorzeichens (\t) gemacht werden. Diese Angabe ist nur für Textdateien mit den Zugriffsmethoden SAM und ISAM relevant.

"...,tabexp=yes"

Das Tabulatorzeichen wird in die entsprechende Anzahl Leerzeichen expandiert. Voreinstellung bei KR-Funktionalität (KR-Funktionalität nur bei C/C++ Versionen kleiner V3.0 vorhanden).

"...,tabexp=no"

Das Tabulatorzeichen wird nicht expandiert.
Voreinstellung bei ANSI-Funktionalität.

Last Byte Pointer (LBP)

Im Parameter *art* kann zusätzlich zur Zugriffsart eine Angabe zur Behandlung des Last Byte Pointer (LBP) gemacht werden. Diese Angabe ist nur für Binärdateien mit Zugriffsart PAM relevant und wirkt sich erst aus, wenn die Datei geschlossen wird.

Beim Öffnen und Lesen einer bestehenden Datei wird der LBP unabhängig vom *lbp*-Schalter immer berücksichtigt:

– Ist der LBP der Datei ungleich 0, wird er ausgewertet. Ein eventuell vorhandener Marker wird ignoriert.

– Ist der LBP = 0, wird nach einem Marker gesucht und die Dateilänge daraus ermittelt. Falls kein Marker gefunden wird, wird das Ende des letzten vollständigen Blocks als Dateiende betrachtet.

"...,lbp=yes"

Beim Schließen einer Datei, die verändert oder neu erstellt wurde, wird kein Marker geschrieben (auch wenn einer vorhanden war) und ein gültiger LBP gesetzt. Auf diese Weise können Dateien mit Marker auf LBP ohne Marker umgestellt werden.

"...,lbp=no"

Beim Schließen einer Datei, die verändert oder neu erstellt wurde, wird der LBP auf Null gesetzt. Für eine neu erstellte Datei wird immer ein Marker geschrieben, für eine veränderte nur dann, wenn vorher bereits ein Marker vorhanden war. War kein Marker vorhanden, wird auch keiner geschrieben und die Datei endet mit dem vollständigen letzten Block.

Wird der Schalter *lbp* nicht angegeben, hängt das Verhalten von der Umgebungsvariablen LAST_BYTE_POINTER ab (siehe auch „Umgebungsvariable LAST_BYTE_POINTER“ auf Seite 90):

`LAST_BYTE_POINTER=YES`

Die Funktion verhält sich so, als ob `lbp=yes` angegeben wäre.

`LAST_BYTE_POINTER=NO`

Die Funktion verhält sich so, als ob `lbp=no` angegeben wäre.

Returnwert Zeiger auf die zugewiesene FILE-Struktur bei Erfolg.

NULL-Zeiger wenn die Datei nicht geöffnet werden konnte, z.B. wegen fehlender Zugriffsberechtigung, falschem Datei- oder Linknamen etc.

Hinweise Der BS2000-Dateiname bzw. -Linkname kann in Klein- und Großbuchstaben geschrieben werden, er wird automatisch in Großbuchstaben umgesetzt.

Durch die Angabe eines "b" an zweiter bzw. dritter Stelle im Parameter *art* wird die Datei als Binärdatei geöffnet. Die Angabe ist nur für SAM-Dateien relevant, da nur SAM-Dateien sowohl im Binär- als auch im Textmodus verarbeitet werden.
Systemdateien und ISAM-Dateien werden immer als Textdateien verarbeitet. Die Angabe des Binärmodus führt bei diesen Dateien zu einem Fehler beim Öffnen.
(INCORE)- und PAM-Dateien werden immer als Binärdateien verarbeitet. Aus Kompatibilitätsgründen funktioniert das Öffnen als Binärdatei auch ohne explizite Angabe des Binärmodus.

Wird eine nicht vorhandene Datei neu angelegt, wird standardmäßig folgende Datei erzeugt:

	Binärdatei	Textdatei
Zugriffsmethode	SAM	SAM (KR-Funktionalität, nur bei C/C++ Versionen kleiner V3.0 vorhanden) ISAM (ANSI-Funktionalität)
Satzformat	F	V

Bei Verwendung eines Linknamens lassen sich mit dem ADD-FILE-LINK-Kommando folgende Dateiattribute ändern: Zugriffsmethode, Satzlänge, Satzformat, Blocklänge und Blockformat.
Siehe auch Abschnitt „Katalogisierte Plattendateien (SAM, ISAM, PAM)“ auf Seite 76.

In allen Fällen, in denen der alte Inhalt einer bereits existierenden Datei gelöscht wird (geöffnet zum Neuschreiben bzw. zum Neuschreiben und Lesen), bleiben die Katalogeigenschaften dieser Datei erhalten.

Wenn eine Datei zum Ändern geöffnet wird, kann das Lesen und Schreiben über denselben Dateizeiger erfolgen. Dennoch sollte auf eine Ausgabe nicht unmittelbar eine Eingabe erfolgen ohne ein vorhergehendes Positionieren (`fseek`/`fseek64`, `fsetpos`/`fsetpos64`, `rewind`) oder einen `fflush`-Aufruf. Dasselbe gilt für eine Ausgabe, die einer Eingabe folgt.

Position des Schreib-/Lesezeigers im Anfügemodus:
Wenn der Schreib-/Lesezeiger in einer Datei, die im Anfügemodus eröffnet wurde, explizit vom Dateiende wegpositioniert wurde (`rewind`, `fsetpos`/`fsetpos64`, `fseek`/`fseek64`), wird er je nach KR- oder ANSI-Funktionalität unterschiedlich behandelt.
KR-Funktionalität (nur bei C/C++ Versionen kleiner V3.0 vorhanden): Der aktuelle Schreib/Lesezeiger wird nur beim Schreiben mit der Elementarfunktion `write` ignoriert und automatisch ans Ende der Datei positioniert.
ANSI-Funktionalität: Der aktuelle Schreib-/Lesezeiger wird bei allen Schreibfunktionen ignoriert und automatisch ans Ende der Datei positioniert.

Der Versuch, eine nicht existierende Datei zum Lesen zu öffnen, endet mit Fehler.

(INCORE)-Dateien können nur zum Neuschreiben ("w"), zum Neuschreiben und Lesen ("w+r") oder zum Lesen ("r") eröffnet werden. Es müssen zuerst Daten geschrieben werden. Um die geschriebenen Daten wieder einlesen zu können, gibt es folgende Möglichkeiten: Wurde die Datei nur zum Neuschreiben eröffnet, kann man sie mit der Funktion `freopen` bzw. `freopen64` zum Lesen öffnen. Wurde die Datei zum Neuschreiben und zum Lesen eröffnet, kann man den Lese-/Schreibzeiger mit der Funktion `rewind` auf Dateianfang positionieren.

Sie können eine Datei gleichzeitig für verschiedene Zugriffsmodi eröffnen, sofern diese Modi im BS2000-Datenverwaltungssystem miteinander verträglich sind.

Wenn ein Programm startet, werden ihm automatisch drei Dateizeiger für Standardeingabe, -ausgabe und -fehlerausgabe zugeordnet und zwar:

stdin	Dateizeiger für Standardeingabe (Datensichtstation)
stdout	Dateizeiger für Standardausgabe (Datensichtstation)
stderr	Dateizeiger für Standardfehlerausgabe (Datensichtstation)

Es können maximal _NFILE Dateien gleichzeitig geöffnet sein. _NFILE ist in <stdio.h> mit 2048 definiert.

Satz-E/A

Für das Eröffnen von Dateien mit Satz-E/A ist der Parameter *art* um zwei Angaben erweitert. Diese Angaben folgen in der Zeichenkette hinter der Zugriffsart (s.o.) jeweils durch ein Komma getrennt.

```
"...,type=record [,forg={seq/key}]"
```

type=record	Die Datei wird für satzweise Ein-/Ausgabe eröffnet. Fehlt diese Angabe, wird die Datei für Strom-E/A eröffnet.
forg=seq	Die Dateiorganisation ist sequenziell. Sequenzielle Dateien können SAM- oder PAM-Dateien sein.
forg=key	Die Dateiorganisation ist indexsequenziell. Indexsequenzielle Dateien sind ISAM-Dateien.

Fehlt die Angabe von `forg`, ist die Dateiorganisation vom FCBTYP der Datei abhängig: Der FCBTYP ist durch den Katalogeintrag einer bereits existierenden Datei festgelegt bzw. durch ein ADD-FILE-LINK-Kommando. Für SAM- und PAM-Dateien wird sequenzielle Dateiorganisation angenommen, für ISAM-Dateien indexsequenzielle Dateiorganisation.

Fehlt die Angabe von `forg` und der FCPTYP ist nicht festgelegt (Datei nicht vorhanden, kein ADD-FILE-LINK-Kommando), wird sequenzielle Dateiorganisation angenommen und eine SAM-Datei erstellt.

Für die Satz-E/A gelten folgende Einschränkungen. Werden diese Einschränkungen nicht eingehalten, wird die Datei nicht eröffnet und ein Fehler-Returnwert geliefert:

a) Die Datei muss im Binärmodus eröffnet werden (Angabe "b" bei der Zugriffsart).

b) "type=record" ist zulässig für SAM-, PAM- oder ISAM-Dateien.

c) "forg=seq" ist zulässig für SAM- oder PAM-Dateien, "forg=key" für ISAM-Dateien.

d) Bei ISAM-Dateien ist der Anfügemodus ’a’ unzulässig. Die Position bestimmt sich aus dem Schlüssel im Satz.

Beispiel

```
/* Programm zum Kopieren von dat1 und dat2 auf dat3 */

#include <stdio.h>
#include <stdlib.h>

FILE *fp_1, *fp_2;
void copy(void);

int main(void)   /* dat1" und dat2 müssen existieren */
{
  if((fp_1 = fopen("dat1","r")) == NULL ?? (fp_2 = fopen("dat3","w")) ==NULL)

      {
                /* Programmabbruch bei Fehler mit Rückgabewert 1 */
        perror("fopen");
        exit(1);
      }

  copy();
                /* Umhängen des Dateizeigers von dat1 auf dat2 */

  if((freopen("dat2","r",fp_1)) == NULL)

                /* Programmabbruch bei Fehler mit Rückgabewert 2 */
     exit(2);

  copy();
  fclose(fp_1);
  fclose(fp_2);
  return 0;
}

void copy(void)
{
  int c;
  while((c = getc(fp_1)) != EOF)
        putc((char)c,fp_2);
}
```

Siehe auch creat, creat64, fdopen, freopen, freopen64, ferror, open, open64, fclose, fseek, fseek64

fprintf - Formatierte Ausgabe in eine Datei

Definition

```
#include <stdio.h>

int fprintf(FILE *dz, const char *format, argumentenliste);
```

`fprintf` bereitet Daten (Zeichen, Zeichenketten, numerische Werte) gemäß den Angaben in der Zeichenkette *format* auf und schreibt sie in die Datei mit Dateizeiger *dz*.

`fprintf` arbeitet wie `printf`, außer dass die aufbereiteten Daten in eine Datei und nicht in die Standardausgabe geschrieben werden.

Parameter

FILE *dz
: Dateizeiger der Ausgabedatei.

const char *format
: Formatzeichenkette wie bei `printf` mit KR- oder ANSI-Funktionalität (Beschreibung siehe dort).

argumentenliste
: Variablen oder Konstanten, deren Werte gemäß den Angaben in den Formatanweisungen für die Ausgabe umgewandelt und formatiert werden sollen.
Wenn die Anzahl der Formatanweisungen nicht mit der Anzahl der Argumente übereinstimmt, gilt Folgendes:
Gibt es mehr Argumente, werden die überzähligen ignoriert.
Gibt es weniger Argumente, führt dies zu undefinierten Ergebnissen.

Returnwert

Anzahl der ausgegebenen Zeichen
: bei Erfolg.

negativer Wert
: bei Fehler.

Hinweise

Bei der Umwandlung von Gleitkommazahlen rundet `fprintf` auf die angegebene Genauigkeit.

`fprintf` nimmt keine Konvertierung von einem Datentyp in einen anderen vor. Soll ein Wert nicht entsprechend seinem Typ ausgegeben werden, muss er explizit konvertiert werden (z.B. mit dem `cast`-Operator).

Die Zeichen werden nicht sofort in die externe Datei geschrieben, sondern in einem C-internen Puffer zwischengespeichert (siehe Abschnitt „Pufferung“ auf Seite 65).

Maximale Anzahl der auszugebenden Zeichen:
Bei KR-Funktionalität (nur bei C/C++ Versionen kleiner V3.0 vorhanden) können pro `fprintf`-Aufruf maximal 1400 Zeichen ausgegeben werden,
bei ANSI-Funktionalität maximal 1400 Zeichen pro Konversionselement (z.B. %s).

Versuche, nicht initialisierte Variablen oder Variablen nicht entsprechend ihrem Datentyp auszugeben, können zu undefinierten Ergebnissen führen.

Das Verhalten ist undefiniert, wenn in einer Formatanweisung dem Prozentzeichen (%) ein nicht definiertes Formatierungs- bzw. Umwandlungszeichen folgt.

Beispiel

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    FILE *fp;
    char c, name[40];
    int i;
    char *string;
    double d;

    printf("Name der Ausgabedatei: \n");
    gets(name);
    if((fp = fopen(name,"w")) == NULL)
    {
       printf("Can't open %s\n", name);
       exit(1);
    }
    c = 'A';
    i = 999;
    string = "Dies ist ein String.";
    d = 123.456;
    fprintf(fp, "%c %d %s %f\n", c, i, string, d);
    fclose(fp);
    puts("Richtige Ausgabe in Datei:A 999 Dies ist ein String. 123.456000");
    return 0;
}
```

Siehe auch printf, sprintf, snprintf, putc, putchar, puts, scanf, fscanf

fputc - Zeichen in eine Datei schreiben

Definition

```
#include <stdio.h>

int fputc(int c, FILE *dz);
```

`fputc` schreibt das Zeichen *c* in die Datei mit Dateizeiger *dz* an die aktuelle Lese-/Schreibposition.

Returnwert Geschriebenes Zeichen *c* als positiver `Integer`-Wert, bei Erfolg.

EOF sonst.

Hinweise Die Zeichen werden nicht sofort in die externe Datei geschrieben, sondern in einem C-internen Puffer zwischengespeichert (siehe Abschnitt „Pufferung“ auf Seite 65).

Bei der Ausgabe in Textdateien werden die Steuerzeichen für Zwischenraum (\n, \t, etc.) je nach Art der Textdatei in ihre entsprechende Wirkung umgesetzt (siehe Abschnitt „Zwischenraum“ auf Seite 68).

Beispiel Folgendes Programm liest Zeichen von SYSDTA ein und gibt sie auf SYSOUT wieder aus.

```
#include <stdio.h>
#include <stdlib.h>

void copy(void);
FILE *fp_in, *fp_out;
int main(void)
{
   fp_in = fopen("(SYSDTA)","r");
   fp_out = fopen ("(SYSOUT)","w");
   copy();
   fclose(fp_in);
   fclose(fp_out);
   return 0;
}
void copy(void)
{
  int c;
  while((c = fgetc(fp_in)) != EOF)
    fputc((char)c, fp_out);
}
```

Siehe auch fopen, fopen64, fputwc, putc, putchar

fputs - Zeichenkette in eine Datei schreiben

Definition

#include <stdio.h>

int fputs(const char *s, FILE *dz);

`fputs` schreibt die mit einem Nullbyte (\0) abgeschlossene Zeichenkette *s* in die Datei mit Dateizeiger *dz*.

Returnwert

0	bei Erfolg
EOF	sonst

Hinweise

Im Gegensatz zu `puts` schließt `fputs` die Ausgabe nicht zusätzlich mit einem Neue-Zeile-Zeichen ab.

Das abschließende Nullbyte von *s* wird nicht mitausgegeben.

Bei der Ausgabe in Textdateien werden die Steuerzeichen für Zwischenraum (\n, \t, etc.) je nach Art der Textdatei in ihre entsprechende Wirkung umgesetzt (siehe Abschnitt „Zwischenraum“ auf Seite 68).

Beispiel

Folgendes Programm liest aus der Datei *dat* Zeichenketten ein und gibt sie an der Datensichtstation (SYSOUT) wieder aus.

```
#include <stdio.h>

int main(void)
{
   FILE *fp_in, *fp_out;
   char s[BUFSIZ];
   int max = 120;

   fp_in = fopen("dat","r");
   fp_out = fopen("(SYSOUT)","w");

   while(fgets(s, max, fp_in) != NULL)
     fputs(s, fp_out);
   return 0;
```

Siehe auch

fopen, fopen64, puts, fgets

fputwc - Langzeichen in Datei schreiben

Definition
```
#include <wchar.h>
#include <stdio.h>

wint_t fputwc(wchar_t wc, FILE *dz);
```

`fputwc` schreibt das Zeichen, das dem Langzeichenwert *wc* entspricht, in die Ausgabedatei mit dem Dateizeiger *dz*.
Das Zeichen wird an die Position geschrieben, die durch den zugehörigen Lese-/Schreibzeiger der Datei angegeben ist (falls dieser definiert wurde). Der Zeiger wird entsprechend weiterbewegt.
Wenn die Datei Positionierungsanforderungen nicht unterstützt oder im Anfügemodus geöffnet wurde, wird das Zeichen an die Datei angehängt.
Wenn während der Schreiboperation ein Fehler auftritt, ist der Einfügemodus der Ausgabedatei in einem undefinierten Zustand.

Returnwert

geschriebenes Zeichen *wc* als `wint_t`-Wert	bei Erfolg.
WEOF	bei Fehler. Das Fehlerkennzeichen für die Datei wird gesetzt. `errno` wird auf EBADF gesetzt, wenn dz kein gültiger Dateizeiger ist.

Hinweise
In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Bei der Ausgabe in Textdateien werden die Steuerzeichen für Zwischenraum (\n, \t, etc.) je nach Art der Textdatei in ihre entsprechende Wirkung umgesetzt (siehe Abschnitt „Zwischenraum“ auf Seite 68).

Siehe auch ferror, fopen, fopen64, setbuf

fputws - Langzeichenkette in Datei schreiben

Definition

```
#include <wchar.h>
#include <stdio.h>

int fputws(const wchar_t *ws, FILE *dz);
```

`fputws` schreibt eine Zeichenkette in die Datei mit dem Dateizeiger *dz*, die der mit einem Null-Langzeichen abgeschlossenen Langzeichenkette entspricht, auf die *ws* zeigt. Das dem abschließenden Null-Langzeichen entsprechende Zeichen wird nicht geschrieben.

Returnwert

nicht negative Zahl	bei erfolgreicher Beendigung.
-1	sonst.

Hinweise

`fputws` schließt die Ausgabe nicht zusätzlich mit einem Neue-Zeile-Zeichen ab.

Bei der Ausgabe in Textdateien werden die Steuerzeichen für Zwischenraum (\n, \t, etc.) je nach Art der Textdatei in ihre entsprechende Wirkung umgesetzt (siehe Abschnitt „Zwischenraum“ auf Seite 68).

In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Siehe auch

fopen, fopen64, fputs, fputwc

fread - Blockweise aus einer Datei einlesen

Definition

```
#include <stdio.h>

size_t fread(void *zg, size_t elgroesse, size_t anz, FILE *dz);
```

`fread` liest *anz* Elemente, die jeweils *elgroesse* Bytes beanspruchen, aus der Datei mit Dateizeiger *dz*. Die gelesenen Datenelemente speichert `fread` in den Bereich, auf dessen Anfang *zg* zeigt. Nach erfolgreichem Lesen steht der Lese-/Schreibzeiger hinter dem letzten gelesenen Byte.

Returnwert Anzahl der tatsächlich gelesenen Elemente.
Diese Anzahl kann kleiner als *anz* sein, wenn ein Fehler auftritt oder Dateiende erreicht wird.

Hinweise Sie müssen dafür sorgen, dass der Bereich, auf den *zg* zeigt, zum Abspeichern der eingelesenen Datenelemente ausreicht.

Um sicherzugehen, dass *elgroesse* die richtige Anzahl Bytes für ein Datenelement angibt, sollten Sie die Funktion `sizeof` für die Größe der Dateneinheit verwenden, auf die *zg* zeigt.

`fread` unterscheidet nicht zwischen Dateiende und Fehler. Sie sollten daher vor bzw. nach jedem `fread` Aufruf mit den Funktionen `feof` und `ferror` überprüfen, ob ein korrekter Lesezugriff möglich ist.

`fread` liest über Zeilenende (\n) hinweg und eignet sich daher dazu, Binärdateien einzulesen.

Satz-E/A `fread` liest einen Satz (bzw. Block) von der aktuellen Dateiposition.

Anzahl der einzulesenden Zeichen: Im Folgenden sei *n* die Gesamtanzahl der einzulesenden Zeichen, d.h.

n = Elementlänge * Elementanzahl

Ist *n* größer als die aktuelle Satzlänge, wird trotzdem nur dieser Satz gelesen.
Ist *n* kleiner als die aktuelle Satzlänge, werden nur die ersten *n* Zeichen des Satzes gelesen. Beim nächsten Lesezugriff werden die Daten des nächsten Satzes gelesen.

`fread` liefert den gleichen Returnwert wie bei Strom-E/A, nämlich die Anzahl der vollständig eingelesenen Elemente. Bei Satz-E/A ist es sinnvoll, ausschließlich die Elementlänge 1 zu verwenden, da in diesem Fall der Returnwert der Länge des gelesenen Satzes entspricht (ohne ein ggf. vorhandenes Satzlängenfeld).

Beispiel Folgendes Programm überträgt zwei Personeneinträge in eine Datei (`fwrite`) und liest diese Einträge wieder ein (`fread`).

```
#include <stdio.h>

int main(void)
{
  FILE *fp;
  size_t result;
  static struct p
  {
    char name[20];
    int a;
  } person[2] =
    {
      {"ANNE", 30},
      {"HANS", 60},
    };

  fp = fopen("link=link", "w+r");

  result = fwrite(person, sizeof(struct p), 2, fp);
  printf("%d Personeneintraege geschrieben\n", result);

  rewind(fp);
  result = fread(person, sizeof(struct p), 2, fp);
  printf("%d Personeneintraege gelesen\n", result);
  printf("Name1: %s, Alter1: %d\n", person[0].name, person[0].a);
  printf("Name2: %s, Alter2: %d\n", person[1].name, person[1].a);
  return 0;
}
```

Siehe auch fwrite, feof, ferror, read, fopen, fopen64, fgetc, fgets, fscanf

free - Speicherplatz freigeben

Definition

#include <stdlib.h>

void free(void *zg);

`free` gibt den Speicherplatz frei, auf den *zg* zeigt. *zg* muss das Ergebnis eines vorangegangenen Aufrufs von `malloc`, `calloc` oder `realloc` sein, andernfalls ist das Ergebnis undefiniert!

`free` ist Teil eines C-spezifischen Speicherverwaltungspaketes mit einer eigenen Freispeicherverwaltung. Der mit `free` freigegebene Speicher wird nicht an das Betriebssystem zurückgegeben, sondern durch die Freispeicherverwaltung erfasst (vgl. Funktion `garbcoll`).

Beispiel

Folgender Programmausschnitt gibt einen zuvor mit `malloc` reservierten Speicherbereich wieder frei.

```
#include <stdlib.h>

char *buf;
buf = (char *)malloc(100);
    .
    .
free(buf);
```

Siehe auch

malloc, calloc, realloc, garbcoll

freopen, freopen64 - Dateizeiger neu zuweisen

Definition

```
#include <stdio.h>

FILE *freopen(const char *d_name, const char *art, FILE *dz);
FILE *freopen64(const char *d_name, const char *art, FILE *dz);
```

`freopen` und `freopen64` dienen dazu, einen bereits definierten Dateizeiger einer neuen Datei zuzuordnen. `freopen` und `freopen64` schließen die Datei mit Dateizeiger *dz*, öffnen die Datei *d_name* und ordnen ihr die FILE-Struktur mit Dateizeiger *dz* zu.

Es besteht kein funktionaler Unterschied zwischen `freopen` und `freopen64`, außer dass in der mit dem Filedescriptor verknüpften Dateibeschreibung das Kennzeichen für eine große Datei hinterlegt wird, dh. es wird das O_LARGEFILE Bit gesetzt. Es wird ein Dateizeiger zurückgegeben, der dazu verwendet werden kann, die Datei über 2 GB hinaus zu zu vergrößern.

Für die Bearbeitung von Dateien > 2 GB verfahren Sie wie folgt:

– Falls das Define `_FILE_OFFSET_BITS 64` (siehe Seite 71) gesetzt ist, rufen Sie `freopen` auf. Implizit wird dann `freopen64` mit den passenden Parametern verwendet.

– Andernfalls müssen Sie `freopen64` aufrufen.

Parameter

const char *d_name

Zeichenkette, die die neu zu öffnende Datei angibt. *d_name* kann sein:

– jeder gültige BS2000-Dateiname
– "link=*linkname*"
  linkname bezeichnet einen BS2000-Linknamen
– "(SYSDTA)", "(SYSOUT)", "(SYSLST)"
  die entsprechende Systemdatei
– "(SYSTERM)"
  Terminal-Ein-/Ausgabe
– "(INCORE)"
  temporäre Binärdatei, die nur im virtuellen Speicher angelegt wird

const char *art

Zeichenkette, die die gewünschte Zugriffsart angibt. Mit optionalen Zusatzangaben können weitere Funktionen gesteuert werden:

Zusatzangabe	**Funktion**
tabexp=yes/no	Behandlung des Tabulatorzeichens (\t)
lbp=yes/no	Behandlung des Last Byte Pointers (LBP)

Zugriffsarten:

"r"	Öffnen Textdatei zum Lesen. Die Datei muss bereits vorhanden sein.
"w"	Öffnen Textdatei zum Neuschreiben. Ist die Datei vorhanden, wird der alte Inhalt gelöscht. Ist die Datei nicht vorhanden, wird sie neu erstellt.
"a"	Öffnen Textdatei zum Anfügen ans Ende der Datei. Ist die Datei vorhanden, wird auf das Dateiende positioniert, d.h. der alte Inhalt bleibt erhalten und die neuen Daten werden ans Ende der Datei angehängt. Ist die Datei nicht vorhanden, wird sie neu erstellt.
"rb"	Öffnen Binärdatei zum Lesen. Die Datei muss bereits vorhanden sein.
"wb"	Öffnen Binärdatei zum Neuschreiben. Ist die Datei vorhanden, wird der alte Inhalt gelöscht. Ist die Datei nicht vorhanden, wird sie neu erstellt.
"ab"	Öffnen Binärdatei zum Anfügen ans Ende der Datei. Ist die Datei vorhanden, wird auf das Dateiende positioniert, d.h. der alte Inhalt bleibt erhalten und die neuen Daten werden ans Ende der Datei angehängt. Ist die Datei nicht vorhanden, wird sie neu erstellt.
"r+w", "r+"	Öffnen Textdatei zum Lesen und Schreiben. Die Datei muss bereits vorhanden sein. Der alte Inhalt bleibt erhalten.
"w+r", "w+"	Öffnen Textdatei zum Neuschreiben und Lesen. Ist die Datei vorhanden, wird der alte Inhalt gelöscht. Ist die Datei nicht vorhanden, wird sie neu erstellt.
"a+r", "a+"	Öffnen Textdatei zum Anfügen ans Ende der Datei und zum Lesen. Ist die Datei vorhanden, bleibt der alte Inhalt erhalten und die neuen Daten werden ans Ende der Datei angehängt. Eine bereits vorhandene Datei ist nach dem Öffnen je nach KR- oder ANSI-Funktionalität unterschiedlich positioniert: bei KR-Funktionalität (nur bei C/C++ Versionen kleiner V3.0 vorhanden) auf das Dateiende, bei ANSI-Funktionalität auf den Dateianfang. Ist die Datei nicht vorhanden, wird sie neu erstellt.
"r+b", "rb+"	Öffnen Binärdatei zum Lesen und Schreiben. Die Datei muss bereits vorhanden sein. Der alte Inhalt bleibt erhalten.
"w+b", "wb+"	Öffnen Binärdatei zum Neuschreiben und Lesen. Ist die Datei vorhanden, wird der alte Inhalt gelöscht. Ist die Datei nicht vorhanden, wird sie neu erstellt.
"a+b", "ab+"	Öffnen Binärdatei zum Anfügen ans Ende der Datei und zum Lesen. Ist die Datei vorhanden, bleibt der alte Inhalt erhalten und die neuen Daten werden ans Ende der Datei angehängt. Eine bereits vorhandene Datei ist nach dem Öffnen je nach KR- oder ANSI-Funktionalität unterschiedlich positioniert: bei KR-Funktionalität auf das Dateiende, bei ANSI-Funktionalität auf den Dateianfang. Ist die Datei nicht vorhanden, wird sie neu erstellt.

Tabulatorzeichen (\t)

Im Parameter *art* kann zusätzlich zur Zugriffsart eine Angabe zur Behandlung des Tabulatorzeichens (\t) gemacht werden. Diese Angabe ist nur für Textdateien mit den Zugriffsmethoden SAM und ISAM relevant.

"...,tabexp=yes"

Das Tabulatorzeichen wird in die entsprechende Anzahl Leerzeichen expandiert. Voreinstellung bei KR-Funktionalität (nur bei C/C++ Versionen kleiner V3.0 vorhanden).

"...,tabexp=no"

Das Tabulatorzeichen wird nicht expandiert.
Voreinstellung bei ANSI-Funktionalität.

Last Byte Pointer (LBP)

Im Parameter *art* kann zusätzlich zur Zugriffsart eine Angabe zur Behandlung des Last Byte Pointer (LBP) gemacht werden. Diese Angabe ist nur für Binärdateien mit Zugriffsart PAM relevant und wirkt sich erst aus, wenn die Datei geschlossen wird.

Beim Öffnen und Lesen einer bestehenden Datei wird der LBP unabhängig vom *lbp*-Schalter immer berücksichtigt:

– Ist der LBP der Datei ungleich 0, wird er ausgewertet. Ein eventuell vorhandener Marker wird ignoriert.

– Ist der LBP = 0, wird nach einem Marker gesucht und die Dateilänge daraus ermittelt. Falls kein Marker gefunden wird, wird das Ende des letzten vollständigen Blocks als Dateiende betrachtet.

"...,lbp=yes"

Beim Schließen einer Datei, die verändert oder neu erstellt wurde, wird kein Marker geschrieben (auch wenn einer vorhanden war) und ein gültiger LBP gesetzt. Auf diese Weise können Dateien mit Marker auf LBP ohne Marker umgestellt werden.

"...,lbp=no"

Beim Schließen einer Datei, die verändert oder neu erstellt wurde, wird der LBP auf Null gesetzt. Für eine neu erstellte Datei wird immer ein Marker geschrieben, für eine veränderte nur dann, wenn vorher bereits ein Marker vorhanden war. War kein Marker vorhanden, wird auch keiner geschrieben und die Datei endet mit dem vollständigen letzten Block.

Wird der Schalter *lbp* nicht angegeben, hängt das Verhalten von der Umgebungsvariablen LAST_BYTE_POINTER ab (siehe auch „Umgebungsvariable LAST_BYTE_POINTER“ auf Seite 90):

`LAST_BYTE_POINTER=YES`

Die Funktion verhält sich so, als ob `lbp=yes` angegeben wäre.

`LAST_BYTE_POINTER=NO`

Die Funktion verhält sich so, als ob `lbp=no` angegeben wäre.

FILE *dz
Dateizeiger, der neu zugewiesen werden soll.

Returnwert Zeiger auf den ursprünglichen Dateizeiger *dz*
bei Erfolg.

NULL-Zeiger wenn die Datei nicht geöffnet werden konnte, z.B. wegen fehlender Zugriffsberechtigung, falschem Datei- oder Linknamen etc.

Hinweise Der BS2000-Dateiname bzw. -Linkname kann in Klein- und Großbuchstaben geschrieben werden, er wird automatisch in Großbuchstaben umgesetzt.

Die Datei, der der Dateizeiger *dz* ursprünglich zugeordnet war, wird auch dann geschlossen, wenn die neue Datei nicht geöffnet werden konnte.

Durch die Angabe eines "b" an zweiter bzw. dritter Stelle im Parameter *art* wird die Datei als Binärdatei geöffnet. Die Angabe ist nur für SAM-Dateien relevant, da nur SAM-Dateien sowohl im Binär- als auch im Textmodus verarbeitet werden.
Systemdateien und ISAM-Dateien werden immer als Textdateien verarbeitet. Die Angabe des Binärmodus führt bei diesen Dateien zu einem Fehler beim Öffnen.
(INCORE)- und PAM-Dateien werden immer als Binärdateien verarbeitet. Aus Kompatibilitätsgründen funktioniert das Öffnen als Binärdatei auch ohne explizite Angabe des Binärmodus.

Wird eine nicht vorhandene Datei neu angelegt, wird standardmäßig folgende Datei erzeugt:

	Binärdatei	Textdatei
Zugriffsmethode	SAM	SAM (KR-Funktionalität, nur bei C/C++ Versionen kleiner V3.0 vorhanden) ISAM (ANSI-Funktionalität)
Satzformat	F	V

Bei Verwendung eines Linknamens lassen sich mit dem ADD-FILE-LINK-Kommando folgende Dateiattribute ändern: Zugriffsmethode, Satzlänge, Satzformat, Blocklänge und Blockformat. Siehe Abschnitt „Systemdateien (SYSDTA, SYSOUT, SYSLST)“ auf Seite 73.

In allen Fällen, in denen der alte Inhalt einer bereits existierenden Datei gelöscht wird (geöffnet zum Neuschreiben bzw. zum Neuschreiben und Lesen), bleiben die Katalogeigenschaften dieser Datei erhalten.

Wenn eine Datei zum Ändern geöffnet wird, kann das Lesen und Schreiben über denselben Dateizeiger erfolgen. Dennoch sollte auf eine Ausgabe nicht unmittelbar eine Eingabe erfolgen ohne ein vorhergehendes Positionieren (`fseek`/`fseek64`, `fsetpos`/`fsetpos64`, `rewind`) oder einen `fflush`-Aufruf. Dasselbe gilt für eine Ausgabe, die einer Eingabe folgt.

Position des Schreib-/Lesezeigers im Anfügemodus
Wenn der Schreib-/Lesezeiger in einer Datei, die im Anfügemodus eröffnet wurde, explizit vom Dateiende wegpositioniert wurde (`rewind`, `fsetpos`/`fsetpos64`, `fseek`/`fseek64`), wird er je nach KR- oder ANSI-Funktionalität unterschiedlich behandelt.
KR-Funktionalität (nur bei C/C++ Versionen kleiner V3.0 vorhanden): Der aktuelle Schreib-/Lesezeiger wird nur beim Schreiben mit der Elementarfunktion write ignoriert und automatisch ans Ende der Datei positioniert.
ANSI-Funktionalität: Der aktuelle Schreib-/Lesezeiger wird bei allen Schreibfunktionen ignoriert und automatisch ans Ende der Datei positioniert.

Der Versuch, eine nicht existierende Datei zum Lesen zu öffnen, endet mit Fehler.

(INCORE)-Dateien können nur zum Neuschreiben ("w"), zum Neuschreiben und Lesen ("w+r") oder zum Lesen ("r") eröffnet werden. Es müssen zuerst Daten geschrieben werden. Um die geschriebenen Daten wieder einlesen zu können, gibt es folgende Möglichkeiten: Wurde die Datei nur zum Neuschreiben eröffnet, kann man sie mit der Funktion `freopen` bzw. `freopen64` zum Lesen öffnen. Wurde die Datei zum Neuschreiben und zum Lesen eröffnet, kann man den Lese-/Schreibzeiger mit der Funktion `rewind` auf Dateianfang positionieren.

Sie können eine Datei gleichzeitig für verschiedene Zugriffsmodi eröffnen, sofern diese Modi im BS2000-Datenverwaltungssystem miteinander verträglich sind.

Wenn ein Programm startet, werden ihm automatisch drei Dateizeiger für Standardeingabe, -ausgabe und -fehlerausgabe zugeordnet und zwar:

stdin	Dateizeiger für Standardeingabe (Datensichtstation)
stdout	Dateizeiger für Standardausgabe (Datensichtstation)
stderr	Dateizeiger für Standardfehlerausgabe (Datensichtstation)

`freopen` und `freopen64` werden häufig dazu benutzt, diese Standardzuordnungen auf andere Dateien umzudefinieren. Eine solche Anwendung entspricht dem Umlenkmechanismus der UNIX-Shell (PARAMETER-PROMPTING in der RUNTIME-Option) oder den entsprechenden ASSIGN-Kommandos im BS2000 (siehe auch Beispiel).

Es können maximal _NFILE Dateien gleichzeitig geöffnet sein. _NFILE ist in <stdio.h> mit 2048 definiert.

Satz-E/A

Für das Eröffnen von Dateien mit Satz-E/A ist der Parameter *art* um zwei Angaben erweitert. Diese Angaben folgen in der Zeichenkette hinter der Zugriffsart (s.o.) jeweils durch ein Komma getrennt.

```
"...,type=record [,forg={seq/key}]"
```

type=record	Die Datei wird für satzweise Ein-/Ausgabe eröffnet. Fehlt diese Angabe, wird die Datei für Strom-E/A eröffnet.
forg=seq	Die Dateiorganisation ist sequenziell. Sequenzielle Dateien können SAM- oder PAM-Dateien sein.
forg=key	Die Dateiorganisation ist indexsequenziell. Indexsequenzielle Dateien sind ISAM-Dateien.

Fehlt die Angabe von `forg`, ist die Dateiorganisation vom FCBTYP der Datei abhängig: Der FCBTYP ist durch den Katalogeintrag einer bereits existierenden Datei festgelegt bzw. durch ein ADD-FILE-LINK-Kommando. Für SAM- und PAM-Dateien wird sequenzielle Dateiorganisation angenommen, für ISAM-Dateien indexsequenzielle Dateiorganisation.

Fehlt die Angabe von `forg` und der FCBTYP ist nicht festgelegt (Datei nicht vorhanden, kein ADD-FILE-LINK-Kommando), wird sequenzielle Dateiorganisation angenommen und eine SAM-Datei erstellt.

Für die Satz-E/A gelten folgende Einschränkungen. Werden diese Einschränkungen nicht eingehalten, wird die Datei nicht eröffnet und ein Fehler-Returnwert geliefert:

a) Die Datei muss im Binärmodus eröffnet werden (Angabe "b" bei der Zugriffsart).

b) "type=record" ist zulässig für SAM-, PAM- oder ISAM-Dateien.

c) "forg=seq" ist zulässig für SAM- oder PAM-Dateien, "forg=key" für ISAM-Dateien.

d) Bei "forg=key" ist der Anfügemodus ’a’ unzulässig. Bei ISAM-Dateien bestimmt sich die Position aus dem Schlüssel im Satz.

Beispiel Folgender Programmausschnitt macht die Datei *out* zur Standardausgabedatei.

```
FILE *fp;

fp = freopen("out","w",stdout)
```

fp und `stdout` sind danach beide Dateizeiger für die Datei *out*.

Siehe auch fopen, fopen64, fdopen

frexp - Gleitkommazahl zerlegen in Mantisse und Exponent

Definition

```
#include <math.h>

double frexp(double wert, int *e_zg);
```

`frexp` zerlegt einen Gleitkommawert *wert* in die Mantisse x und den Exponenten n, nach der Formel:

$wert = x * 2^n$

$\|x\|$	liegt im Intervall [0.5, 1.0[
n	ist ganzzahlig

Als Ergebnis liefert `frexp` die Mantisse x und indirekt über einen Ergebnisparameter *e_zg* den ganzzahligen Exponenten n zur Basis 2.
`frexp` ist die Umkehrfunktion von `ldexp`.

Returnwert

Mantisse x	eine Gleitkommazahl vom Typ `double` im Intervall [0.5, 1.0[, die die Gleichung erfüllt: $wert = x * 2^n$.
0	falls *wert* gleich 0 ist (in diesem Fall ist auch der Exponent gleich 0).

Hinweis Beachten Sie, dass das Argument *e_zg* ein Zeiger sein muss!

Beispiel Normierte Darstellung der Zahl 5 zur Basis 2:

```
#include <stdio.h>
#include <math.h>

int main(void)
{
   double z;
   int exp;

   z = frexp((double)5, &exp);
   printf("5 = %g * 2 ** %d\n", z, exp);
   return 0;
}
```

Siehe auch ldexp, modf

fscanf - Formatierte Eingabe aus einer Datei

Definition #include <stdio.h>

int fscanf(FILE *dz, const char *format, argumentenliste);

`fscanf` liest Daten (Eingabefelder) aus einer Datei mit Dateizeiger *dz*, wandelt diese gemäß den Angaben in der Formatzeichenkette *format* um und speichert die Ergebnisse in den Bereichen ab, die Sie mit den Ergebniszeigern in der Argumentenliste angeben.

`fscanf` arbeitet wie `scanf`, nur dass die Eingabefelder nicht von der Standardeingabe (`stdin`), sondern aus einer Datei gelesen werden.

Parameter FILE *dz
Dateizeiger der Eingabedatei.

const char *format
Formatzeichenkette wie bei `scanf` mit KR- oder ANSI-Funktionalität (Beschreibung siehe dort).

argumentenliste
Zeiger auf Variablen, in die `fscanf` die umgewandelten Ergebnisse abspeichern soll. Für %*-Anweisungen (Zuweisung überspringen) in *format* dürfen keine Zeigerargumente angegeben werden. Für alle anderen %-Anweisungen muss es jeweils ein Zeigerargument geben. Der Datentyp des Zeigerarguments richtet sich nach der Typangabe in der zugehörigen Formatanweisung.

Returnwert Anzahl der eingelesenen und erfolgreich umgewandelten Eingabefelder.
Dazu zählen nicht die Eingabefelder, für die %* (Zuweisung überspringen) angegeben wurde.

EOF bei einem Fehler vor Beginn der Umwandlungen.

Hinweis Die ausführliche Beschreibung, Hinweise und Beispiele zur formatierten Eingabe finden Sie bei `scanf`.

Siehe auch scanf, sscanf

fseek, fseeko, fseek64, fseeko64 - Lese-/Schreibzeiger positionieren

Definition

```
#include <stdio.h>

int fseek(FILE *dz, long distanz, int ort);
int fseeko(FILE *dz, off_t distanz; int ort);
int fseek64(FILE *dz, long long distanz, int ort);
int fseeko64(FILE *dz, off64_t distanz, int ort);
```

`fseek`/`fseeko` und `fseek64`/`fseeko64` positionieren den Lese-/Schreibzeiger für die Datei mit Dateizeiger *dz* gemäß den Angaben in *distanz* und *ort*. Sie haben damit die Möglichkeit, eine Datei nicht-sequenziell zu bearbeiten.

In Textdateien (SAM im Textmodus, ISAM) kann man absolut auf Dateianfang und Dateiende positionieren sowie auf eine vorher mit `ftell`/`ftello` bzw. `ftell64`/`ftello64` gemerkte Position.

In Binärdateien (SAM im Binärmodus, PAM, INCORE) kann man sowohl absolut positionieren (s.o.) als auch relativ um eine gewünschte Anzahl Bytes, bezogen auf Dateianfang, Dateiende oder aktuelle Position.

Für die Bearbeitung von Dateien > 2 GB verfahren Sie wie folgt:

– Falls das Define `_FILE_OFFSET_BITS 64` (siehe Seite 71) gesetzt ist, rufen Sie `fseeko` auf. Implizit wird dann `fseeko64` mit den passenden Parametern verwendet. (Die automatische Umsetzung für fseek wird nicht unterstützt.)

– Andernfalls müssen Sie `fseek64` bzw. `fseeko64` aufrufen.

Es besteht kein funktionaler Unterschied zwischen `fseek` und `fseek64` bzw. `fseeko` und `fseeko64`. Die Funktionen unterscheiden sich nur hinsichtlich des verwendeten Offset-Typs.

Parameter

FILE *dz
: Dateizeiger der Datei, deren Lese-/Schreibzeiger positioniert werden soll.

long distanz / off_t distanz / long long distanz / off64_t distanz
: Bedeutung, Kombinationsmöglichkeiten und Wirkung dieser Parameter sind für Text- und Binärdateien unterschiedlich und werden deshalb im Folgenden getrennt beschrieben.

Textdateien (SAM im Textmodus, ISAM)

Mögliche Werte der Parameter:

distanz	0L oder Wert, der durch einen vorhergehenden `ftell`/`ftello`-Aufruf ermittelt wurde.
distanz (64-Bit-Schnittstelle)	0LL oder Wert, der durch einen vorhergehenden `ftell`/`ftello`/`ftell64`/`ftello64`-Aufruf ermittelt wurde.
ort	SEEK_SET (Dateianfang) SEEK_END (Dateiende)

Sinnvolle Kombinationsmöglichkeiten und Wirkung:

distanz	**ort**	**Wirkung**
`ftell`- / `ftello`-Wert bzw. `ftell64`- / `ftello64`-Wert	SEEK_SET	Positionieren auf die durch `ftell`/`ftello` bzw. `ftell64`/`ftello64` ermittelte Position.
0L bzw. 0LL	SEEK_SET	Positionieren auf Dateianfang.
0L bzw. 0LL	SEEK_END	Positionieren auf Dateiende.

Binärdateien (SAM im Binärmodus, PAM, INCORE)

Mögliche Werte der Parameter:

distanz	Anzahl der Bytes, um die der aktuelle Lese-/Schreibzeiger verschoben werden soll, und zwar positive Zahl: Vorwärtspositionieren Richtung Dateiende negative Zahl: Rückwärtspositionieren Richtung Dateianfang 0L: absolut Positionieren auf Dateianfang bzw. -ende.
ort	Bei absoluter Positionierung auf Dateianfang oder -ende, wohin der Lese-/Schreibzeiger verschoben werden soll und bei relativer Positionierung, von wo aus der Lese-/Schreibzeiger um *distanz* Bytes verschoben werden soll: SEEK_SET (Dateianfang) SEEK_CUR (aktuelle Position) SEEK_END (Dateiende)

Sinnvolle Kombinationsmöglichkeiten und Wirkung:

distanz	ort	Wirkung
0L bzw. für 64-Bit: 0LL	SEEK_SET	Positionieren auf Dateianfang.
0L bzw. für 64-Bit: 0LL	SEEK_END	Positionieren auf Dateiende.
positive Zahl	SEEK_SET SEEK_CUR SEEK_END	Vorwärtspositionieren ab Dateianfang, ab aktueller Position, ab Dateiende (über das Dateiende hinaus).
negative Zahl	SEEK_CUR SEEK_END	Rückwärtspositionieren ab aktueller Position, ab Dateiende.
`ftell`- / `ftello` -Wert bzw. `ftell64`- / `ftello64`-Wert	SEEK_SET	Positionieren auf die durch einen `ftell`/`ftello`- bzw. `ftell64`/`ftello64`-Aufruf gemerkte Position.

Returnwert 0 bei Erfolg.

-1 bei Fehler.
Wenn Sie bei einer nur zum Lesen geöffneten Binärdatei hinter das Dateiende positionieren, wird `errno` auf EMDS gesetzt.

Hinweise Der Aufruf `fseek(dz,0L,SEEK_SET)` bzw. `fseek64(dz,0LL,SEEK_SET)` ist äquivalent zu dem Aufruf `rewind(dz)`.

Werden in eine Textdatei neue Sätze geschrieben (geöffnet zum Neuerstellen oder Anhängen) und erfolgt ein `fseek`/`fseeko`- bzw. `fseek64`/`fseeko64`-Aufruf, dann werden zunächst ggf. restliche Daten aus dem Puffer in die Datei geschrieben und mit Zeilenende (\n) abgeschlossen.
Ausnahme bei ANSI-Funktionalität:
Wenn die Daten einer ISAM-Datei im Puffer nicht mit einem Neue-Zeile-Zeichen enden, bewirkt `fseek`/`fseeko` bzw. `fseek64`/`fseeko64` keinen Zeilenwechsel (bzw. Satzwechsel). D.h., die Daten werden beim Schreiben aus dem Puffer nicht automatisch mit einem Neue-Zeile-Zeichen abgeschlossen. Nachfolgende Daten verlängern den Satz in der Datei. Beim Lesen einer ISAM-Datei werden daher nur Neue-Zeile-Zeichen eingelesen, die vom Programm explizit geschrieben wurden.

Wenn Sie bei einer zum Schreiben geöffneten Binärdatei hinter das Dateiende positionieren, entsteht ein „Loch“ zwischen den letzten physisch gespeicherten Daten und den neu geschriebenen Daten. Lesen aus diesem „Loch“ liefert binäre Nullen.
Wenn Sie bei einer nur zum Lesen geöffneten Binärdatei hinter das Dateiende positionieren, führt das zu einem Fehler (EMDS).

Auf Systemdateien (SYSDTA, SYSLST, SYSOUT) kann nicht positioniert werden.

Ein erfolgreicher Aufruf der Funktion `fseek`/`fseeko` bzw. `fseek64`/`fseeko64` löscht das EOF-Flag der Datei und hebt alle Effekte der vorangegangenen `ungetc`-Aufrufe für diese Datei auf.

Satz-E/A

`fseek`/`fseeko` und `fseek64`/`fseeko64` können nur zum Positionieren auf Dateianfang oder Dateiende verwendet werden.

`fseek(dz,0L,SEEK_SET)` bzw. `fseek64(dz,0LL,SEEK_SET)` positioniert auf den ersten Satz der Datei.

`fseeko(dz,0L,SEEK_SET)` bzw. `fseeko64(dz,0LL,SEEK_SET)` positioniert auf den ersten Satz der Datei.

`fseek(dz,0L,SEEK_END)` bzw. `fseek64(dz,0LL,SEEK_SET)` positioniert hinter den letzten Satz der Datei.

`fseeko(dz,0L,SEEK_END)` bzw. `fseeko64(dz,0LL,SEEK_SET)` positioniert hinter den letzten Satz der Datei.

Bei Aufrufen mit anderen Argumenten liefern `fseek`/`fseeko` und `fseek64`/`fseeko64` EOF.

Beispiel 1

Folgendes Programm liest die Datei *dat* ab dem elften Zeichen bis Dateiende (funktioniert nur für Binärdateien).

```
#include <stdio.h>

int main(void)
{
  FILE *fp;
  int c;

  if((fp = fopen("dat","rb")) != NULL)
  {
              /* die ersten 10 Zeichen überspringen */
    fseek(fp,10L,SEEK_SET);

    while((c=getc(fp)) != EOF)
          putc((char)c,stdout);
    fclose(fp);
  }
  return 0;
}
```

Beispiel 2 Folgendes Programm verarbeitet eine Datei im Update-Modus. Kleinbuchstaben werden als Großbuchstaben zurückgeschrieben, alle anderen Zeichen bleiben unverändert.

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
   FILE *fp;
   int c;
   long n;
   fp = fopen("link=link","r+w");
     do
     {
       n = ftell(fp);
       c = getc(fp);
       if (islower(c) == 0) continue; /* Wenn Zeichen kein Kleinbuchstabe, */
                                      /* nächstes Zeichen einlesen */


       else
       {                              /* Wenn Zeichen ein Kleinbuchstabe, */
         fseek(fp, n, SEEK_SET);      /* auf dieses positionieren und als */
         fputc((toupper(c)), fp);     /* Großbuchstaben zurückschreiben. */
       }
     }
     while(c != EOF);
     fclose(fp);
     return 0;
}
```

Siehe auch ftell, ftello, ftell64, ftello64, fsetpos, fsetpos64, lseek, lseek64, rewind, tell

fsetpos, fsetpos64 - Lese-/Schreibzeiger positionieren

Definition

```
#include <stdio.h>

int fsetpos(FILE *dz, const fpos_t *pos);
int fsetpos64(FILE *dz, const fpos64_t *pos);
```

`fsetpos` und `fsetpos64` positionieren den Lese-/Schreibzeiger der Datei mit Dateizeiger *dz* auf eine zuvor mit `fgetpos` bzw. `fgetpos64` ermittelte Position *pos*.
Nach der Positionierung kann die nächste Operation sowohl eine Lese- als auch eine Schreibfunktion sein.

Für die Bearbeitung von Dateien > 2 GB verfahren Sie wie folgt:

– Falls das Define `_FILE_OFFSET_BITS 64` (siehe Seite 71) gesetzt ist, rufen Sie `freopen` auf. Implizit wird dann `freopen64` mit den passenden Parametern verwendet.

– Andernfalls müssen Sie `freopen64` aufrufen.

Es besteht kein funktioneller Unterschied zwischen `fsetpos` und `fsetpos64`, außer dass `fsetpos64` einen `fpos64_t` Typ verwendet.

Returnwert

0	bei erfolgreicher Ausführung von `fsetpos`.
≠ 0	im Fehlerfall. Zusätzlich wird `errno` auf EBADF gesetzt.

Hinweise

`fsetpos` und `fsetpos64` lassen sich auf Binärdateien (SAM im Binärmodus, PAM, INCORE) und Textdateien (SAM im Textmodus, ISAM) anwenden.
`fsetpos` und `fsetpos64` sind nicht anwendbar für Systemdateien (SYSDTA, SYSLST, SYSOUT).

Ein erfolgreicher Aufruf der Funktion `fsetpos` bzw. `fsetpos64` löscht das EOF-Flag der Datei und hebt alle Effekte der vorangegangenen `ungetc`-Aufrufe für diese Datei auf.

Werden in eine Textdatei neue Sätze geschrieben (geöffnet zum Neuerstellen oder Anhängen) und erfolgt ein `fsetpos`- bzw. `fsetpos64`-Aufruf, dann werden zunächst ggf. restliche Daten aus dem Puffer in die Datei geschrieben und mit Zeilenende (\n) abgeschlossen.
Ausnahme bei ANSI-Funktionalität:
Wenn die Daten einer ISAM-Datei im Puffer nicht mit einem Neue-Zeile-Zeichen enden, bewirkt `fsetpos` bzw. `fsetpos64` keinen Zeilenwechsel (bzw. Satzwechsel). D.h., die Daten werden beim Schreiben aus dem Puffer nicht automatisch mit einem Neue-Zeile-Zeichen abgeschlossen. Nachfolgende Daten verlängern den Satz in der Datei. Beim Lesen einer ISAM-Datei werden daher nur Neue-Zeile-Zeichen eingelesen, die vom Programm explizit geschrieben wurden.

Für ISAM-Dateien sind die Funktionspaare `fgetpos`/`fsetpos` bzw. `fgetpos64`/`fsetpos64` wesentlich performanter als die vergleichbaren Funktionspaare `ftell`/`fseek` bzw. `ftell64`/`fseek64`.

Satz-E/A In ISAM-Dateien mit Schlüsselverdoppelung kann mit `fsetpos` bzw. `fsetpos64` nicht auf den zweiten oder höheren Schlüssel einer Gruppe mit gleichen Schlüsseln positioniert werden. Dies lässt sich nur durch sequenzielles Lesen bzw. Löschen erreichen.
Mit `fsetpos` und `fsetpos64` kann nur auf den ersten Satz oder hinter den letzten Satz einer solchen Gruppe positioniert werden.

Siehe auch fgetpos, fgetpos64, fseek, fseek64, ftell, ftell64

ftell, ftello, ftell64, ftello64 - Aktuelle Position des Lese-/Schreibzeigers ermitteln

Definition

```
#include <stdio.h>

long ftell(FILE *dz);
off_t ftello(FILE *dz);
long long ftell64(FILE *dz);
off64_t ftello64(FILE *dz);
```

`ftell`/`ftello` und `ftell64`/`ftello64` liefern die aktuelle Position des Lese-/Schreibzeigers für die Datei mit Dateizeiger *dz*.

`ftell`/`ftello` und `ftell64`/`ftello64` lassen sich auf Binärdateien (SAM im Binärmodus, PAM, INCORE) und Textdateien (SAM im Textmodus, ISAM) anwenden.

Für die Bearbeitung von Dateien > 2 GB verfahren Sie wie folgt:

– Falls das Define `_FILE_OFFSET_BITS 64` (siehe Seite 71) gesetzt ist, rufen Sie `ftello` auf. Implizit wird dann `ftello64` mit den passenden Parametern verwendet.

– Andernfalls müssen Sie `ftell64` bzw. `ftello64` aufrufen.

Es besteht kein funktionaler Unterschied zwischen `ftell` und `ftell64` bzw. `ftello` und `ftello64`. Die Funktionen unterscheiden sich nur hinsichtlich des für den Rückgabewert verwendeten Offset-Typs.

Returnwert

Position in der Datei bei Erfolg, und zwar
bei Binärdateien die Anzahl Bytes, die der Lese-/Schreibzeiger vom Dateianfang entfernt ist,
bei Textdateien die absolute Position des Lese-/Schreibzeigers.

-1 im Fehlerfall. Wenn der Wert für die Dateiposition nicht innerhalb des Wertebereichs des Rückgabetyps liegt, wird zusätzlich `errno` auf ERANGE gesetzt.

Hinweise

Auf die von `ftell`/`ftello` bzw. `ftell64`/`ftello64` gelieferte Position kann mit den Funktionen `fseek`/`fseeko` bzw. `fseek64`/`fseeko64` positioniert werden.

`ftell`/`ftello` und `ftell64`/`ftello64` sind nicht anwendbar für Systemdateien (SYSDTA, SYSLST, SYSOUT).

Beispiel In folgendem Programm wird ab dem elften Zeichen jedes Zeichen von *dat* mit der Position des Lese-/Schreibzeigers ausgegeben (funktioniert nur mit Binärdateien).

```
#include <stdio.h>

int main(void)
{
   FILE *fp;
   int c;
   if((fp = fopen("dat","rb")) != NULL)
     {
               /* die ersten 10 Zeichen werden übersprungen */
       fseek(fp,10L,SEEK_SET);
       while((c=getc(fp)) != EOF)
            printf("Position : %ld, Zeichen : %c\n",ftell(fp),c);
       fclose(fp);
     }
   return 0;
}
```

Siehe auch fseek, fseek64, fgetpos, fgetpos64, ftell, ftell64, tell

ftime, ftime64 - Aktuelle Zeit

Definition

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

int ftime(struct timeb *zg);
int ftime64(struct timeb64 *zg);
```

`ftime` und `ftime64` liefern in der Struktur, auf die *zg* zeigt, die aktuelle Zeit (Ortszeit) als Anzahl der seit dem Stichtag (Epoche) vergangenen Sekunden und Millisekunden.

Bei `ftime` hängt der Stichtag von der Verwendung des TIMESHIFT-Bindeschalters ab (siehe Abschnitt „Zeitfunktionen“ auf Seite 41):
– ohne TIMESHIFT-Bindeschalter (Standard): 1.1.1950 00:00:00.
– mit TIMESHIFT-Bindeschalter: 1.1.1970 00:00:00.
Bei `ftime64` ist der Stichtag immer der 1.1.1970 00:00:00.

Ab dem 19.01.2018 03:14:08 (ohne TIMESHIFT-Bindeschalter) bzw. ab dem 19.01.2038 03:14:08 (mit TIMESHIFT-Bindeschalter) gibt `ftime` die Meldung `CCM0014` aus und beendet das Programm.

`ftime64` liefert unabhängig von der Verwendung des TIMESHIFT-Bindeschalters bis zum 18.3.4317 02:44:48 korrekte Ergebnisse.

Aus Gründen der Portabilität sind weitere Komponenten in den Strukturen `timeb` und `timeb64` enthalten. Sie werden jedoch in BS2000-Umgebung nicht versorgt.

Returnwert Immer 0.

Hinweise Wie immer in solchen Fällen müssen Sie den Speicherplatz für die Ergebnisstruktur explizit bereitstellen!

Siehe auch time, time64, ctime, ctime64

fwide - Orientierung einer Datei festlegen

Definition
```
#include <stdio.h>
#include <wchar.h>

int fwide(FILE *dz, int mode);
```

`fwide` legt die Orientierung der Datei mit dem Dateizeiger *dz* fest, sofern diese noch keine Orientierung hat. Ist die Orientierung bereits festgelegt - zum Beispiel durch eine vorherige Ein-/Ausgabe-Operation - verändert `fwide` diese Orientierung nicht.

Abhängig vom Argument *mode* versucht `fwide`, die Orientierung folgendermaßen einzustellen:

mode > 0	Datei wird Langzeichen-orientiert.
mode < 0	Datei wird Byte-orientiert.
mode = 0	die Orientierung der Datei wird nicht verändert.

Returnwert

> 0	wenn *dz* nach dem Aufruf von `fwide` Langzeichen-orientiert ist.
< 0	wenn *dz* nach dem Aufruf von `fwide` Byte-orientiert ist.
0	wenn *dz* keine Orientierung hat.

Hinweis In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

fwprintf, swprintf, vfwprintf, vswprintf, vwprintf, wprintf - Langzeichen formatiert ausgeben

Definition

```
#include <stdio.h>
#include <wchar.h>

int fwprintf(FILE *dz, const wchar_t *format [, arglist]);

#include <stdarg.h>
#include <wchar.h>

int vwprintf(const wchar_t *format, va_list arg);

#include <wchar.h>

int wprintf(const wchar_t *format [, arglist]);
int swprintf(wchar_t *s, size_t n, const wchar_t *format [, arglist]);

#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>

int vfwprintf(FILE *dz, const wchar_t *format, va_list arg);
int vswprintf(wchar_t *s, size_t n, const wchar_t *format, va_list arg);
```

Die Funktionen dienen der formatierten Ausgabe.

`fwprintf` bereitet die Argumente in der Liste *arglist* gemäß den Angaben in der Langzeichenkette *format* auf und schreibt sie in die Datei mit dem Dateizeiger *dz*.
`fwprintf` kehrt zurück, wenn das Ende von *format* erreicht wird.

`vwprintf` entspricht der Funktion `fwprintf` mit *dz* = `stdout`, wobei die Argumentliste durch ein Argument vom Typ `va_list` ersetzt wird, das durch das Makro `va_start` initialisiert worden sein muss (möglicherweise gefolgt von `va_arg`-Aufrufen). Die Funktion ruft nicht das Makro `va_end` auf.

`wprintf` entspricht der Funktion `fwprintf` mit *dz* = `stdout`.

`swprintf` schreibt Ausgaben formatiert in die Langzeichenkette *s*. `swprintf` entspricht ansonsten der Funktion `fwprintf`. Es werden maximal *n* Langzeichen geschrieben, inklusive des abschließenden Nullzeichens, das für *n* > 0 automatisch angefügt wird.

`vfwprintf` entspricht der Funktion `fwprintf`, wobei die Liste durch ein Argument vom Typ `va_list` ersetzt wird, das durch das Makro `va_start` initialisiert worden sein muss (möglicherweise gefolgt von `va_arg`-Aufrufen). Die Funktion ruft nicht das Makro `va_end` auf.

`vswprintf` entspricht der Funktion `swprintf`, wobei die Liste durch ein Argument vom Typ `va_list` ersetzt wird, das durch das Makro `va_start` initialisiert worden sein muss (möglicherweise gefolgt von `va_arg`-Aufrufen). Die Funktion ruft nicht das Makro `va_end` auf.

Parameter *format* ist eine Langzeichenkette, die keine, eine oder mehrere Umwandlungsanweisungen und Langzeichen enthält:

- Umwandlungsanweisungen beginnen mit dem Prozentzeichen (`%`). Jede Umwandlungsanweisung wird keinem, einem oder mehreren Argumenten in *arglist* zugeordnet. Wenn in *arglist* weniger Argumente übergeben werden, als in *format* festgelegt sind, ist das Ergebnis undefiniert. Wenn in *format* weniger Argumente festgelegt sind, als in *arglist* übergeben werden, werden die überflüssigen Argumente ignoriert.
  Die einer Umwandlungsanweisung zugeordneten Argumente werden gemäß der Anweisung konvertiert und formatiert in den Ausgabedatenstrom geschrieben.
- Zeichen vom Typ `wchar_t` (aber nicht %), die 1 : 1 in die Ausgabe kopiert werden.
- Zwischenraumzeichen (siehe Abschnitt „Zwischenraum“ auf Seite 68)

Umwandlungsanweisungen

Jede Umwandlungsanweisung wird mit dem Zeichen `%` eingeleitet; darauf folgen:

- Keines oder mehrere **Formatierungszeichen**, die die Bedeutung der Umwandlungsanweisung verändern.
- Eine optionale Ganzzahl (bestehend aus Dezimalziffern) oder ein Asterisk (*), die eine minimale **Feldbreite** für die Ausgabe eines Arguments angibt. Wenn der umgewandelte Wert aus weniger Zeichen als der Feldbreite besteht, wird links bis zur Feldbreite aufgefüllt (bzw. rechts, wenn das Formatierungszeichen "–" für linksbündige Ausrichtung angegeben wurde).
- Eine optionale **Genauigkeit**, die angibt, wie viele Ziffern mindestens für die Umwandlungen `d`, `i`, `o`, `u`, `x` oder `X` erscheinen sollen, wie viele Ziffern nach dem Dezimalzeichen für die Umwandlungen `e`, `E` und `f` erscheinen sollen, wie viele signifikante Stellen bei den Umwandlungen `g` und `G` vorhanden sind oder wie viele Zeichen maximal aus der Zeichenkette für die Umwandlung *s* ausgegeben werden sollen. Die Genauigkeit hat die Form ".", gefolgt von einer Ganzzahl aus dezimalen Ziffern oder einem Asterisk (*). Ist nur der Punkt angegeben, wird 0 als Genauigkeit eingesetzt.
- Ein optionales `h`, `l` oder `L` vor einem Umwandlungszeichen:
  `l` vor `c` bedeutet, dass ein Argument vom Typ `wint_t` umgewandelt werden soll;
  `l` vor `s`: bedeutet, dass ein Argument vom Typ `wchar_t` (Zeiger auf eine Langzeichenkette) umgewabdelt werden soll ;
  `h` vor `d`, `i`, `o`, `u`, `x` oder `X`: Umwandlung eines Arguments vom Typ `short int` oder `unsigned short int` (das Argument ist entsprechend der ganzzahligen Erweiterung

erweitert worden, und sein Wert wird vor der Ausgabe in ein `short int` oder `unsigned short int` umgewandelt);
`h` vor `n`: Umwandlung eines Arguments vom Typ Zeiger auf `short int`;

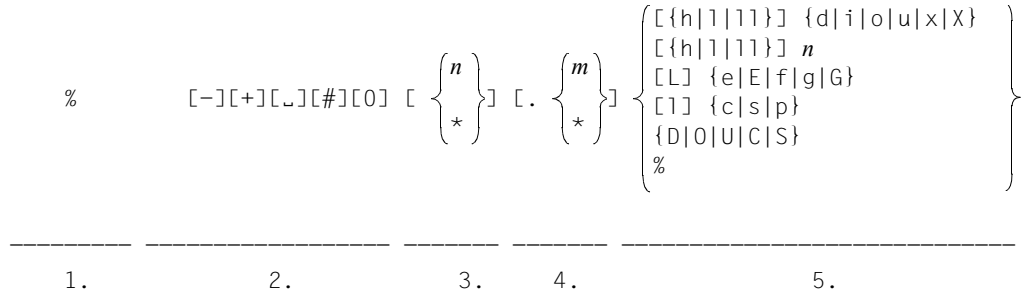
`l` vor `d`, `i`, `o`, `u`, `x` oder `X`: Umwandlung eines Arguments vom Typ `long int` oder `unsigned long int`;
`l` vor `n`: Umwandlung eines Arguments vom Typ Zeiger auf `long int`;
`ll` vor `d`, `i`, `o`, `u`, `x` oder `X`: Umwandlung eines Arguments vom Typ `long long int` oder `unsigned long long int`;
`ll` vor `n`: Umwandlung eines Arguments vom Typ Zeiger auf `long long int`;
`L` vor `e`, `E`, `f`, `g` oder `G`: Umwandlung eines Arguments vom Typ `long double`.

Wenn `h` , `l` oder `L` vor einem anderen Umwandlungszeichen steht, ist das Verhalten undefiniert.

– Ein **Umwandlungszeichen** vom Typ `wchar_t`, das den Typ der durchzuführenden Umwandlung angibt, siehe Auflistung unten.

Feldbreite, Genauigkeit oder beides können durch das Zeichen * (Asterisk) angegeben werden. In diesem Fall werden die Werte statt aus der Formatangabe aus der Argumentliste entnommen: Die (ganzzahligen) Werte für Feldbreite und/oder Genauigkeit müssen unmittelbar vor dem Argument stehen, das umgewandelt werden soll.
Ist eine negative Feldbreite angegeben, wird "–" als Formatierungszeichen interpretiert, dem eine positive Feldbreite folgt. Eine negative Genauigkeit wird interpretiert, als ob die Genauigkeit weggelassen wird.

Umwandlungsanweisungen sehen also wie folgt aus:



1. Anfang einer Umwandlungsanweisung
2. Formatierungszeichen
3. Feldbreite
4. Genauigkeit
5. Zeichen, die die eigentliche Umwandlung festlegen

Formatierungszeichen

− Das Ergebnis der Umwandlung wird linksbündig innerhalb des Felds ausgerichtet.

\+ Das Ergebnis einer Umwandlung mit Vorzeichen wird immer mit einem Vorzeichen ausgegeben (+ oder −).

␣ Wenn das erste Langzeichen einer vorzeichenbehafteten Umwandlung kein Vorzeichen ist oder das Ergebnis einer vorzeichenbehafteten Umwandlung keine Langzeichen ergibt, wird dem Resultat ein Leerzeichen vorangestellt. Wird sowohl das Leerzeichen als auch das Zeichen + angegeben, wird das Formatierungszeichen Leerzeichen ignoriert.

\# Dieses Formatierungszeichen gibt an, dass der umzuwandelnde Wert in einer "alternativen Form" darzustellen ist. Für die Umwandlung o wird die Genauigkeit so weit erhöht, dass die erste Ziffer des Ergebnisses die Ziffer 0 ist. Für x (oder X) wird einem Resultat ungleich 0 die Zeichenfolge "0x" (oder "0X") vorangestellt. Für e, E, f, g oder G enthält das Ergebnis immer ein Dezimapunktl-Langzeichen, auch wenn keine weiteren Ziffern folgen (normalerweise erscheint ein Dezimalpunkt-Langzeichen nur dann im Ergebnis, wenn ihm eine Ziffer folgt). Für g und G werden abschließende Nullen nicht aus dem Ergebnis entfernt, wie sonst üblich.
Das Verhalten bei anderen Umwandlungszeichen ist undefiniert.

`0` Für `d`, `i`, `o`, `u`, `x`, `X`, `e`, `E`, `f`, `g` und `G` werden zum Auffüllen bis zur Feldbreite führende Nullen verwendet (nach Anzeige eines Vorzeichens oder einer Basis); es wird nicht mit Leerzeichen aufgefüllt. Wenn sowohl das Formatierungszeichen `0` als auch `-` angegeben werden, wird das Formatierungszeichen `0` ignoriert.
Ist eine Genauigkeit angegeben, wird für `d`, `i`, `o`, `u`, `x` und `X` das Formatierungszeichen `0` ignoriert. Für andere Umwandlungen ist das Verhalten undefiniert.

Umwandlungszeichen

`d, i` Das `int`-Argument wird in eine vorzeichenbehaftete Dezimalzahl der Form [−]*dddd* umgewandelt. Die Genauigkeit legt die minimale Anzahl von Ziffern fest, die ausgegeben werden sollen. Wenn der umzuwandelnde Wert weniger Ziffern ergibt, wird er um führende Nullen erweitert. Die voreingestellte Genauigkeit ist 1.
Die Umwandlung des Werts 0 mit einer ausdrücklich genannten Genauigkeit von 0 liefert kein Langzeichen.

`o, u` Das `unsigned int`-Argument wird in eine vorzeichenlose Oktalzahl (o) oder in eine vorzeichenlose Dezimalzahl (u) der Form *dddd* umgewandelt. Die Genauigkeit legt die minimale Anzahl von Ziffern fest, die erscheinen sollen; wenn der umzuwandelnde Wert weniger Ziffern ergibt, wird er um führende Nullen erweitert. Die voreingestellte Genauigkeit ist 1.
Die Umwandlung des Werts 0 mit einer ausdrücklich genannten Genauigkeit von 0 liefert kein Langzeichen.

`x, X` Das `unsigned int`-Argument wird in ein eine vorzeichenlose Hexadezimalzahl der Form *dddd* umgewandelt; außer den Zahlen werden die Buchstaben `abcdef` (bei `x`) bzw. ABCDEF (bei `X`) als numerische Zeichen verwendet. Die Genauigkeit legt die minimale Anzahl von Ziffern fest, die erscheinen sollen; wenn der umzuwandelnde Wert weniger Ziffern ergibt, wird er um führende Nullen erweitert. Die voreingestellte Genauigkeit ist 1.
Die Umwandlung des Werts 0 mit einer ausdrücklich genannten Genauigkeit von 0 liefert kein Langzeichen.

`f` Das `double`-Argument wird in die dezimale Schreibweise der Form [−]*ddd*.*ddd* umgewandelt, wobei die Anzahl der Ziffern nach dem Dezimalzeichen gleich der angegebenen Genauigkeit ist.
Ist keine Genauigkeit angegeben, wird die Genauigkeit 6 eingesetzt.
Wenn die Genauigkeit gleich 0 ist und kein #-Formatierungszeichen gesetzt ist, wird kein Dezimalzeichen ausgegeben.

Wenn das Dezimalzeichen erscheint, wird davor mindestens eine Ziffer ausgegeben. Der Wert wird auf die entsprechende Zahl von Ziffern gerundet.

`e, E` Das `double`-Argument wird in die Form `[−]`*d.ddde±dd* umgewandelt, wobei genau eine Ziffer vor dem Dezimalzeichen ausgegeben wird (diese Ziffer ist ungleich 0, wenn das Argument ungleich 0 ist). Die Anzahl der Nachkommastellen ist gleich der Genauigkeit. Ist keine Genauigkeit angegeben, wird die Genauigkeit 6 eingesetzt.
Wenn die Genauigkeit gleich 0 und kein #-Formatierungszeichen gesetzt ist, wird kein Dezimalzeichen ausgegeben. Der Wert wird auf die entsprechende Zahl von Ziffern gerundet.
Das Umwandlungszeichen `E` erzeugt eine Zahl mit `E` an Stelle von `e` für die Anzeige des Exponenten. Der Exponent enthält immer mindestens zwei Ziffern. Wenn der Wert gleich 0 ist, ist der Exponent gleich 0.

`g, G` Das `double`-Argument wird in die Form von `f` oder `e` umgewandelt (bzw. in die Form `E` für das Umwandlungszeichen `G`). Die Genauigkeit gibt die Anzahl der signifikanten Stellen an. Die Angabe einer Genauigkeit 0 wird durch Genauigkeit 1 ersetzt.
Die Form hängt vom umgewandelten Wert ab; die Form `e` wird nur dann verwendet, wenn der Exponent einer solchen Umwandlung kleiner als -4 oder größer gleich der Genauigkeit ist. Abschließende Nullen werden vom gebrochenen Teil des Ergebnisses entfernt; ein Dezimalzeichen erscheint nur dann, wenn es von einer Ziffer gefolgt wird.

`c` Ist das Zeichen `l` vorangestellt, wird das Argument vom Typ `wint_t` in den Typ `wchar_t` umgewandelt, das resultierende Zeichen wird geschrieben.
Ist kein `l` vorangestellt, wird das Argument vom Typ `int` wie beim Aufruf der Funktion `btowc` in ein Langzeichen umgewandelt; das resultierende Zeichen wird geschrieben.

`s` Ist kein Zeichen `l` vorangestellt, soll das Argument vom Typ Zeiger auf ein `char`-Feld sein. Zeichen aus dem Feld werden so konvertiert wie bei Aufrufen der Funktion `mbrtowc`. Der Konversions-Status wird in einem Objekt vom Typ `mbstate_t` beschrieben und mit 0 initialisiert, bevor das erste Multibyte-Zeichen konvertiert wird. Es wird bis zum abschließenden Nullzeichen geschrieben (ausschließlich).
Ist das Zeichen `l` vorangestellt, soll das Argument vom Typ Zeiger auf ein `wchar_t`-Feld sein. Langzeichen aus dem Feld werden bis zum abschließenden Nullzeichen geschrieben (ausschließlich).

Wenn eine Genauigkeit m angegeben ist, werden nicht mehr als m Langzeichen geschrieben. Wird die Genauigkeit nicht angegeben oder ist diese größer als die Länge des konvertierten Feldes, sollte das Feld das Langzeichen 0 enthalten (als Endekriterium).

`S` entspricht `ls`.

`C` entspricht `lc`.

`p` Das Argument muss ein Zeiger auf `void` sein. Die Ausgabe erfolgt als 8-stellige Sedezimalzahl.

`n` Das Argument muss ein Zeiger auf `int` sein, in welches die Anzahl der bisher von `fwprintf` beim aktuellen Aufruf geschriebenen Bytes eingetragen wird. Es wird kein Argument umgewandelt.

`%` Es wird das Langzeichen `%` ausgegeben; es wird kein Argument umgewandelt. Die vollständige Umwandlungsanweisung muss die Form %% haben.

Wenn das Zeichen nach `%` kein gültiges Umwandlungszeichen ist, ist das Ergebnis der Umwandlung undefiniert.

Falls ein Argument eine `UNION` oder ein Zeiger auf eine `UNION` ist, ist das Ergebnis der Umwandlung undefiniert.
Das Gleiche gilt, wenn ein Argument ein Feld oder ein Zeiger auf ein Feld ist, ausgenommen die drei folgenden Fälle:
das Argument ist ein Feld vom Typ `char` und verwendet `%s`,
das Argument ist ein Feld vom Typ `wchar_t` und verwendet `%ls` oder
das Argument ist ein Zeiger und verwendet `%p`.

In keinem Fall verursacht eine nicht existierende oder zu kleine Feldbreite das Abschneiden eines Feldes; wenn das Ergebnis einer Umwandlung breiter als die Feldbreite ist, wird das Feld einfach erweitert, um die Ausgabe aufzunehmen.

Returnwert Anzahl der ausgegebenen Langzeichen
bei erfolgreicher Beendigung.

negativer Wert bei Fehler.

Hinweis In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Siehe auch btowc, fprintf, mbrtowc, printf

fwscanf, swscanf, wscanf - formatiert lesen

Definition

```
#include <stdio.h>
#include <wchar.h>

int fwscanf(FILE *dz, const wchar_t *format [, arglist]);


#include <wchar.h>

int swscanf(const wchar_t *s, const wchar_t *format [, arglist]);
int wscanf(const wchar_t *format [, arglist]);
```

Die Funktionen dienen der formatierten Eingabe.
Sie lesen Eingaben, wandeln sie gemäß den Angaben in der Formatzeichenkette *format* um und speichern die Ergebnisse in den Bereichen ab, die in der optionalen Argumentliste *arglist* angegeben wurden.

`fwscanf` liest Eingaben formatiert aus der Datei mit dem Dateizeiger *dz*.

`swscanf` liest Eingaben formatiert aus der Langzeichenkette *s*. `swscanf` entspricht ansonsten der Funktion `fwscanf`. Das Ende der Langzeichenkette entspricht EOF.

`wscanf` liest Eingaben formatiert aus der Standardeingabe `stdin`. `wscanf` entspricht der Funktion `fwscanf` mit *dz* = `stdin`.

Parameter *format* ist eine Zeichenkette, die in ihrem anfänglichen Umschaltmodus beginnt und endet, (sofern ein Umschaltmodus definiert ist) und keine, eine oder mehrere Umwandlungsanweisungen enthält. *format* kann drei Arten von Zeichen enthalten:

- Zeichen vom Typ `wchar_t` (aber kein Zwischenraumzeichen oder %), die 1 : 1 in den Ausgabedatenstrom kopiert werden.
- Zwischenraumzeichen, beginnend mit einem Gegenschrägstrich (`\`) (siehe `iswspace`).
- Umwandlungsanweisungen, beginnend mit dem Prozentzeichen (%), von denen jede keinem, einem oder mehreren Argumenten in *arglist* zugeordnet wird. Wenn in *arglist* weniger Argumente übergeben werden, als in *format* festgelegt sind, ist das Ergebnis undefiniert. Wenn in *format* weniger Argumente festgelegt sind, als in *arglist* übergeben werden, werden die überflüssigen Argumente ignoriert.

Die `wscanf`-Funktionen lesen das Eingabezeichen zunächst ohne es umzuwandeln und in einer Variablen abzuspeichern. Stimmt das Eingabezeichen nicht mit dem in *format* angegebenen Zeichen überein, wird die Eingabebearbeitung abgebrochen und die Funktion kehrt zurück. Wenn die Umwandlung wegen eines nicht passenden Langzeichens abbricht, verbleibt dieses Zeichen ungelesen im Eingabestrom.

Zwischenraumzeichen

Die Formatzeichenkette *format* kann beliebig viele oder keine Zwischenraumzeichen enthalten. Diese Zeichen haben keine Steuerfunktion.

Zwischenraumzeichen in der Eingabe werden als Trennzeichen zwischen Eingabefeldern behandelt und nicht mit umgewandelt (Ausnahme siehe `%c`, `%n` und `%[]`). Führende Zwischenraumzeichen werden bei der Eingabe ignoriert.

Umwandlungsanweisungen

Alle Formen von `fwscanf` erlauben das Erkennen eines landessprach-spezifischen Dezimalzeichens in der Eingabezeichenkette. Das Dezimalzeichen wird durch die Lokalität des Progamms definiert (Kategorie `LC_NUMERIC`). In der Lokalität `POSIX` oder einer Lokalität, bei der das Dezimalzeichen nicht definiert ist, ist das Dezimalzeichen auf . (Punkt) voreingestellt.

Jede Umwandlungsanweisung muss mit einem Prozentzeichen (`%`) beginnen; darauf folgen:

– Ein optionales Langzeichen Stern (*) zum Überspringen einer Zuweisung.

– Eine optionale Ganzzahl (Dezimalziffern) ungleich 0, welche die maximale **Feldbreite** angibt.

– Ein optionales `h` ,`l` oder `L`, das die Größe des aufnehmenden Objekts angibt:
`l` vor den Umwandlungszeichen `c`, `s` und `[` : das entsprechende Argument ist ein Zeiger auf `wchar_t` .
`h` bzw. `l` vor `d`, `i` und `n` : das entsprechende Argument ist ein Zeiger auf `short int` (`h`) bzw. `long int` (`l`).
`h` bzw. `l` vor `o`, `u` und `x` : das entsprechende Argument ist ein Zeiger auf `unsigned short int` (`h`) bzw. `unsigned long int` (`l`).
`ll` vor `d`, `i` und `n` : das entsprechende Argument ist ein Zeiger auf `long long int`.
`ll` vor `o`, `u` und `x` : das entsprechende Argument ist ein Zeiger auf `unsigned long long int`.
`l` bzw. `L` vor `e`, `f` und `g` : das entsprechende Argument ist ein Zeiger auf `double` (`l`) bzw.`long double (L)`.

  Wenn `h` ,`l` oder `L` vor einem anderen Umwandlungszeichen steht, ist das Verhalten undefiniert.

– Ein **Umwandlungszeichen**, das den Typ der durchzuführenden Umwandlung angibt.

`fwscanf` führt jede Anweisung einzeln aus. Wenn eine Anweisung fehlschlägt, wie unten genauer erläutert, kehrt die Funktion zurück. Fehler werden als Eingabefehler bezeichnet, wenn Eingabezeichen fehlen, oder als Formatfehler, wenn Eingabezeichen nicht zu dem Format passen.

Eine Anweisung, die aus einem Zwischenraumzeichen besteht, wird so ausgeführt, dass die Eingabe bis zum ersten Langzeichen gelesen wird, das kein Zwischenraumzeichen ist (dieses Langzeichen selbst wird nicht gelesen) oder bis keine Langzeichen mehr gelesen werden können (EOF).

Eine Anweisung, die aus einem normalen Langzeichen besteht, wird ausgeführt, indem das nächste Langzeichen aus der Eingabe gelesen wird. Wenn dieses Langzeichen nicht mit dem vorgegebenen Langzeichen übereinstimmt, schlägt die Anweisung fehl und das unpassende und alle nachfolgenden Langzeichen werden nicht gelesen.

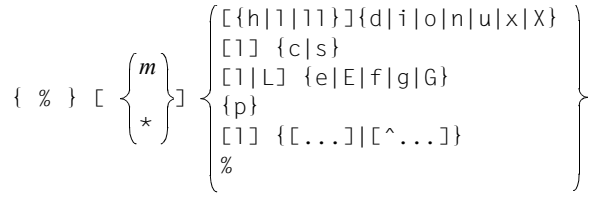
Eine Anweisung, die eine Umwandlungsanweisung ist, definiert eine Menge von passenden Eingabefolgen, wie dies unten für jede einzelne Umwandlungsanweisung beschrieben wird. Eine Umwandlungsanweisung wird in den folgenden Schritten ausgeführt:

Die Eingabe von Zwischenraumzeichen wird überlesen, solange die Anweisung weder ein `[` noch eines der Umwandlungszeichen `c` oder *n* enthält.

Eingabeelemente werden aus der Eingabe gelesen, solange die Anweisung nicht das Umwandlungszeichen *n* enthält. Ein Eingabeelement ist definiert als die längste Folge von Eingabezeichen (bis zu einer eventuell angegebenen maximalen Feldbreite), die ein Anfang einer passenden Folge ist. Das erste Langzeichen nach einem Eingabeelement bleibt, sofern es vorhanden ist, ungelesen.
Wenn die Länge des Eingabeelements gleich 0 ist, schlägt die Ausführung der Anweisung fehl; diese Bedingung bedeutet einen Formatfehler, sofern nicht ein Eingabefehler wie zum Beispiel EOF oder das Auftreten eines Lese-Fehlers weitere Eingaben verhindert.

Sofern nicht das Umwandlungszeichens `%` angegeben ist, wird das Eingabeelement (bzw. bei `%n` die Anzahl der gelesenen Eingabezeichen), umgewandelt in einen Datentyp, der dem Umwandlungszeichen entspricht. Wenn das Eingabeelement nicht zu der Umwandlungsanweisung passt, schlägt die Ausführung dieser Anweisung mit einem Formatfehler fehl.
Passt das Eingabeelement, wird - sofern die Zuweisung nicht durch das Zeichen * unterdrückt wird - das Ergebnis der Umwandlung in dem Objekt abgelegt, welches das erste auf *format* folgende Argument ist, in dem bisher noch kein Umwandlungsergebnis abgelegt wurde. Wenn dieses Objekt nicht den passenden Datentyp hat oder wenn das Ergebnis der Umwandlung nicht in dem zur Verfügung stehenden Platz dargestellt werden kann, ist das Verhalten undefiniert.

Umwandlungsanweisungen sehen also wie folgt aus:



Umwandlungszeichen

`d` Liest eine optional mit einem Vorzeichen versehene dezimale Ganzzahl ein, deren Format dasselbe ist, das die Funktion `wcstol` erwartet (*base* = 10). Das zugehörige Argument sollte ein Zeiger auf `int` sein.

`i` Liest eine optional mit einem Vorzeichen versehene dezimale Ganzzahl ein, deren Format dasselbe ist, das die Funktion `wcstol` erwartet (*base* = 8). Das entsprechende Argument sollte vom Typ Zeiger auf `int` sein.

`o` Liest eine optional mit einem Vorzeichen versehene oktale Ganzzahl ein, deren Format dasselbe ist, das die Funktion `wcstoul` erwartet (*base* = 8). Das entsprechende Argument sollte vom Typ Zeiger auf `unsigned integer` sein.

`u` Liest eine optional mit einem Vorzeichen versehene dezimale Ganzzahl ein, deren Format dasselbe ist, das die Funktion `wcstoul` erwartet (*base* = 10). Das entsprechende Argument sollte vom Typ Zeiger auf `unsigned integer` sein.

`x, X` Liest eine optional mit einem Vorzeichen versehene hexadezimale Ganzzahl ein, deren Format dasselbe ist, das die Funktion `wcstoul` erwartet (*base* = 16). Das entsprechende Argument sollte vom Typ Zeiger auf `unsigned integer` sein.

`e, E, f, g, G` Diese Umwandlungszeichen lesen eine optional mit einem Vorzeichen versehene Gleitpunktzahl ein. Deren Format ist dasselbe, das auch `wcstod` erwartet. Das entsprechende Argument sollte vom Typ Zeiger auf `float` sein.

`s` Liest eine Folge von Langzeichen ein, die keine Zwischenraumzeichen sind.
Ist kein `l` angegeben, werden Zeichen aus dem Eingabefeld so konvertiert wie bei Aufrufen der Funktion `wcrtomb`. Der Konversions-Status wird dabei in einem Objekt vom Typ `mbstate_t` beschrieben und mit 0 initialisiert, bevor das erste Langzeichen konvertiert wird. Es wird bis zum abschließenden Nullzeichen geschrieben.Das entsprechende Argument sollte ein Zeiger ein `char`-Feld sein, das groß genug ist, um die konvertierte Folge und ein abschließendes Nullzeichen aufzunehmen, das automatisch angefügt wird.
Ist `l` angegeben, sollte das entsprechende Argument ein Zeiger auf das erste Element eines `wchar_t`-Feldes sein, das groß genug ist, um die Folge und ein abschließendes Nullzeichen aufzunehmen, das automatisch angefügt wird.

`[` Liest eine nichtleere Folge von Langzeichen aus einer Menge von erwarteten Langzeichen (der Eingabemenge).
Ist kein `l` angegeben, werden Zeichen aus dem Eingabefeld so konvertiert wie bei Aufrufen der Funktion `wcrtomb`. Der Konversions-Status wird dabei in einem Objekt vom Typ `mbstate_t` beschrieben und mit 0 initialisiert, bevor das erste Langzeichen konvertiert wird. Es wird bis zum abschließenden Nullzeichen geschrieben.Das entsprechende Argument sollte ein Zeiger ein `char`-Feld sein, das groß genug ist, um die konvertierte Folge und ein abschließendes Nullzeichen aufzunehmen, das automatisch angefügt wird.
Ist `l` angegeben, sollte das entsprechende Argument ein Zeiger auf das erste Element eines `wchar_t`-Feldes sein, das groß genug ist, um die Folge und ein abschließendes Nullzeichen aufzunehmen, das automatisch angefügt wird.
Die Umwandlungsanweisung umfasst alle auf `[` folgenden Langzeichen in der Zeichenkette *format* bis einschließlich der zugehörigen schließenden eckigen Klammer `]`. Die Langzeichen zwischen den Klammern stellen die Eingabemenge dar, sofern nicht das erste Langzeichen nach der linken Klammer das Zeichen `^` ist. In diesem Fall enthält die Eingabemenge alle Langzeichen, die nicht in der Liste zwischen dem Zeichen `^` und der Klammer `]` aufgeführt sind.
Als Sonderfall gilt, dass die rechte eckige Klammer in den beiden Fällen, in denen die Umwandlungsanweisung mit den Zeichenketten `[]` bzw. `[^]` beginnt, zur Eingabemenge gehört und erst die nächste rechte eckige Klammer diejenige ist, welche die Umwandlungsanweisung abschließt.
Wenn das Zeichen `-` in der Liste auftritt und weder das letzte Zeichen noch das erste Zeichen nach `[` bzw. `[^` ist, dann ist das Verhalten undefiniert.

`c` Liest eine Folge von Langzeichen, deren Anzahl durch die Feldbreite bestimmt wird. Ist keine Feldbreite angegeben, wird 1 Langzeichen gelesen. Ist kein `l` angegeben, werden Zeichen aus dem Eingabefeld so konvertiert wie bei Aufrufen der Funktion `wcrtomb`. Der Konversions-Status wird dabei in einem Objekt vom Typ `mbstate_t` beschrieben und mit 0 initialisiert, bevor das erste Langzeichen konvertiert wird. Das entsprechende Argument sollte ein Zeiger auf ein `char`-Feld sein, das groß genug ist, um die konvertierte Folge aufzunehmen. Es wird kein Nullzeichen angefügt.
Ist `l` angegeben, sollte das entsprechende Argument ein Zeiger auf das erste Element eines `wchar_t`-Feldes sein, das groß genug ist, um die Folge aufzunehmen Es wird kein Nullzeichen angefügt.

Das Überlesen von Zwischenraumzeichen wird in diesem Fall unterdrückt; um das nächste Langzeichen zu lesen, das kein Zwischenraumzeichen ist, sollte `%1s` verwendet werden.

`p` Liest eine Menge von Folgen, die denen entsprechen sollten, die von der Umwandlungsanweisung `%p` der `fwprintf`-Funktionen erzeugt werden. Das entsprechende Argument sollte ein Zeiger auf einen Zeiger auf `void` sein. Die Interpretation des Eingabeelements ist jeweils implementierungsabhängig; für ein Eingabeelement, das nicht zuvor während derselben Programmausführung umgewandelt wurde, ist das Verhalten der Umwandlungsanweisung `%p` undefiniert. Dies gilt insbesondere für Zeigerausgaben, die von anderen Systemen erzeugt worden sind.

`n` Es wird keine Eingabe verarbeitet. Das entsprechende Argument sollte ein Zeiger auf `int` sein, in das die bisher von diesem Aufruf gelesene Zahl der Langzeichen eingetragen wird. Die Ausführung einer Anweisung des Typs *%n* erhöht nicht den Zuweisungszähler, der bei Beendigung der Ausführung der Funktion zurückgeliefert wird.

`%` Liest ein einzelnes `%`. Dabei findet keine Umwandlung oder Zuweisung statt. Die vollständige Umwandlungsanweisung lautet `%%`.

Wenn ein Umwandlungszeichen ungültig ist, ist das Verhalten von `fwscanf` undefiniert.

Wenn das Dateiende während der Eingabe gefunden wird, wird die Umwandlung abgebrochen. Wenn das Dateiende auftritt, bevor irgendwelche, zur aktuellen Anweisung passenden Langzeichen gelesen wurden (abgesehen von zulässigen Zwischenraumzeichen), wird die Ausführung der aktuellen Anweisung mit einem Eingabefehler abgebrochen. Andernfalls wird, falls die Bearbeitung der aktuellen Anweisung nicht mit einem Formatfehler abbricht, eine von `%n` verschiedene darauffolgende Anweisung mit einem Eingabefehler abgebrochen.

Wenn während eines `swscanf`-Aufrufs das Ende einer Zeichenkette erreicht wird, ist dies äquivalent zum Erreichen des Dateiendekennzeichens während eines `fwscanf`-Aufrufs.

Abschließende Zwischenraumzeichen (einschließlich der Zeilenendezeichen) bleiben ungelesen, sofern nicht eine entsprechende Umwandlungsanweisung vorhanden ist.

Der Erfolg des 1:1 Einlesens von Buchstaben und von unterdrückten Zuweisungen kann nicht direkt bestimmt werden, außer über die Anweisung *%n*.

Returnwert Anzahl der eingelesenen und erfolgreich zugewiesenen Eingabeelemente
falls nicht vor der ersten Zuweisung ein Eingabefehler auftritt.
Die Anzahl ist Null, wenn bereits beim ersten Eingabelement ein Formatfehler auftritt.

EOF falls vor der ersten Zuweisung ein Eingabefehler auftritt.

Hinweis In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Siehe auch scanf, sscanf, fscanf, wcstod, wcstol, wcstoul, wcrtomb

fwrite - Blockweise in eine Datei schreiben

Definition

```
#include <stdio.h>

size_t fwrite(const void *zg, size_t elgroesse, size_t anz, FILE *dz);
```

`fwrite` schreibt *anz* Elemente von jeweils *elgroesse* Bytes Größe aus dem Bereich, auf den *zg* zeigt, in die Datei mit Dateizeiger *dz*.
Die Position des Lese-/Schreibzeigers ist anschließend um die Anzahl der geschriebenen Bytes erhöht.

Returnwert

Anzahl der tatsächlich geschriebenen Elemente	bei Erfolg.
0	bei Dateiende oder Fehler.

Hinweise

Um sicherzugehen, dass *elgroesse* die richtige Anzahl Bytes für ein Datenelement angibt, sollten Sie die Funktion `sizeof` für die Größe einer Dateneinheit verwenden, auf die *zg* zeigt.

Bei der Ausgabe in Dateien mit Strom-E/A werden die Daten nicht sofort in die externe Datei geschrieben, sondern in einem C-internen Puffer zwischengespeichert (siehe Abschnitt „Pufferung“ auf Seite 65).

Bei der Ausgabe in Textdateien werden die Steuerzeichen für Zwischenraum (\n, \t, etc.) je nach Art der Textdatei in ihre entsprechende Wirkung umgesetzt (siehe Abschnitt „Zwischenraum“ auf Seite 68).

Satz-E/A

`fwrite` schreibt einen Satz in die Datei.

Bei sequenziellen Dateien (SAM, PAM) wird der Satz an die aktuelle Dateiposition geschrieben.
Bei indexsequenziellen Dateien (ISAM) wird der Satz an die Position geschrieben, die dem Schlüsselwert im Satz entspricht.

Anzahl der auszugebenden Zeichen:

Im Folgenden sei n die Gesamtanzahl der auszugebenden Zeichen, d.h.

$$n = \text{Elementlänge} * \text{Elementanzahl}$$

Ist n größer als die maximale Satzlänge, wird nur ein Satz mit maximaler Satzlänge geschrieben. Die restlichen Daten gehen verloren.

Ist n kleiner als die minimale Satzlänge, wird kein Satz geschrieben. Die minimale Satzlänge ist nur für ISAM-Dateien definiert und bedeutet, dass n mindestens den Bereich des Schlüssels im Satz umfassen muss.

Ist *n* beim Neuschreiben eines Satzes in eine Datei mit fester Satzlänge kleiner als die Satzlänge, wird der Satz am Ende mit binären Nullen aufgefüllt.

Beim Update eines bestehenden Satzes in einer sequenziellen Datei (SAM, PAM) muss *n* gleich der Länge des zu aktualisierenden Satzes sein. Im anderen Fall tritt ein Fehler auf. Als Satzlänge für PAM-Dateien gilt die Länge eines logischen Blocks.

Beim Update eines bestehenden Satzes in einer indexsequenziellen Datei (ISAM) braucht *n* nicht gleich der Länge des zu aktualisierenden Satzes sein. Ein Satz kann also verkürzt oder verlängert werden.

In ISAM-Dateien, für die Schlüsselverdoppelung zugelassen ist, ist kein direkter Update eines Satzes möglich. Beim Schreiben eines Satzes mit einem bereits existierenden Schlüssel wird stets ein neuer Satz geschrieben. Der alte Satz muss explizit gelöscht werden.

`fwrite` liefert den gleichen Returnwert wie bei Strom-E/A, nämlich die Anzahl der vollständig geschriebenen Elemente. Bei Satz-E/A ist es sinnvoll, ausschließlich die Elementlänge 1 zu verwenden, da in diesem Fall der Returnwert der Länge des geschriebenen Satzes entspricht (ohne ein ggf. vorhandenes Satzlängenfeld).
Bei fester Satzlänge wird jedoch das (ggf. notwendige) Auffüllen mit binären Nullen im Returnwert nicht berücksichtigt.

Beispiel Folgendes Programm überträgt zwei Personeneinträge auf die Datei mit Dateizeiger *p_list*.

```
#include <stdio.h>

int main(void)
{
  FILE *p_list;
  size_t result;
  static struct p
  {
    char name[20];
    int a;
  } person[2] =
    {
      {"ANNE", 30},
      {"HANS", 60},
    };

  p_list = fopen("link=link", "w");

  result = fwrite(person, sizeof(struct p), 2, p_list);
  printf("%d Personeneintraege geschrieben\n", result);
  return 0;
}
```

Siehe auch fread, feof, ferror

gamma - Logarithmische Gammafunktion

Definition

```
#include <math.h>

double gamma(double x);
```

`gamma` berechnet die mathematische Gammafunktion für Gleitkommazahlen x:

$$\int_0^\infty e^{-t} \quad t^{x-1} \quad dt$$

Das Vorzeichen dieses Wertes wird in der C-internen Variablen `signgam` als +1 oder -1 abgelegt. `signgam` darf nicht vom Anwender definiert werden.

Returnwert

`gamma(x)`	bei Erfolg.
HUGE_VAL	falls der korrekte Wert einen Überlauf ergibt. Zusätzlich wird `errno` auf ERANGE gesetzt (Resultat zu groß).
HUGE_VAL	falls x eine nicht-positive Ganzzahl ist. Zusätzlich wird `errno` auf EDOM gesetzt (unzulässiges Argument).

garbcoll - Speicherplatz an das System freigeben

Definition

```
#include <stdlib.h>

void garbcoll(void);
```

Die Funktionen `calloc`, `malloc`, `realloc` und `free` bilden das C-spezifische Speicherverwaltungspaket. Dieses Paket besteht im Wesentlichen aus einer internen Freispeicherverwaltung.
Der mit `free` freigegebene Speicher wird nicht an das System zurückgegeben (RELM-SVC), sondern durch die Freispeicherverwaltung erfasst.
Die Funktionen für Speicheranforderungen (`calloc`, `malloc`, `realloc`) versuchen, den Speicher zuerst über die Freispeicherverwaltung zu besorgen und erst in zweiter Linie vom Betriebssystem (REQM-SVC).

Falls auch vom System kein Speicher mehr erhältlich ist, wird der in der Freispeicherverwaltung erfasste Speicher so weit wie möglich seitenweise an das System zurückgegeben (Garbage Collection).

Dieser Garbage-Collection-Mechanismus wird im Adressraum ≤ 2 GB wirksam und ist mit der Funktion `garbcoll` auch explizit aufrufbar.

Hinweis Die Funktion `garbcoll` gibt alle Speicherbereiche an das System zurück, die zuvor mit `free` freigegeben wurden und sich zu freien Seiten zusammenstellen lassen.

Siehe auch calloc, malloc, realloc, free

gcvt - Gleitkommazahl in Zeichenkette umwandeln

Definition

```
#include <stdlib.h>

char *gcvt(double wert, int anz, char *puf);
```

`gcvt` wandelt einen Gleitkommawert *wert* in eine Zeichenkette aus Ziffern um und speichert die Zeichenkette in dem Bereich, auf den *puf* zeigt. Als Ergebnis wird ein Zeiger auf diesen Bereich geliefert.

Je nach Aufbau des umzuwandelnden Gleitkommawertes, entspricht das Ausgabeformat

- dem FORTRAN F-Format: *anz* signifikante Stellen, keine führenden und nachgestellten Nullen von *wert*, ein ggf. negatives Vorzeichen und Dezimalpunkt (sofern dem Dezimalpunkt Ziffern ungleich 0 folgen)
- oder dem FORTRAN E-Format (Exponential-Schreibweise).

Parameter

double wert
: Gleitkommawert, der für die Ausgabe aufbereitet werden soll.

int anz
: Anzahl der Ziffern in der Ergebniszeichenkette (gerechnet ab der ersten Ziffer ungleich 0 aus dem umzuwandelnden Gleitkommawert).

  Ist *anz* kleiner als die Ziffernzahl von *wert*, wird die niedrigste Stelle gerundet.
  Ist *anz* größer, endet die Zeichenkette mit der letzten Ziffer ungleich 0.

char *puf
: Zeiger auf die umgewandelte Zeichenkette.
  Der Speicherbereich, auf den *puf* zeigt, sollte mindestens (*anz* + 4) Bytes groß sein!

Returnwert Zeiger auf die umgewandelte Zeichenkette.
`gcvt` schließt die Zeichenkette mit dem Nullbyte (\0) ab.

Hinweise Falsche Parameter, etwa ein `integer`- statt `double`-Wert, führen zum Programmabbruch!

Sie müssen dafür sorgen, dass der Ergebniszeiger *puf* auf einen Speicherbereich von mindestens (*anz* + 4) Bytes zeigt (siehe auch Beispiel).

Beispiel Das Programm liest einen Gleitkommawert *x* ein, wandelt ihn nach der Angabe in *n* um und gibt ihn als Zeichenkette in den char-Vektor *buf* aus. Für die Reservierung von (*n* + 4) Bytes wird die Funktion `malloc` benutzt.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  double x;
  int n;
  char *buf;
  printf("Bitte Gleitkommazahl eingeben: \n");
  if ( scanf("%lf",&x) == 1)
  {
    printf("Wieviel signifikante Stellen : \n");
    if ( scanf("%d",&n) == 1)
    {
      buf = (char *)malloc(n + 4);
      printf("Die Zahl lautet umgewandelt :  %s \n", gcvt(x, n, buf));

    }
  }
  return 0;
}
```

Siehe auch ecvt, gcvt

getc - Zeichen aus einer Datei einlesen

Definition

```
#include <stdio.h>
int getc(FILE *dz);
```

`getc` liest ein Zeichen aus der Datei mit Dateizeiger *dz* von der aktuellen Lese-/Schreibposition.

Returnwert

Gelesenes Zeichen als positiver `integer`-Wert	bei Erfolg.
EOF	bei Fehler oder Dateiende.

Hinweise

`getc` ist sowohl als Makro als auch als Funktion realisiert (siehe Abschnitt „Funktionen und Makros“ auf Seite 19).

Der Aufruf `getc(stdin)` ist identisch mit `getchar`.

Wenn Sie in Ihrem Programm einen Vergleich verwenden, wie etwa
`while((c = getc(dz)) != EOF)`,
dann muss die Variable *c* immer als `integer`-Größe vereinbart werden. Wenn Sie *c* als `char` definieren, wird die Bedingung EOF aus folgendem Grund nie erfüllt: -1 wird nach `char '0xFF'` (also +255) konvertiert, EOF ist jedoch als -1 definiert.

Wenn `getc` von der Standardeingabe `stdin` liest und EOF das Einlese-Endekriterium ist, erreichen Sie die EOF-Bedingung durch folgende Maßnahmen an der Datensichtstation: Taste K2 drücken, die Systemkommandos EOF und RESUME-PROGRAM eingeben.

Beispiel Folgendes Programm liest aus einer Datei mit Dateizeiger *fp* zeichenweise bis Dateiende ein und speichert die Zeichen in einen Bereich *buf*.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int c, i = 0;
  char buf[BUFSIZ];
  FILE *fp;
  char name[40];
  printf("Bitte einzulesende Datei eingeben\n");
  scanf("%s", name);
  if(( fp = fopen(name, "r")) == NULL)
    {
    perror("fopen");  /* Abbruch bei nicht vorhandener Datei */
    exit(1);          /* mit Fehlermeldung 'fopen'           */
    }
  while (( c = getc(fp)) != EOF )
    buf[i++] = c;
  puts(buf);
  fclose(fp);
  return 0;
}
```

Siehe auch fgetc, getchar, getwc, ungetc, fopen, fopen64

getchar - Zeichen von Standardeingabe einlesen

Definition

```
#include <stdio.h>
int getchar(void);
```

`getchar` liest ein Zeichen von der Standardeingabe (Dateizeiger `stdin`).

Returnwert

Zeichen als positiver `integer`-Wert	bei Erfolg.
EOF	bei Fehler oder Dateiende.

Hinweise

`getchar` ist sowohl als Makro als auch als Funktion realisiert (siehe Abschnitt „Funktionen und Makros“ auf Seite 19).

Wenn Sie in Ihrem Programm einen Vergleich verwenden, wie etwa

```
while((c = getchar()) != EOF)
```

dann muss die Variable *c* immer als `integer`-Größe vereinbart werden. Wenn Sie *c* als `char` definieren, wird die Bedingung EOF aus folgendem Grund nie erfüllt: -1 wird nach `char 'OxFF'` (also +255) konvertiert, EOF ist jedoch als -1 definiert.

Das Endekriterium EOF beim Einlesen von der Datensichtstation (`stdin`) erreichen Sie durch folgende Maßnahmen: Taste K2 drücken, die Systemkommandos EOF und RESUME-PROGRAM eingeben.

Siehe auch getc, fgetc, getwchar

getenv - Wert einer Umgebungsvariablen ermitteln

Definition

```
#include <stdlib.h>

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

`getenv()` durchsucht die aktuelle Umgebung des Prozesses, d.h. den Zeichenkettenvektor, auf den `environ` zeigt, nach einer Zeichenkette der Form "*name*=*value*" und gibt einen Zeiger auf die Zeichenkette zurück, die den Wert *value* für den angegebenen Variablennamen *name* enthält.

Returnwert

Wert von *name*
: wenn eine entsprechende Zeichenkette vorhanden ist.

Nullzeiger
: wenn keine entsprechende Zeichenkette vorhanden ist, oder wenn die S-Variable `SYSPOSIX` nicht existiert.

Hinweise

Die Zeichenkette "*name*=*value*" darf nicht verändert werden. Sie kann jedoch von nachfolgenden `putenv`-Aufrufen überschrieben werden. Andere Bibliotheksfunktionen überschreiben die Zeichenkette nicht.

Beim Start eines Programms wird neben den Voreinstellungen für die Umgebung auch die S-Variable `SYSPOSIX` als Umgebungsdefinition ausgewertet (siehe Abschnitt „Umgebungsvariablen“ auf Seite 118).

Siehe auch

environ, putenv(), stdlib.h

getlogin - Benutzerkennung ermitteln

Definition

```
#include <stdlib.h>

char *getlogin(void);
```

`getlogin` liefert den Namen der Benutzerkennung (Userid), in der das aufrufende Programm abläuft.

Returnwert Zeiger auf den Namen der Benutzerkennung.

Hinweis `getlogin` schreibt sein Ergebnis in einen C-internen Datenbereich, der bei jedem Aufruf überschrieben wird!

Beispiel

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  printf("Beispiel fuer getlogin():\n");
  printf("Userid = %s\n", getlogin());
  return 0;
}
```

getpgmname - Programmnamen ermitteln

Definition

```
#include <stdlib.h>

char *getpgmname(void);
```

`getpgmname` liefert den Namen des aufrufenden Programmes. Das Ergebnis entspricht `argv[0]` der Funktion `main`.

Returnwert Zeiger auf den Programmnamen.

Beispiel

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  printf("Beispiel fuer getpgmname():");
  printf("Programmname = %s\n", getpgmname());
  return 0;
}
```

gets - Zeichenkette von Standardeingabe einlesen

Definition #include <stdio.h>

char *gets(char *s);

`gets` liest von der Standardeingabe `stdin` Zeichen bis zum nächsten Zeilenende und speichert die gelesene Zeichenkette in den Bereich, auf den *s* zeigt. Das Zeilenende wird ersetzt durch das Nullbyte (\0).

Returnwert Zeiger auf die Ergebniszeichenkette
bei Erfolg. `gets` schließt die Zeichenkette mit dem Nullbyte (\0) ab.

NULL-Zeiger wenn Dateiende erreicht wird oder ein Lesefehler auftritt.

Hinweise Den Bereich, in den gets die gelesene Zeichenkette abspeichern soll, müssen Sie explizit bereitstellen!

Im Unterschied zu `fgets` löscht gets ein gelesenes Neue-Zeile-Zeichen bzw. überschreibt es mit dem Nullbyte.

Das Endekriterium beim Einlesen von der Datensichtstation (`stdin`) erreichen Sie durch folgende Maßnahmen: Taste K2 drücken, die Systemkommandos EOF und RESUME-PROGRAM eingeben.

Beispiel Folgendes Programm liest Zeichenketten von der Standardeingabe ein und gibt sie auf die Standardausgabe wieder aus. Das Einlesen kann beendet werden mit der K2-Taste und den Kommandos EOF, RESUME-PROGRAM.

```
#include <stdio.h>
int main(void)
{
 char s[BUFSIZ];
 while(gets(s) != NULL) puts(s);
 return 0;
}
```

Siehe auch fgets, puts, fputs, getws

gettsn - TSN-Nummer ermitteln

Definition

```
#include <stdlib.h>

char *gettsn(void);
```

`gettsn` liefert die Task-Sequence-Number (TSN) des aufrufenden Programmes.

Returnwert Zeiger auf die Task-Sequence-Number (TSN).

Hinweis `gettsn` schreibt sein Ergebnis in einen C-internen Datenbereich, der bei jedem Aufruf überschrieben wird!

Beispiel

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  printf("Beispiel fuer gettsn():\n");
  printf("TSN-Nummer des Programms %s : %s\n", getpgmname(), gettsn());
  return 0;
}
```

getw - Wortweise aus einer Datei einlesen

Definition

```
#include <stdio.h>

int getw(FILE *dz);
```

`getw` liest ein Maschinenwort aus der Datei mit Dateizeiger *dz* und positioniert den Lese-/Schreibzeiger hinter das gelesene Wort.

Unter Maschinenwort ist ein binärer `Integer`-Wert zu verstehen.

Returnwert

Gelesenes Wort als `Integer`-Wert	bei Erfolg.
EOF	bei Dateiende oder Fehler.

Hinweise

Weil Wortlänge und Anordnung der Bytes systemabhängig sind, können unter Umständen Dateien, die mit `putw` auf einem Betriebssystem ungleich BS2000 beschrieben wurden, nicht mit `getw` im BS2000 gelesen werden.

Da EOF einen gültigen `Integer`-Wert darstellt, sollten Sie die Funktionen `feof` und `ferror` verwenden, um die Bedingungen Dateiende bzw. Fehler zu überprüfen.

Beispiel

Folgender Programmausschnitt liest aus der Datei mit Dateizeiger *fp* wortweise ein, bis Dateiende erreicht ist.

```
int buf[MAX];
int i = 0;
FILE *fp;

while(!feof(fp) && !ferror(fp))
     buf[i++] = getw(fp);
```

Siehe auch putw

getwc - Langzeichen aus Datei lesen

Definition

```
#include <wchar.h>
#include <stdio.h>

wint_t getwc(FILE *dz);
```

`getwc` entspricht `fgetwc` mit dem Unterschied, dass `getwc` als Makro *dz* mehrmals auswerten kann. Das Argument sollte also niemals ein Ausdruck mit Seiteneffekten sein.

Returnwert

Langzeichenwert vom Typ `wint_t`
bei erfolgreicher Beendigung.

WEOF — wenn das Dateiende erreicht ist. Das Dateiendekennzeichen für die Datei wird gesetzt.
Oder
wenn ein Lesefehler auftritt. Die Fehleranzeige für die Datei wird gesetzt.
`errno` wird auf EBADF gesetzt, wenn *dz* kein gültiger Dateizeiger ist. .

Hinweise

In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

`getwc` ist sowohl als Makro als auch als Funktion implementiert (siehe Abschnitt „Funktionen und Makros“ auf Seite 19).

Diese Schnittstelle wird zur Verfügung gestellt, um einige aktuelle Implementierungen und mögliche zukünftige ISO-Standards zu unterstützen.

Wenn `getwc` als Makro verwendet wird, kann ein Ausdruck *dz* mit Seiteneffekten inkorrekt behandelt werden. Inbesondere kann `getwc(*f++)` anders funktionieren als erwartet. Daher wird in solchen Situationen empfohlen, stattdessen `fgetwc` zu benutzen.

Das Endekriterium WEOF beim Einlesen von der Datensichtstation (`stdin`) erreichen Sie durch folgende Maßnahmen: Taste K2 drücken, die Systemkommandos EOF und RESUME-PROGRAM eingeben.

Siehe auch

fgetwc, getc

getwchar - Langzeichen aus Standardeingabe lesen

Definition

```
#include <wchar.h>

wint_t getwchar(void);
```

Der Funktionsaufruf `getwchar(void)` entspricht `getwc(stdin)`, d.h. es wird ein Langzeichen aus der Standardeingabe gelesen.

Das Endekriterium WEOF beim Einlesen von der Datensichtstation (`stdin`) erreichen Sie durch folgende Maßnahmen: Taste K2 drücken, die Systemkommandos EOF und RESUME-PROGRAM eingeben.

Returnwert

Langzeichenwert vom Typ `wint_t`
bei erfolgreicher Beendigung.

WEOF — wenn das Dateiende erreicht ist. Das Dateiendekennzeichen für die Datei wird gesetzt.
Oder
wenn ein Lesefehler auftritt. Die Fehleranzeige für die Datei wird gesetzt.
`errno` wird auf EBADF gesetzt, wenn *dz* kein gültiger Dateizeiger ist. .

Hinweis

In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Siehe auch

fgetwc, getwc

gmtime, gmtime64 - Datum und Uhrzeit in UTC umwandeln

Definition

```
#include <time.h>

struct tm *gmtime(const time_t *sek_zg);
struct tm *gmtime64(const time64_t *sek_zg);
```

`gmtime` und `gmtime64` interpretieren die Zeitangabe, auf die *sek_zg* zeigt, als Anzahl der seit dem Stichtag (Epoche) vergangenen Sekunden. Die Funktionen berechnen daraus Datum und Uhrzeit und speichern das Ergebnis im UTC-Format (Universal Time Coordinated) in einer Struktur vom Typ `tm`. Negative Werte werden als Sekunden vor dem Stichtag interpretiert. Das kleinste darstellbare Datum ist der 01.01.1900 00:00:00 lokale Zeit.

`gmtime` und `gmtime64` entsprechen den Funktionen `localtime und localtime64`, sie liefern jeweils die lokale Zeit.

Bei `gmtime` hängt der Stichtag von der Verwendung des TIMESHIFT-Bindeschalters ab (siehe Abschnitt „Zeitfunktionen“ auf Seite 41):
– ohne TIMESHIFT-Bindeschalter (Standard): 1.1.1950 00:00:00.
– mit TIMESHIFT-Bindeschalter: 1.1.1970 00:00:00.
Bei `gmtime64` ist der Stichtag immer der 1.1.1970 00:00:00.

Das größte darstellbare Datum ist bei `gmtime` der 19.01.2018 03:14:07 (ohne TIMESHIFT-Bindeschalter) bzw. der 19.01.2038 03:14:07 (mit TIMESHIFT-Bindeschalter).

`gmtime64` kann unabhängig von der Verwendung des TIMESHIFT-Bindeschalters Daten bis zum 18.3.4317 02:44:48 darstellen.

Returnwert Zeiger auf die berechnete Struktur. `gmtime` und `gmtime64` legen das Ergebnis in einer Struktur ab, die wie folgt in <time.h> deklariert ist:

```
struct tm
{
  int   tm_sec;         /* Sekunden (0-59) */
  int   tm_min;         /* Minuten (0-59) */
  int   tm_hour;        /* Stunden (0-23) */
  int   tm_mday;        /* Tag des Monats (1-31) */
  int   tm_mon;         /* Monate ab Jahresbeginn (0-11) */
  int   tm_year;        /* Jahre seit 1900 */
  int   tm_wday;        /* Wochentag (0-6, Sonntag=0) */
  int   tm_yday;        /* Tage seit dem 1. Januar (0-365) */
  int   tm_isdst;       /* Sommerzeitanzeige  */
};
```

NULL im Fehlerfall

Hinweise Die Funktionen `asctime`, `ctime`, `ctime64`, `gmtime`, `gmtime64`, `localtime` und `localtime64` schreiben ihre Ergebnisse in denselben C-internen Datenbereich, so dass der Aufruf einer dieser Funktionen das vorherige Ergebnis einer der anderen Funktionen überschreibt.

Beispiel

```
#include <time.h>
#include <stdio.h>

struct tm *t;
char *s;
time_t clk;

int main(void)
{
  clk = time((time_t *)0);
  t = gmtime(&clk);
  printf("Jahr: %d\n", t->tm_year + 1900);
  printf("Uhrzeit in Stunden: %d\n", t->tm_hour);
  printf("Jahrestag: %d\n", t->tm_yday);
  s = asctime(t);
  printf("%s", s);
  return 0;
}
```

Siehe auch asctime, ctime, ctime64, localtime, localtime64

hypot - Euklidischer Abstand

Definition #include <math.h>

double hypot(double x, double y);

`hypot` berechnet den euklidischen Abstand für den Punkt mit den Koordinaten (x,y).

Returnwert `sqrt(x*x + y*y)`
bei Erfolg, d.i. Wurzel aus der Summe der quadrierten Koordinaten.

HUGE_VAL bei Überlauf. Zusätzlich wird `errno` auf ERANGE gesetzt (Resultat zu groß).

Beispiel

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main(void)
{
  double x, y, alpha, r, pi;

  printf("Koordinaten x und y eingeben:\n");
  scanf("%lf %lf", &x, &y);

  pi = 2.0 * asin(1.0);

  if(x > 0.0)
    alpha = atan(y/x);
  else if (x < 0.0)
          if (y >= 0.0)
             alpha = atan(y/x) + pi;
          else alpha = atan(y/x) - pi;
       else if (y > 0)
               alpha = pi/2.0;
            else if (y < 0)
                    alpha = -pi/2.0;
                 else
                 {
                   printf("Winkel nicht definiert!\n");
                   exit(1);
                 }
  r = hypot(x, y);
  alpha = alpha * (180.0/pi);
```

```
  printf("Die Polarkoordinaten lauten:\n");
  printf("Abstand vom Nullpunkt: %g\n",r);
  printf("Winkel zur x-Achse:\n");
  printf("%g Grad\n",((y < 0.0)? alpha + 360 : alpha) );
  return 0;
}
```

Siehe auch cabs, sqrt

ieee2double - Gleitpunktzahl vom IEEE-Format in das /390-Format konvertieren

Definition #include <ieee_390.h>

extern double ieee2double (double num);

`ieee2double` konvertiert eine 8-byte-Gleitpunktzahl *num* des IEEE-Formats in das /390-Format und liefert sie als Ergebnis zurück. Dabei geht keine Genauigkeit verloren.

Parameter double num
8-byte-Gleitpunktzahl im IEEE-Format

Returnwert 8-byte-Gleitpunktzahl im /390-Format (bei Erfolg)

`0.0` falls die IEEE-Gleitpunktzahl betragsmäßig kleiner ist als die kleinste darstellbare Zahl des /390-Formats oder falls `NaN` oder `inf` als Parameter übergeben wird.

Falls die IEEE-Gleitpunktzahl betragsmäßig größer ist als die größte darstellbare Zahl des /390-Formats, wird diese größte darstellbare Zahl mit dem entsprechenden Vorzeichen zurückgeliefert.

Die globale Variable *float_exceptions_flag* enthält Informationen für den Fall einer nicht ordnungsgemäßen Konvertierung und ist wie wie folgt definiert:

```
extern int float_exception_flags;
enum {
   float_flag_inexact   =  1,
   float_flag_divbyzero =  2,
   float_flag_underflow =  4,
   float_flag_overflow  =  8,
   float_flag_invalid   = 16
};
```

Falls die IEEE-Gleitpunktzahl betragsmäßig größer ist als die größte darstellbare Zahl des /390-Formats, wird *float_flag_overflow* gesetzt.

Falls die IEEE-Gleitpunktzahl betragsmäßig kleiner ist als die kleinste darstellbare Zahl des /390-Formats, wird *float_flag_underflow* gesetzt.

Falls `NaN` oder `inf` als Parameter übergeben wird, wird *float_flag_invalid* gesetzt.

Siehe auch float2ieee, float2ieee, double2ieee

ieee2float - Gleitpunktzahl vom IEEE-Format in das /390-Format konvertieren

Definition #include <ieee_390.h>

extern float ieee2float (float num);

`ieee2float` konvertiert eine 4-byte-Gleitpunktzahl *num* des IEEE-Formats in das /390-Format und liefert sie als Ergebnis zurück. Dabei kann weder Overflow noch Underflow auftreten, es können aber bis zu drei Bit-Stellen verloren gehen.

Parameter float num
4-byte-Gleitpunktzahl im IEEE-Format

Returnwert 4-byte-Gleitpunktzahl im /390-Format.

Die globale Variable *float_exceptions_flag* enthält Informationen für den Fall einer nicht ordnungsgemäßen Konvertierung und ist wie wie folgt definiert:

```
extern int float_exception_flags;
enum {
   float_flag_inexact   =  1,
   float_flag_divbyzero =  2,
   float_flag_underflow =  4,
   float_flag_overflow  =  8,
   float_flag_invalid   = 16
};
```

Falls bei der Konvertierung Bit-Stellen verloren gehen und das Ergebnis dadurch ungenau wird, wird *float_flag_invalid* gesetzt.

Siehe auch float2ieee, double2ieee, ieee2double

index - Erstes Vorkommen eines Zeichens in einer Zeichenkette

Definition #include <string.h>

char *index(const char *s, int c);

`index` sucht das erste Vorkommen des Zeichens *c* in der Zeichenkette *s* und liefert bei Erfolg einen Zeiger auf die gesuchte Position in *s*.

Das abschließende Nullbyte (\0) der Zeichenkette wird als Zeichen mitberücksichtigt.

Returnwert Zeiger auf die Position von *c* in der Zeichenkette *s*
bei Erfolg.

NULL-Zeiger wenn *c* in der Zeichenkette *s* nicht enthalten ist.

Hinweis Die Funktionen `index` und `strchr` sind äquivalent.

Beispiel Finde das erste 'k':

```
#include <string.h>
#include <stdio.h>

int main(void)
{
   char *s = "was fuer ein Spass im kkuehlen Nass!";
   printf("%s\n", s);
   printf("Wo steckt der Fehler? %s\n", index(s, 'k'));
   return 0;
}
```

Siehe auch rindex, strchr

isalnum - Buchstabe oder Ziffer?

Definition

```
#include <ctype.h>

int isalnum(int c);
```

`isalnum` überprüft, ob das Zeichen *c* alphanumerisch ist, d.h. ein Buchstabe (A-Z, a-z) oder eine Ziffer (0-9).

Returnwert

≠ 0	alphanumerisch.
0	nicht alphanumerisch.

Hinweis `isalnum` ist sowohl als Makro als auch als Funktion realisiert (siehe Abschnitt „Funktionen und Makros“ auf Seite 19).

Beispiel

```
#include <ctype.h>
#include <stdio.h>

int main(void)
{
  int c;
  while((c = getchar()) != EOF)
    printf("%s : %c\n", ((isalnum(c)) ? "alphanumerisch" : "Sonstiges"), c);
  return 0;
}
```

Siehe auch isalpha, isascii, iscntrl, isdigit, isgraph, islower, ispunct, isprint, isspace, isupper, isxdigit, isebcdic, iswalnum

isalpha - Buchstabe?

Definition
```
#include <ctype.h>
int isalpha(int c);
```

`isalpha` überprüft, ob das Zeichen c ein Buchstabe (A-Z, a-z) ist.

Returnwert

$\neq 0$	Buchstabe.
0	kein Buchstabe.

Hinweis `isalpha` ist sowohl als Makro als auch als Funktion realisiert (siehe Abschnitt „Funktionen und Makros“ auf Seite 19).

Beispiel
```
#include <ctype.h>
#include <stdio.h>

int main(void)
{
  int c;
  while((c = getchar()) != EOF)
     printf("%s : %c\n", ((isalpha(c)) ? "Buchstabe" : "Sonstiges"), c);
  return 0;
}
```

Siehe auch isalnum, isascii, iscntrl, isdigit, isgraph, islower, ispunct, isprint, isspace, isupper, isxdigit, isebcdic, iswalpha

isascii - EBCDIC-Zeichen?

Definition

```
#include <ctype.h>

int isascii(int c);
```

`isascii` ist ein Synonym für `isebcdic`. Auf EBCDIC-Rechnern prüft `isascii`, ob der Wert des Zeichens *c* ein EBCDIC-Zeichen repräsentiert (Werte 0 - 255). Ist Portabilität zu ASCII-Rechnern erforderlich, sollte `isascii` verwendet werden.

Returnwert

≠ 0	der Wert von *c* repräsentiert ein EBCDIC-Zeichen (Werte 0 - 255).
0	kein EBCDIC-Zeichen (Werte ≠ 0 - 255).

Siehe auch isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit, isebcdic

iscntrl - Kontrollzeichen?

Definition

```
#include <ctype.h>

int iscntrl(int c);
```

`iscntrl` überprüft, ob das Zeichen *c* ein Kontrollzeichen ist. Kontrollzeichen sind nicht abdruckbare Zeichen, z.B. für Druckersteuerung. Ausgenommen davon sind die nicht abdruckbaren Zeichen für „Zwischenraum“ (siehe bei `isspace`).

Returnwert

≠ 0	Kontrollzeichen.
0	kein Kontrollzeichen.

Hinweis `iscntrl` ist sowohl als Makro als auch als Funktion realisiert (siehe Abschnitt „Funktionen und Makros“ auf Seite 19).

Beispiel

```
#include <ctype.h>
#include <stdio.h>

int main(void)
{
  int c;
  while((c = getchar()) != EOF)
     printf("%s : %c\n", ((iscntrl(c)) ? "Kontrollzeichen":"Sonstiges"), c);
  return 0;
}
```

Siehe auch isalnum, isascii, isalpha, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit, isebcdic, iswcntrl

isdigit - Ziffer?

Definition

```
#include <ctype.h>
int isdigit(int c);
```

`isdigit` überprüft, ob das Zeichen c eine Ziffer (0-9) ist.

Returnwert

≠ 0	Ziffer.
0	keine Ziffer.

Hinweis `isdigit` ist sowohl als Makro als auch als Funktion realisiert (siehe Abschnitt „Funktionen und Makros“ auf Seite 19).

Beispiel

```
#include <ctype.h>
#include <stdio.h>

int main(void)
{
  int c;
  while((c = getchar()) != EOF)
     printf("%s : %c\n", ((isdigit(c)) ? "Ziffer" : "Sonstiges"), c);
  return 0;
}
```

Siehe auch isalnum, isascii, iscntrl, isalpha, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit, isebcic, iswdigit

isebcdic - EBCDIC-Zeichen?

Definition

```
#include <ctype.h>
int isebcdic(int c);
```

`isebcdic` prüft, ob der Wert des Zeichens *c* ein EBCDIC-Zeichen repräsentiert (Werte 0 - 255).

Returnwert

≠ 0	der Wert von *c* repräsentiert ein EBCDIC-Zeichen (Werte 0 - 255).
0	kein EBCDIC-Zeichen (Werte ≠ 0 - 255).

Hinweise

`isebcdic` ist sowohl als Makro als auch als Funktion realisiert (siehe Abschnitt „Funktionen und Makros“ auf Seite 19).

`isebcdic` ist ein Synonym für `isascii`. Ist Portabilität zu ASCII-Rechnern erforderlich, sollten Sie statt `isebcdic` `isascii` verwenden.

Beispiel

```
#include <ctype.h>
#include <stdio.h>

int main(void)
{
  int c;
  while((c = getchar()) != EOF)
    printf("%s : %c\n", ((isebcdic(c)) ? "EBCDIC-Zeichen" : "Sonstiges"), c);
  return 0;
}
```

Siehe auch isalpha, isalnum, isascii, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit

isgraph - Abdruckbares Zeichen exklusive Leerzeichen?

Definition

#include <ctype.h>

int isgraph(int c);

`isgraph` überprüft, ob das Zeichen *c* ein abdruckbares Zeichen ist, d.h. ein alphanumerisches oder ein Sonderzeichen. Als nicht abdruckbar gelten auch Leerzeichen.
c ist der Wert des zu überprüfenden Zeichens.

Returnwert

≠ 0	abdruckbar und kein Leerzeichen.
0	nicht abdruckbar oder Leerzeichen.

Hinweis

`isgraph` ist sowohl als Makro als auch als Funktion realisiert (siehe Abschnitt „Funktionen und Makros“ auf Seite 19).

Beispiel

```
#include <ctype.h>
#include <stdio.h>

int main(void)
{
  int c;
  while((c = getchar()) != EOF)
     printf("%s : %c\n",((isgraph(c))? "Zeichen" : "kann nicht drucken"), c);
  return 0;
}
```

Siehe auch

isalnum, isascii, iscntrl, isdigit, islower, isalpha, ispunct, isprint, isspace, isupper, isxdigit, isebcdic, iswgraph

islower - Kleinbuchstabe?

Definition

```
#include <ctype.h>
int islower(c);
```

`islower` überprüft, ob das Zeichen *c* ein Kleinbuchstabe (a-z) ist.
c ist der Wert des zu überprüfenden Zeichens.

Returnwert

≠ 0	Kleinbuchstabe.
0	kein Kleinbuchstabe.

Hinweis `islower` ist sowohl als Makro als auch als Funktion realisiert (siehe Abschnitt „Funktionen und Makros“ auf Seite 19).

Beispiel

```
#include <ctype.h>
#include <stdio.h>

int main(void)
{
  int c;
  while((c = getchar()) != EOF)
     printf("%s : %c\n", ((islower(c)) ? "Kleinbuchstabe" : "Sonstiges"), c);
  return 0;
}
```

Siehe auch isalnum, isascii, iscntrl, isdigit, isgraph, isalpha, isprint, ispunct, isspace, isupper, isxdigit, isebcdic, iswlower

isprint - Abdruckbares Zeichen inklusive Leerzeichen?

Definition

```
#include <ctype.h>

int isprint(int c);
```

`isprint` überprüft, ob das Zeichen *c* ein abdruckbares Zeichen ist, d.h. ein alphanumerisches Zeichen, ein Sonderzeichen oder ein Leerzeichen.

Returnwert

≠ 0	abdruckbar (inklusive dem Leerzeichen).
0	nicht abdruckbar.

Beispiel

```
#include <ctype.h>
#include <stdio.h>

int main(void)
{
  int c;
  while((c = getchar()) != EOF)
     printf("%s : %c\n",((isprint(c))? "Zeichen" : "kann nicht drucken"), c);
  return 0;
}
```

Siehe auch isalnum, isascii, iscntrl, isdigit, isgraph, islower, isalpha, ispunct, isspace, isupper, isxdigit, isebcdic, iswprint

ispunct - Sonderzeichen?

Definition

```
#include <ctype.h>
int ispunct(c);
```

`ispunct` überprüft, ob das Zeichen *c* ein Sonderzeichen ist, d.h. weder ein Kontrollzeichen, alphanumerisches Zeichen oder Zeichen für „Zwischenraum“ (siehe `isspace`).

Returnwert

≠ 0	Sonderzeichen.
0	kein Sonderzeichen.

Hinweis `ispunct` ist sowohl als Makro als auch als Funktion realisiert (siehe Abschnitt „Funktionen und Makros“ auf Seite 19).

Beispiel

```
#include <ctype.h>
#include <stdio.h>

int main(void)
{
  int c;
  while((c = getchar()) != EOF)
     printf("%s : %c\n",((ispunct(c))? "Sonderzeichen" : "Sonstiges"),c);
  return 0;
}
```

Siehe auch isalnum, isascii, iscntrl, isdigit, isgraph, islower, isalpha, isprint, isspace, isupper, isxdigit, iswpunct

isspace - Zwischenraum?

Definition

```
#include <ctype.h>

int isspace(int c);
```

`isspace` überprüft, ob das Zeichen c ein Zwischenraum ist, d.h. ein Leerzeichen, horizontaler Tabulator (\t), Wagenrücklauf (\r), Zeilenvorschub (\n), Seitenvorschub (\f) oder vertikaler Tabulator (\v).

Returnwert

$\neq 0$	Zwischenraum.
0	kein Zwischenraum.

Hinweise

`isspace` ist sowohl als Makro als auch als Funktion realisiert (siehe Abschnitt „Funktionen und Makros“ auf Seite 19).

Zur Auswertung der Steuerzeichen für Zwischenraum siehe Abschnitt „Zwischenraum“ auf Seite 68.

Beispiel

```
#include <ctype.h>
#include <stdio.h>

int main(void)
{
  int c;
  while((c = getchar()) != EOF)
     printf("%s : %c\n",((isspace(c))? "Zwischenraum " : "Sonstiges"),c);
  return 0;
}
```

Siehe auch

isalnum, isalpha, isascii, iscntrl, isdigit, islower, isprint, isgraph, ispunct, isupper, isxdigit, isebcdic, iswspace

isupper - Großbuchstabe?

Definition
```
#include <ctype.h>
int isupper(int c);
```

`isupper` überprüft, ob das Zeichen c ein Großbuchstabe (A-Z) ist.

Returnwert

≠ 0	Großbuchstabe.
0	kein Großbuchstabe.

Hinweis `isupper` ist sowohl als Makro als auch als Funktion realisiert (siehe Abschnitt „Funktionen und Makros“ auf Seite 19).

Beispiel
```
#include <ctype.h>
#include <stdio.h>

int main(void)
{
  int c;
  while((c = getchar()) != EOF)
     printf("%s : %c\n",((isupper(c))? "Großbuchstabe " : "Sonstiges"),c);
  return 0;
}
```

Siehe auch salnum, isascii, iscntrl, isdigit, islower, isprint, ispunct, isgraph, isspace, isalpha, isxdigit, isebcdic, iswupper

iswalnum - auf alphanumerisches Langzeichen prüfen

Definition

```
#include <wctype.h>

int iswalnum(wint_t wc);
```

`iswalnum` überprüft, ob das Langzeichen *wc* alphanumerisch ist.

In allen Fällen ist das Argument *wc* vom Typ *wint_t*. Der Wert von *wc* muss ein Langzeichenwert sein, der einem gültigen Zeichen in der aktuellen Lokalität entspricht, oder er muss gleich dem Wert des Makros WEOF sein. Wenn *wc* irgendeinen anderen Wert besitzt, ist das Verhalten undefiniert.

Returnwert

$\neq 0$	*wc* ist alphanumerisch
0	*wc* ist nicht alphanumerisch

Hinweise

In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

`iswalnum` ist sowohl als Makro als auch als Funktion realisiert (siehe Abschnitt „Funktionen und Makros“ auf Seite 19).

Das Verhalten von `iswalnum` wird von den Klassen `alpha` und `digit` der aktuellen Lokalität bestimmt. Die aktuelle Lokalität ist die C-Lokalität, wenn nicht explizit mit `setlocale` umgeschaltet wurde.

Siehe auch

isalnum, iswalpha, iswcntrl, iswdigit, iswgraph, iswlower, iswprint, iswpunct, iswspace, iswupper, iswxdigit, setlocale

iswalpha - auf alphabetisches Langzeichen prüfen

Definition

```
#include <wctype.h>

int iswalpha(wint_t wc);
```

`iswalpha` überprüft, ob das Langzeichen *wc* ein Buchstabe ist.

In allen Fällen ist das Argument *wc* vom Typ *wint_t*. Der Wert von *wc* muss ein Langzeichenwert sein, der einem gültigen Zeichen in der aktuellen Lokalität entspricht, oder er muss gleich dem Wert des Makros WEOF sein. Wenn *wc* irgendeinen anderen Wert besitzt, ist das Verhalten undefiniert.

Returnwert

$\neq 0$	*wc* ist ein Buchstabe
0	*wc* ist kein Buchstabe

Hinweise

In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

`iswalpha` ist sowohl als Makro als auch als Funktion realisiert (siehe Abschnitt „Funktionen und Makros“ auf Seite 19).

Das Verhalten von `iswalpha` wird von der Klasse `alpha` der aktuellen Lokalität bestimmt. Die aktuelle Lokalität ist die C-Lokalität, wenn nicht explizit mit `setlocale` umgeschaltet wurde.

Siehe auch

isalpha, iswalnum, iswcntrl, iswdigit, iswgraph, iswlower, iswprint, iswpunct, iswspace, iswupper, iswxdigit, setlocale

iswcntrl - auf Kontrollzeichen prüfen

Definition

```
#include <wctype.h>

int iswcntrl(wint_t wc);
```

`iswcntrl` überprüft, ob das Langzeichen *wc* ein Kontrollzeichen (Steuerzeichen) ist. Kontrollzeichen sind nicht abdruckbare Zeichen, z. B. für die Druckersteuerung.

In allen Fällen ist das Argument *wc* vom Typ `wint_t`. Der Wert von *wc* muss ein Langzeichenwert sein, der einem gültigen Zeichen in der aktuellen Lokalität entspricht, oder er muss gleich dem Wert des Makros WEOF sein. Wenn *wc* irgendeinen anderen Wert besitzt, ist das Verhalten undefiniert.

Returnwert

≠ 0	*wc* ist Kontrolllangzeichen
0	*wc* ist kein Kontrolllangzeichen

Hinweise

In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

`iswcntrl` ist sowohl als Makro als auch als Funktion realisiert (siehe Abschnitt „Funktionen und Makros“ auf Seite 19).

Das Verhalten von `iswcntrl` wird von der Klasse `cntrl` der aktuellen Lokalität bestimmt. Die aktuelle Lokalität ist die C-Lokalität, wenn nicht explizit mit `setlocale` umgeschaltet wurde.

Siehe auch

iscntrl, iswalnum, iswalpha, iswdigit, iswgraph, iswlower, iswprint, iswpunct, iswspace, iswupper, iswxdigit, setlocale

iswctype - Langzeichen auf Klasse prüfen

Definition

```
#include <wctype.h>

int iswctype(wint_t wc, wctype_t charclass);
```

`iswctype` überprüft, ob das Langzeichen *wc* zur Zeichenklasse *charclass* gehört.
In allen Fällen ist das Argument *wc* vom Typ `wint_t`. Der Wert von *wc* muss ein Langzeichenwert sein, der einem gültigen Zeichen in der aktuellen Lokalität entspricht, oder er muss gleich dem Wert des Makros WEOF sein. Wenn *wc* irgendeinen anderen Wert besitzt, ist das Verhalten undefiniert.

Returnwert

≠ 0	Langzeichen gehört zu Zeichenklasse *charclass*
0	Langzeichen gehört nicht zu Zeichenklasse *charclass*

Hinweise

In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Die zwölf Zeichenketten `"alnum"`, `"alpha"`, `"blank"`, `"cntrl"`, `"digit"`, `"graph"`, `"lower"`, `"print"`, `"punct"`, `"space"`, `"upper"` und `"xdigit"` sind für die Standard-Zeichenklassen reserviert. In der folgenden Tabelle sind die Funktionen der linken Spalte mit denen der rechten Spalte jeweils gleichwertig.

```
iswalnum(wc)      iswcty-     wctype("alnum"))
                  pe(wc,
iswalpha(wc)      iswcty-     wctype("alpha"))
                  pe(wc,
iswcntrl(wc)      iswcty-     wctype("cntrl"))
                  pe(wc,
iswdigit(wc)      iswcty-     wctype("digit"))
                  pe(wc,
iswgraph(wc)      iswcty-     wctype("graph"))
                  pe(wc,
iswlower(wc)      iswcty-     wctype("lower"))
                  pe(wc,
iswprint(wc)      iswcty-     wctype("print"))
                  pe(wc,
iswpunct(wc)      iswcty-     wctype("punct"))
                  pe(wc,
iswspace(wc)      iswcty-     wctype("space"))
                  pe(wc,
```

`iswupper(wc)`	`iswcty-pe(wc,`	`wctype("upper"))`
`iswxdigit(wc)`	`iswcty-pe(wc,`	`wctype("xdigit"))`

Der Aufruf `iswctype(wc,wctype("blank"))` hat keine gleichwertige `isw*`–Funktion.

Siehe auch wctype, iswalnum, iswalpha, iswcntrl, iswdigit, iswgraph, iswlower, iswprint, iswpunct, iswspace, iswupper, iswxdigit

iswdigit - auf dezimales Langzeichen prüfen

Definition

```
#include <wctype.h>

int iswdigit(wint_t wc);
```

`iswdigit` überprüft, ob das Langzeichen *wc* eine Dezimalziffer ist.

In allen Fällen ist das Argument *wc* vom Typ `wint_t`. Der Wert von *wc* muss ein Langzeichenwert sein, der einem gültigen Zeichen in der aktuellen Lokalität entspricht, oder er muss gleich dem Wert des Makros WEOF sein. Wenn *wc* irgendeinen anderen Wert besitzt, ist das Verhalten undefiniert.

Returnwert

≠ 0	*wc* ist Dezimalziffer
0	*wc* ist keine Dezimalziffer

Hinweise

In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

`iswdigit` ist sowohl als Makro als auch als Funktion realisiert (siehe Abschnitt „Funktionen und Makros“ auf Seite 19).

Das Verhalten von `iswdigit` wird von der Klasse `digit` der aktuellen Lokalität bestimmt. Die aktuelle Lokalität ist die C-Lokalität, wenn nicht explizit mit `setlocale` umgeschaltet wurde.

Siehe auch

isdigit, iswalnum, iswalpha, iswcntrl, iswgraph, iswlower, iswprint, iswpunct, iswspace, iswupper, iswxdigit

iswgraph - auf darstellbares Langzeichen prüfen

Definition

```
#include <wctype.h>

int iswgraph(wint_t wc);
```

`iswgraph` überprüft, ob das mit *wc* angegebene Langzeichen ein darstellbares Zeichen ist. Darstellbare Zeichen sind: alphanumerische Zeichen und Sonderzeichen. Leerzeichen gelten als nicht darstellbar.
In allen Fällen ist das Argument *wc* vom Typ `wint_t`. Der Wert von *wc* muss ein Langzeichenwert sein, der einem gültigen Zeichen in der aktuellen Lokalität entspricht, oder er muss gleich dem Wert des Makros WEOF sein. Wenn *wc* irgendeinen anderen Wert besitzt, ist das Verhalten undefiniert.

Returnwert

$\neq 0$	*wc* ist darstellbar
0	*wc* ist nicht darstellbar

Hinweise

In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

`iswgraph` ist sowohl als Makro als auch als Funktion realisiert (siehe Abschnitt „Funktionen und Makros“ auf Seite 19).

Das Verhalten von `iswgraph` wird von der Klasse `graph` der aktuellen Lokalität bestimmt. Die aktuelle Lokalität ist die C-Lokalität, wenn nicht explizit mit `setlocale` umgeschaltet wurde.

Siehe auch

isgraph, iswalnum, iswalpha, iswcntrl, iswdigit, iswlower, iswprint, iswpunct, iswspace, iswupper, iswxdigit, setlocale

iswlower - auf Kleinbuchstaben-Langzeichen prüfen

Definition

```
#include <wctype.h>

int iswlower(wint_t wc);
```

`iswlower` überprüft, ob das Langzeichen *wc* ein Kleinbuchstabe ist.

In allen Fällen ist das Argument *wc* vom Typ `wint_t`. Der Wert von *wc* muss ein Langzeichenwert sein, der einem gültigen Zeichen in der aktuellen Lokalität entspricht, oder er muss gleich dem Wert des Makros WEOF sein. Wenn *wc* irgendeinen anderen Wert besitzt, ist das Verhalten undefiniert.

Returnwert

$\neq 0$	*wc* ist Kleinbuchstabe
0	*wc* ist kein Kleinbuchstabe

Hinweise

In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

`iswlower` ist sowohl als Makro als auch als Funktion realisiert (siehe Abschnitt „Funktionen und Makros“ auf Seite 19).

Das Verhalten von `iswlower` wird von der Klasse `lower` der aktuellen Lokalität bestimmt. Die aktuelle Lokalität ist die C-Lokalität, wenn nicht explizit mit `setlocale` umgeschaltet wurde.

Siehe auch

islower, iswalnum, iswalpha, iswcntrl, iswdigit, iswgraph, iswprint, iswpunct, iswspace, iswupper, iswxdigit, setlocale

iswprint - auf druckbares Langzeichen prüfen

Definition

```
#include <wctype.h>

int iswprint(wint_t wc);
```

`iswprint` überprüft, ob *wc* ein druckbares Langzeichen ist. Druckbare Langzeichen sind: alphanumerische Zeichen, Sonderzeichen und Leerzeichen .

In allen Fällen ist das Argument *wc* vom Typ `wint_t`. Der Wert von *wc* muss ein Langzeichenwert sein, der einem gültigen Zeichen in der aktuellen Lokalität entspricht, oder er muss gleich dem Wert des Makros WEOF sein. Wenn *wc* irgendeinen anderen Wert besitzt, ist das Verhalten undefiniert.

Returnwert

$\neq 0$	*wc* ist druckbar (alphanum. Zeichen, Sonderzeichen oder Leerzeichen).
0	*wc* ist nicht druckbar

Hinweise

In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

`iswprint` ist sowohl als Makro als auch als Funktion realisiert (siehe Abschnitt „Funktionen und Makros“ auf Seite 19).

Das Verhalten von `iswprint` wird von der Klasse `print` der aktuellen Lokalität bestimmt. Die aktuelle Lokalität ist die C-Lokalität, wenn nicht explizit mit `setlocale` umgeschaltet wurde.

Siehe auch

isprint, iswalnum, iswalpha, iswcntrl, iswdigit, iswgraph, iswlower, iswpunct, iswspace, iswupper, iswxdigit, setlocale

iswpunct - auf Sonderlangzeichen prüfen

Definition

```
#include <wctype.h>

int iswpunct(wint_t wc);
```

`iswpunct` überprüft, ob *wc* ein Sonderlangzeichen ist, d.h. weder ein Steuerlangzeichen noch ein alphanumerisches Langzeichen, noch ein Langzeichen für Zwischenraum (siehe `iswspace`).
In allen Fällen ist das Argument *wc* vom Typ `wint_t`. Der Wert von *wc* muss ein Langzeichenwert sein, der einem gültigen Zeichen in der aktuellen Lokalität entspricht, oder er muss gleich dem Wert des Makros WEOF sein. Wenn *wc* irgendeinen anderen Wert besitzt, ist das Verhalten undefiniert.

Returnwert

$\neq 0$	*wc* ist Sonderlangzeichen
0	*wc* ist kein Sonderlangzeichen

Hinweise

In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

`iswpunct` ist sowohl als Makro als auch als Funktion realisiert (siehe Abschnitt „Funktionen und Makros“ auf Seite 19).

Das Verhalten von `iswpunct` wird von der Klasse `punct` der aktuellen Lokalität bestimmt. Die aktuelle Lokalität ist die C-Lokalität, wenn nicht explizit mit `setlocale` umgeschaltet wurde.

Siehe auch

ispunct, iswalnum, iswalpha, iswcntrl, iswdigit, iswgraph, iswlower, iswprint, iswspace, iswupper, iswxdigit, setlocale

iswspace - auf Zwischenraum-Langzeichen prüfen

Definition

```
#include <wctype.h>

int iswspace(wint_t wc);
```

`iswspace` überprüft, ob *wc* ein Zwischenraum-Langzeichen ist. Zwischenraum-Langzeichen sind: Leerzeichen, horizontaler Tabulator, Wagenrücklauf, Zeilenvorschub, Seitenvorschub oder vertikaler Tabulator.

In allen Fällen ist das Argument *wc* vom Typ `wint_t`. Der Wert von *wc* muss ein Langzeichenwert sein, der einem gültigen Zeichen in der aktuellen Lokalität entspricht, oder er muss gleich dem Wert des Makros WEOF sein. Wenn *wc* irgendeinen anderen Wert besitzt, ist das Verhalten undefiniert.

Returnwert

$\neq 0$	*wc* ist Zwischenraum-Langzeichen
0	*wc* ist kein Zwischenraum-Langzeichen

Hinweise

In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

`iswspace` ist sowohl als Makro als auch als Funktion realisiert (siehe Abschnitt „Funktionen und Makros“ auf Seite 19).

Das Verhalten von `iswspace` wird von der Klasse `space` der aktuellen Lokalität bestimmt. Die aktuelle Lokalität ist die C-Lokalität, wenn nicht explizit mit `setlocale` umgeschaltet wurde.

Siehe auch

isspace, iswalnum, iswalpha, iswcntrl, iswdigit, iswgraph, iswlower, iswprint, iswpunct, iswupper, iswxdigit, setlocale

iswupper - auf Großbuchstaben-Langzeichen prüfen

Definition

```
#include <wctype.h>

int iswupper(wint_t wc);
```

`iswupper` überprüft, ob das Langzeichen *wc* ein Großbuchstabe ist.

In allen Fällen ist das Argument *wc* vom Typ `wint_t`. Der Wert von *wc* muss ein Langzeichenwert sein, der einem gültigen Zeichen in der aktuellen Lokalität entspricht, oder er muss gleich dem Wert des Makros WEOF sein. Wenn *wc* irgendeinen anderen Wert besitzt, ist das Verhalten undefiniert.

Returnwert

$\neq 0$	*wc* ist Großbuchstabe
0	*wc* ist kein Großbuchstabe

Hinweise

In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

`iswupper` ist sowohl als Makro als auch als Funktion realisiert (siehe Abschnitt „Funktionen und Makros“ auf Seite 19).

Das Verhalten von `iswupper` wird von der Klasse `upper` der aktuellen Lokalität bestimmt. Die aktuelle Lokalität ist die C-Lokalität, wenn nicht explizit mit `setlocale` umgeschaltet wurde.

Siehe auch

isupper, iswalnum, iswalpha, iswcntrl, iswdigit, iswgraph, iswlower, iswprint, iswpunct, iswspace, iswxdigit, setlocale

iswxdigit - auf Hexadezimal-Langzeichen prüfen

Definition

```
#include <wctype.h>

int iswxdigit(wint_t wc);
```

`iswxdigit` überprüft, ob das Langzeichen *wc* ein hexadezimales Ziffernzeichen (0-9, A-F bzw. a-f) ist.

In allen Fällen ist das Argument *wc* vom Typ `wint_t`. Der Wert von *wc* muss ein Langzeichenwert sein, der einem gültigen Zeichen in der aktuellen Lokalität entspricht, oder er muss gleich dem Wert des Makros WEOF sein. Wenn *wc* irgendeinen anderen Wert besitzt, ist das Verhalten undefiniert.

Returnwert

$\neq 0$	*wc* ist hexadezimales Ziffernzeichen
0	*wc* ist keine hexadezimales Ziffernzeichen

Hinweise

In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

`iswxdigit` ist sowohl als Makro als auch als Funktion realisiert (siehe Abschnitt „Funktionen und Makros“ auf Seite 19).

Das Verhalten von `iswxdigit` wird von der Klasse `xdigit` der aktuellen Lokalität bestimmt. Die aktuelle Lokalität ist die C-Lokalität, wenn nicht explizit mit `setlocale` umgeschaltet wurde.

Siehe auch

iswalnum, iswalpha, iswcntrl, iswdigit, iswgraph, iswlower, iswprint, iswpunct, iswspace, iswupper, isxdigit

isxdigit - Sedezimale Ziffer?

Definition

```
#include <ctype.h>

int isxdigit(int c);
```

`isxdigit` überprüft, ob das Zeichen *c* eine sedezimale Ziffer (0-9), (A-F) bzw. (a-f) ist. *c* ist der EBCDIC-Wert des zu überprüfenden Zeichens.

Returnwert

≠ 0	Sedezimale Ziffer.
0	keine sedezimale Ziffer.

Hinweis

`isxdigit` ist sowohl als Makro als auch als Funktion realisiert (siehe Abschnitt „Funktionen und Makros“ auf Seite 19).

Beispiel

```
#include <ctype.h>
#include <stdio.h>

int main(void)
{
  int c;
  while((c = getchar()) != EOF)
  printf("%s : %c\n",((isxdigit(c))? "Sedezimale Ziffer" : "Sonstiges"), c);
  return 0;
}
```

Siehe auch

isalnum, isascii, iscntrl, isalpha, islower, isprint, ispunct, isgraph, isspace, isupper, isdigit, isebcdic, iswxdigit

j0, j1, jn - Besselfunktionen der ersten Art

Definition

```
#include <math.h>
double j0(double x);
double j1(double x);
double jn(int n, double x);
```

Die Funktionen `j0`, `j1` und `jn` berechnen die Besselfunktionen der ersten Art für Gleitkommawerte x und die ganzzahligen Ordnungen 0, 1 bzw. n.

Returnwert Besselfunktion für x.

Siehe auch y0, y1, yn

kill - Signal an eigenes Programm senden

Definition

```
#include <signal.h>

int kill(int pnr, int sig);
```

`kill` wird aus Kompatibilitätsgründen weiter unterstützt und arbeitet wie die ANSI-Funktion `raise`. Beschreibung siehe dort.

Der einzige Unterschied: Die Funktion `kill` erwartet als erstes Argument die Programmnummer *pnr*, die immer 0 sein muss, da das Signal nur an das eigene Programm geschickt werden kann (siehe auch Returnwert -1).

Returnwert

0	Das Signal wurde erfolgreich geschickt.
-1	Das Signal konnte nicht gesendet werden, weil *sig* keine gültige Signalnummer ist oder die Programmnummer *pnr* ungleich 0 ist. Zusätzlich wird `errno` auf den entsprechenden Fehlercode gesetzt: EINVAL (ungültige Signalnummer) ESRCH (Programmnummer ungleich 0).

Beispiel Ein Programm, das sich selbst abbricht.

```
#include <signal.h>

int main(void)
{
  for(;;)
    kill(0, SIGKILL);
  return 0;
}
```

Siehe auch alarm, raise, signal

labs - Absolutbetrag einer ganzen Zahl (long int) berechnen

Definition

```
#include <stdlib.h>
long int labs(long int j);
```

`labs` berechnet den Absolutbetrag einer ganzen Zahl *j* vom Typ `long int`.

Returnwert

`|j|`	für einen ganzzahligen Wert *j*. bei Erfolg.
undefiniert	bei Über- oder Unterlauf. `errno` wird auf ERANGE gesetzt, um den Fehler anzuzeigen.

Hinweis

Der Absolutbetrag der betragsmäßig größten negativen Zahl des Typs `long int` ist nicht darstellbar. Wenn als Argument *j* diese Zahl angegeben wird, wird das Programm mit Fehler (ERANGE) beendet.

Siehe auch

abs, cabs, fabs, llabs

ldexp - Wert im Zweiersystem berechnen

Definition #include <math.h>

double ldexp(double x, int exp);

`ldexp` berechnet aus seinen Argumenten x (Mantisse) und exp (Exponent) die Zahl:

$x * 2^{exp}$

`ldexp` ist die Umkehrfunktion zu `frexp`.

Returnwert $x * 2^{exp}$ bei Erfolg.

+/-HUGE_VAL bei Überlauf (je nach Vorzeichen von x). Zusätzlich wird `errno` auf ERANGE gesetzt (Resultat zu groß).

Beispiel `ldexp` ist die Umkehrfunktion zu `frexp`:
`frexp` zerlegt sein Gleitkommaargument in Mantisse und Exponent zur Basis 2, und `ldexp` berechnet aus diesen Teilen wieder den ursprünglichen Wert in der internen Gleitkommazahl-Darstellung, hier vorgeführt für die Zahl 5.342:

```
#include <stdio.h>
#include <math.h>

int main(void)
{
  double x;
  int ex;

  x = frexp(5.342, &ex);
  printf("Mantisse : %f\nExponent : %d\n", x, ex);
  printf("Ausgangswert : %f\n", ldexp(x, ex));
  return 0;
}
```

Siehe auch frexp, modf

ldiv - Division mit ganzen Zahlen (long int)

Definition #include <stdlib.h>

ldiv_t ldiv(long int dividend, long int divisor);

`ldiv` berechnet den Quotienten und den Rest der Division *dividend* durch *divisor*. Sowohl die Argumente als auch das Ergebnis sind vom Typ `long int`.

Das Vorzeichen des Quotienten ist gleich dem Vorzeichen des algebraischen Quotienten. Die Größe des Quotienten ist die größte ganze Zahl kleiner oder gleich dem absoluten Wert des algebraischen Quotienten.

Der Rest ergibt sich aus der Gleichung

Quotient * Divisor + Rest = Dividend

Returnwert Struktur vom Typ `ldiv_t`,
die sowohl den Quotienten *quot* als auch den Rest *rem* als `long`-Werte enthält.

Beispiel
```
ldiv_t d;

d = ldiv( 7, 3);        /*  d.quot =  2,  d.rem =  1  */
d = ldiv(-7, 3);        /*  d.quot = -2,  d.rem = -1  */
d = ldiv( 7,-3);        /*  d.quot = -2,  d.rem =  1  */
d = ldiv(-7,-3);        /*  d.quot =  2,  d.rem = -1  */
```

Siehe auch div, lldiv

_ _LINE_ _ - Ausgabe der aktuellen Zeilennummer

Definition _ _LINE_ _

Dieses Makro generiert die aktuelle Zeilennummer des Quellprogramms als Dezimalzahl.

Hinweis Das Makro muss in keiner Include-Datei definiert werden. Sein Name wird vom Compiler erkannt und ersetzt.

llabs - Absolutbetrag einer ganzen Zahl (long long int)

Definition

```
#include <stdlib.h>

long long int llabs(long long int j);
```

`llabs` berechnet den Absolutbetrag einer ganzen Zahl *j* vom Typ `long long int`.

Returnwert

`|j|`	für einen ganzzahligen Wert *j*.
undefiniert	bei Über- oder Unterlauf. `errno` wird auf ERANGE gesetzt, um den Fehler anzuzeigen.

Hinweis Der Absolutbetrag der betragsmäßig größten negativen Zahl des Typs `long long int` ist nicht darstellbar. Wenn als Argument *j* die betragsmäßig größte negative Zahl angegeben wird, wird das Programm mit Fehler (ERANGE) beendet.

Siehe auch abs, cabs, labs

lldiv - Division mit ganzen Zahlen (long long int)

Definition #include <stdlib.h>

lldiv_t lldiv(long long int dividend, long long int divisor);

`lldiv` berechnet den Quotienten und den Rest der Division *dividend* durch *divisor*. Sowohl die Argumente als auch das Ergebnis sind vom Typ `long long int`.

Das Vorzeichen des Quotienten ist gleich dem Vorzeichen des algebraischen Quotienten. Die Größe des Quotienten ist die größte ganze Zahl kleiner oder gleich dem absoluten Wert des algebraischen Quotienten.

Der Rest ergibt sich aus der Gleichung

Quotient * Divisor + Rest = Dividend

Returnwert Struktur vom Typ `lldiv_t`,
die sowohl den Quotienten *quot* als auch den Rest *rem* als `long long`-Werte enthält.

Beispiel siehe `ldiv`.

Siehe auch div, ldiv

llrint, llrintf, llrintl - auf nächste ganze Zahl runden

Definition

```
#include <math.h>

long long int llrint(double x);

long long int llrintf (float x);

long long int llrintl (long double x);
```

Die Funktionen geben jeweils die ganze Zahl zurück, die x am nächsten liegt - dargestellt als Zahl vom Typ `long long int`.

Der zurückgegebene Wert ist entsprechend dem aktuell gesetzten Rundungsmodus des Rechners gerundet. Wenn der Rundungsmodus 'round-to-nearest' gesetzt ist und die Differenz zwischen x und dem gerundeten Ergebnis genau 0.5 ist, wird die nächste gerade Ganzzahl zurückgegeben.

Wenn der aktuell eingestellte Rundungsmodus in Richtung positiv unendlich rundet, ist ll`rint` äquivalent zu `ceil`. Wenn der aktuell eingestellte Rundungsmodus in Richtung negativ unendlich rundet, ist ll`rint` äquivalent zu `floor`.
In dieser Version ist der Rundungsmodus fest auf Richtung positiv unendlich eingestellt.

Returnwert

ganze Zahl	dargestellt als Zahl vom Typ `long long int` bei Erfolg.
undefiniert	bei Über- oder Unterlauf. `errno` wird auf ERANGE gesetzt, um den Fehler anzuzeigen.

Hinweis In dieser Version ist der Rundungsmodus fest auf Richtung positiv unendlich eingestellt.

Siehe auch abs, ceil, floor, llround, lrint, lround, rint, round

llround, llroundf, llroundl - auf nächste ganze Zahl runden

Definition

```
#include <math.h>

long long int llround(double x);

long long int llroundf (float x);

long long int llroundl (long double x);
```

Die Funktionen geben jeweils die ganze Zahl zurück, die x am nächsten liegt, dargestellt als Zahl vom Typ `long long int`.

Der zurückgegebene Wert ist unabhängig vom eingestellten Rundungsmodus. Wenn die Differenz zwischen x und dem gerundeten Ergebnis genau 0.5 ist, wird die betragsmäßig größere ganze Zahl zurückgegeben.

Returnwert

ganze Zahl	dargestellt als Zahl vom Typ `long long int`. bei Erfolg.
undefiniert	bei Über- oder Unterlauf. `errno` wird auf ERANGE gesetzt, um den Fehler anzuzeigen.

Siehe auch abs, ceil, floor, llrint, lrint, lround, rint, round

localeconv - Lokalitätsspezifische Daten abfragen/ändern

Definition

```
#include <locale.h>

struct lconv *localeconv(void);
```

`localeconv` setzt die Komponenten einer Struktur vom Typ `struct lconv` auf Werte, die zu der aktuellen Lokalität passen. Die gelieferten Werte können bei der formatierten Ausgabe benutzt werden, um monetäre und nicht-monetäre numerische Werte lokalitätsspezifisch darzustellen.

Zu Programmbeginn ist die Lokalität "C" (LC_C_C) voreingestellt. Durch Aufruf der Funktion `setlocale` mit den Kategorien LC_MONETARY, LC_NUMERIC bzw. LC_ALL lässt sich die Lokalität ändern. Ein neuerlicher Aufruf von `localeconv` passt dann die Werte in den Strukturkomponenten der neuen Lokalität an.

Returnwert Zeiger auf die Struktur, in die die Werte eingetragen wurden.

1. Komponenten für nicht-monetäre numerische Werte (LC_NUMERIC):

   char *decimal_point
   : Dezimalpunkt.

   char *thousands_sep
   : Trennzeichen für die Gruppierung der Stellen vor dem Dezimalpunkt.

   char *grouping
   : Zeichenkette, deren Elemente die Länge jeder Gruppe von Stellen angeben.

2. Komponenten für monetäre Werte (LC_MONETARY):

   char *int_curr_symbol
   : Das zur Lokalität passende internationale Währungssymbol. Die ersten drei Zeichen enthalten das alphabetische internationale Währungssymbol, entsprechend der in ISO 4217:1897 festgelegten Konvention. Das vierte Zeichen ist das Trennzeichen zwischen dem internationalen Währungssymbol und dem Betrag.
   In der Lokalität "De.EDF04F@euro" ist der Wert “EUR“ als alphabetisches Währungssymbol eingetragen.

   char *currency_symbol
   : Das der Lokalität entsprechende lokale Währungssymbol.

   char *mon_decimal_point
   : Dezimalpunkt.

   char *mon_thousands_sep
   : Trennzeichen für die Gruppierung der Stellen vor dem Dezimalpunkt.

char *mon_grouping
Zeichenkette, deren Elemente die Länge jeder Gruppe von Stellen angeben.

char *positive_sign
Zeichenkette, die einen nicht-negativen Betrag kennzeichnet.

char *negative_sign
Zeichenkette, die einen negativen Betrag kennzeichnet.

char int_frac_digits
Anzahl der Dezimalstellen bei einem international strukturierten Betrag.

char frac_digits
Anzahl der Dezimalstellen bei einem lokal strukturierten Betrag.

char p_cs_precedes

1 falls das Währungssymbol dem nicht-negativen Betrag vorangeht.

0 falls das Währungssymbol dem nicht-negativen Betrag nachfolgt.

char n_cs_precedes

1 falls das Währungssymbol dem negativen Betrag vorangeht.

0 falls das Währungssymbol dem negativen Betrag nachfolgt.

char p_sep_by_space

1 falls das Währungssymbol von einem nicht-negativen Betrag durch ein Leerzeichen getrennt ist.

0 falls nicht.

char n_sep_by_space

1 falls das Währungssymbol von einem negativen Betrag durch ein Leerzeichen getrennt ist.

0 falls nicht.

char p_sign_posn
Position des *positive_sign* bei einem nicht-negativen Betrag.

char n_sign_posn
Position des *negative_sign* bei einem negativen Betrag.

Die `char`-Elemente von *grouping* und *mon_grouping* legen die Anzahl der Stellen für die Gruppen links vom Dezimalpunkt fest, beginnend mit der ersten Gruppe links vom Dezimalpunkt (z.B. Tausenderstellen). Die Einträge werden wie folgt interpretiert:

CHAR_MAX Entspricht dem höchsten EBCDIC-Wert (255) und bewirkt, dass keine weitere Gruppierung ausgeführt wird.

0 Das Nullbyte bewirkt, dass der Eintrag des vorhergehenden `char`-Elements für die Gruppierung aller restlichen Stellen gilt.

andere Der Intergerwert gilt für die Anzahl der Stellen der gegenwärtigen Gruppe. Das nächste `char`-Element legt die Anzahl der Stellen der nächsten Gruppe fest.

Die Werte von *p_sign_posn* und *n_sign_posn* werden wie folgt interpretiert:

0 Betrag und *currency_symbol* sind in Klammern eingeschlossen.

1 Das Vorzeichen steht vor Betrag und *currency_symbol*.

2 Das Vorzeichen steht hinter Betrag und *currency_symbol*.

3 Das Vorzeichen steht unmittelbar vor *currency_symbol*.

4 Das Vorzeichen steht unmittelbar hinter *currency_symbol*.

Hinweise Die verfügbaren Lokalitäten sind im Kapitel 6 beschrieben.

Die Komponenten der gelieferten Struktur dürfen nicht explizit vom Anwender überschrieben werden. Die Struktur kann ausschließlich durch den Aufruf von `localeconv` neu versorgt werden.

In der aktuellen Lokalität können für verschiedene Strukturkomponenten keine Werte definiert sein. Dies wird bei Komponenten vom Typ `char *` durch einen Zeiger auf "" dargestellt, bei Komponenten vom Typ `char` durch den Wert CHAR_MAX (Wert 255).

Siehe auch setlocale

localtime, localtime64 - Datum und aktuelle Uhrzeit als Struktur

Definition

```
#include <time.h>

struct tm *localtime(const time_t *sek_zg);
struct tm *localtime64(const time64_t *sek_zg);
```

`localtime` und `localtime64` interpretieren die Zeitangabe, auf die *sek_zg* zeigt, als Anzahl der seit dem Stichtag (Epoche) vergangenen Sekunden. Die Funktionen berechnen daraus Datum und Uhrzeit und speichern das Ergebnis in einer Struktur vom Typ `tm`. Negative Werte werden als Sekunden vor dem Stichtag interpretiert. Das kleinste darstellbare Datum ist der 01.01.1900 00:00:00 lokale Zeit.

Bei `localtime` hängt der Stichtag von der Verwendung des TIMESHIFT-Bindeschalters ab (siehe Abschnitt „Zeitfunktionen“ auf Seite 41):
– ohne TIMESHIFT-Bindeschalter (Standard): 1.1.1950 00:00:00.
– mit TIMESHIFT-Bindeschalter: 1.1.1970 00:00:00.
Bei `localtime64` ist der Stichtag immer der 1.1.1970 00:00:00.

Das größte darstellbare Datum ist bei `localtime` der 19.01.2018 03:14:07 (ohne TIMESHIFT-Bindeschalter) bzw. der 19.01.2038 03:14:07 (mit TIMESHIFT-Bindeschalter).

`localtime64` kann unabhängig von der Verwendung des TIMESHIFT-Bindeschalters Daten bis zum 18.3.4317 02:44:48 darstellen.

Returnwert Zeiger auf die berechnete Struktur. `localtime` und `localtime64` legen das Ergebnis in einer Struktur ab, die wie folgt in <time.h> deklariert ist:

```
struct tm
{
  int   tm_sec;         /* Sekunden (0-59) */
  int   tm_min;         /* Minuten (0-59) */
  int   tm_hour;        /* Stunden (0-23) */
  int   tm_mday;        /* Tag des Monats (1-31) */
  int   tm_mon;         /* Monate ab Jahresbeginn (0-11) */
  int   tm_year;        /* Jahre seit 1900 */
  int   tm_wday;        /* Wochentag (0-6, Sonntag=0) */
  int   tm_yday;        /* Tage seit dem 1. Januar (0-365) */
  int   tm_isdst;       /* Sommerzeitanzeige */
};
```

NULL im Fehlerfall

Hinweise Die Funktionen `asctime`, `ctime`, `ctime64`, `gmtime`, `gmtime64`, `localtime` und `localtime64` schreiben ihre Ergebnisse in denselben C-internen Datenbereich, so dass der Aufruf einer dieser Funktionen das vorherige Ergebnis einer der anderen Funktionen überschreibt.

`localtime` und `localtime64` bilden alle Daten vor dem 1.1.1900 01:00:00 auf den 1.1.1900 01:00:00 ab.

Beispiel

```
#include <time.h>
#include <stdio.h>

struct tm *t;
time_t clk;
char *s;

int main(void)
{
  clk = time((time_t *)0);
  t = localtime(&clk);
  printf("Jahr: %d\n", t->tm_year + 1900);
  printf("Uhrzeit in Stunden: %d\n", t->tm_hour);
  printf("Jahrestag: %d\n", t->tm_yday);
  s = asctime(t);
  printf("%s", s);
  return 0;
}
```

Siehe auch asctime, ctime, ctime64, gmtime, gmtime64, time, time64

log - Natürlicher Logarithmus

Definition #include <math.h>

double log(double x);

`log` berechnet den natürlichen Logarithmus der positiven Gleitkommazahl x zur Basis e.

Returnwert

ln(x)	für positive x.
-HUGE_VAL	falls x kleiner 0 ist. Zusätzlich wird `errno` auf EDOM gesetzt (domain error). Oder falls x gleich 0 ist. Zusätzlich wird `errno` auf ERANGE gesetzt.

Beispiel

```
#include <math.h>
#include <stdio.h>

int main(void)
{
  double x;
  printf("Beispiel fuer log(x): Bitte x eingeben\n");
  if(scanf("%lf", &x) == 1)
  printf("x = %g log(x) = %g\n", x, log(x));
  return 0;
}
```

Siehe auch log10, exp

log10 - Logarithmus zur Basis 10

Definition

```
#include <math.h>
double log10(double x);
```

`log10` berechnet den Logarithmus der positiven Gleitkommazahl x zur Basis 10.

Returnwert

lg(x)	für positive x.
-HUGE_VAL	falls x kleiner 0 ist. Zusätzlich wird `errno` auf EDOM gesetzt (domain error). Oder falls x gleich 0 ist. Zusätzlich wird `errno` auf ERANGE gesetzt.

Beispiel

```
#include <math.h>
#include <stdio.h>

int main(void)
{
  double x;
  printf("Beispiel fuer log10(x): Bitte x eingeben\n");
  if(scanf("%lf", &x) == 1)
  printf("x = %g log10(x)= %g\n", x, log10(x));
  return 0;
}
```

Siehe auch log, exp

longjmp - Nicht lokaler Sprung

Definition

```
#include <setjmp.h>

void longjmp(jmp_buf env, int wert);
```

`longjmp` ist nur zusammen mit der Funktion `setjmp` anwendbar: Der Aufruf von `longjmp` bewirkt, dass das Programm an eine zuvor mit `setjmp` „gemerkte“ Stelle verzweigt. Im Unterschied zu `goto`-Sprüngen, die nur innerhalb derselben Funktion (also lokal) zulässig sind, erlauben `longjmp` und `setjmp` Sprünge von einer beliebigen Funktion in eine andere, noch aktive Funktion (nicht lokale Sprünge).

`setjmp` speichert den aktuellen Programmzustand (Adresse im C-Laufzeitstack, Befehlszähler, Registerinhalte) in eine Variable vom Typ `jmp_buf` (definiert in <setjmp.h>). `longjmp` stellt den durch `setjmp` gesicherten Programmzustand wieder her, und das Programm wird mit der Anweisung fortgesetzt, die unmittelbar auf den `setjmp`-Aufruf folgt.

Parameter

jmp_buf env
: Feld, in das setjmp seine Werte abgelegt hat. Der Typ `jmp_buf` ist in <setjmp.h> definiert.

int wert
: Ganze Zahl, die beim Wiederaufsetzen der Programmausführung als Rückgabewert des `setjmp`-Aufrufs interpretiert wird. Falls *wert* gleich 0 ist, liefert `setjmp` den Wert 1 zurück; 0 würde bedeuten, dass an die Stelle nach dem `setjmp`-Aufruf „normal“, d.h. nicht mit `longjmp` verzweigt wurde (siehe auch `setjmp`).

Hinweise

Das Verhalten ist undefiniert, wenn `longjmp` mit einem Argument *env* aufgerufen wird, das nicht zuvor durch einen `setjmp`-Aufruf belegt wurde.

Die Funktion, die den `setjmp`-Aufruf mit der Variablen *env* enthält, muss beim `longjmp`-Ansprung mit derselben Variablen noch aktiv sein, d.h. sie darf inzwischen nicht beendet worden sein (z.B. mit `exit` oder `return`).

Nicht lokale Sprünge sind nützlich bei der Unterbrechungsbehandlung (siehe `signal`). Erfolgt z.B. die Behandlung von Fehlern oder Unterbrechungen in Routinen auf niedriger Stufe (d.h. es sind eine Reihe zuvor aufgerufener Funktionen noch aktiv), lässt sich mit `longjmp` und `setjmp` die normale Abarbeitung der noch aktiven Funktionen umgehen und sofort zu einer Funktion auf höherer Ebene verzweigen. Ein `longjmp`-Aufruf aus einer Unterbrechungs- oder Fehlerroutine leert die Einträge im Laufzeitstack bis zu der durch `setjmp` markierten Stelle, d.h. alle bis dahin noch aktiven Funktionen auf niedrigerer Ebene sind nicht mehr aktiv und das Programm wird auf höherer Ebene fortgesetzt.

Beim Wiederaufsetzen der Programmausführung sind die Variablen wie nach einem `goto` belegt: Globale Variablen haben die Werte, die sie zum Zeitpunkt des `longjmp`-Aufrufs hatten. Register- und sonstige lokale Variablen sind undefiniert, d.h. sie sollten überprüft und ggf. neu belegt werden.

Beispiel

Ein typischer Anwendungsfall für `setjmp` und `longjmp` ist die Text-Ein-/Ausgabe in einem interaktiven Texteditor.
Wird während der Ein- bzw. Ausgabe das Programm unterbrochen, indem von außen ein Signal geschickt wird (z.B. Drücken der K2-Taste nach „please acknowledge“ oder nach einer Eingabeaufforderung), bewirkt dies das Ende der Text-Ein-/Ausgabe, ansonsten fährt der Texteditor mit der Ein-/Ausgabe fort.
Wie das mit `setjmp` und `longjmp` realisiert werden kann, sehen Sie in folgendem Programm (nur Signalbehandlung, kein Editor!):

```
#include <stdio.h>
#include <setjmp.h>
#include <signal.h>

FILE *fp;
jmp_buf env;

void intr(int sig)
{
   printf("\n   ***** Sie wollen den Text nicht? ****** \n");
   longjmp(env,0);
}

int main(void)
{  int c; char reply;
   setjmp(env);
   signal(SIGINT,intr);
   printf("Textausgabe? (y?n):\n");
   scanf("%1s",&reply);              /* Unterbrechung mit K2 möglich */
   if(reply == 'y')
     {
      fp = fopen("text","r");        /* Datei text muss existieren */
      while((c=getc(fp)) != EOF)
            putc((char)c,stdout);    /* Unterbrechung der Textausgabe mit K2
                                        nach "please acknowledge" möglich */
     }
   else printf("Keine Textausgabe\n");
   return 0;  }
```

Siehe auch setjmp, signal

lrint, lrintf, lrintl - auf nächste ganze Zahl runden

Definition

```
#include <math.h>

long int lrint(double x);

long int lrintf (float x);

long int lrintl (long double x);
```

Die Funktionen geben jeweils die ganze Zahl zurück, die x am nächsten liegt - dargestellt als Zahl vom Typ `long int`.

Der zurückgegebene Wert ist entsprechend dem aktuell gesetzten Rundungsmodus des Rechners gerundet. Wenn der Rundungsmodus 'round-to-nearest' gesetzt ist und die Differenz zwischen x und dem gerundeten Ergebnis genau 0.5 ist, wird die nächste gerade Ganzzahl zurückgegeben.

Wenn der aktuell eingestellte Rundungsmodus in Richtung positiv unendlich rundet, ist l`rint` äquivalent zu `ceil`. Wenn der aktuell eingestellte Rundungsmodus in Richtung negativ unendlich rundet, ist l`rint` äquivalent zu `floor`.
In dieser Version ist der Rundungsmodus fest auf Richtung positiv unendlich eingestellt.

Returnwert

ganze Zahl	dargestellt als Zahl vom Typ `long int` bei Erfolg.
undefiniert	bei Über- oder Unterlauf. `errno` wird auf ERANGE gesetzt, um den Fehler anzuzeigen.

Hinweis In dieser Version ist der Rundungsmodus fest auf Richtung positiv unendlich eingestellt.

Siehe auch abs, ceil, floor, llrint, llround, lround, rint, round

lround, lroundf, lroundl - auf nächste ganze Zahl runden

Definition

```
#include <math.h>

long int lround(double x);

long int lroundf (float x);

long int lroundl (long double x);
```

Die Funktionen geben jeweils die ganze Zahl zurück, die x am nächsten liegt, `d`argestellt als Zahl vom Typ `long int`.

Der zurückgegebene Wert ist unabhängig vom eingestellten Rundungsmodus. Wenn die Differenz zwischen x und dem gerundeten Ergebnis genau 0.5 ist, wird die betragsmäßig größere ganze Zahl zurückgegeben.

Returnwert

ganze Zahl	dargestellt als Zahl vom Typ `long long int`. bei Erfolg.
undefiniert	bei Über- oder Unterlauf. `errno` wird auf ERANGE gesetzt, um den Fehler anzuzeigen.

Siehe auch abs, ceil, floor, llrint, llround, lrint, rint, round

lseek, lseek64 - Lese-/Schreibzeiger positionieren (elementar)

Definition

```
#include <stdio.h>

off__t lseek(int dk, off_t distanz, int ort);
off64_t lseek64(int dk, off64_t distanz, int ort);
```

`lseek` und `lseek64` positionierten den Lese-/Schreibzeiger für die Datei mit Dateikennzahl *dk* gemäß den Angaben in *distanz* und *ort*. Sie haben damit die Möglichkeit, eine Datei nichtsequenziell zu bearbeiten. Als Ergebnis liefern `lseek` und `lseek64` die aktuelle Position in der Datei.

Für die Bearbeitung von Dateien > 2 GB verfahren Sie wie folgt:

– Falls das Define `_FILE_OFFSET_BITS 64` (siehe Seite 71) gesetzt ist, rufen Sie `lseek` auf. Implizit wird dann `lseek64` mit den passenden Parametern verwendet.

– Andernfalls müssen Sie `lseek64` aufrufen.

Es besteht kein funktionaler Unterschied zwischen `lseek` und `lseek64`, außer dass bei `lseek64` der Offset-Typ `off64_t` und der Rückgabetyp `off64_t` verwendet werden.

Textdateien (SAM, ISAM) lassen sich absolut auf Dateianfang und -ende positionieren sowie auf eine vorher mit `tell` gemerkte Position.

Binärdateien (PAM, INCORE) lassen sich sowohl absolut positionieren (s.o.) als auch relativ um eine gewünschte Anzahl Bytes, bezogen auf Dateianfang, Dateiende oder aktuelle Position.
SAM-Dateien werden mit elementaren Funktionen stets als Textdateien verarbeitet.

Parameter

int dk
: Dateikennzahl der Datei, deren Lese-/Schreibzeiger positioniert werden soll.

off_t / off64_t distanz, int ort
: Bedeutung, Kombinationsmöglichkeiten und Wirkung dieser Parameter sind für Text- und Binärdateien unterschiedlich und werden deshalb im Folgenden getrennt beschrieben.

Textdateien (SAM, ISAM)

Mögliche Werte der Parameter:

distanz	0L oder Wert, der durch einen vorhergehenden tell/lseek-Aufruf ermittelt wurde. 0LL oder Wert, der durch einen vorhergehenden seek64-Aufruf ermittelt wurde.
distanz (64-Bit-Schnittstelle)	0LL oder Wert, der durch einen vorhergehenden `ftell`/`ftell64`-Aufruf ermittelt wurde.
ort	SEEK_SET (Dateianfang) SEEK_CUR (aktuelle Position) SEEK_END (Dateiende)

Sinnvolle Kombinationsmöglichkeiten und Wirkung:

distanz	**ort**	**Wirkung**
tell/lseek-Wert bzw. lseek64-Wert	SEEK_SET	Positionieren auf die durch `tell` oder `lseek/lseek64` gemerkte Position.
0L bzw. 0LL	SEEK_SET	Positionieren auf Dateianfang.
0L bzw. 0LL	SEEK_CUR	Abfrage der aktuellen Position ohne Positionierung.
0L bzw. 0LL	SEEK_END	Positionieren auf Dateiende.

Binärdateien (PAM, INCORE)

Mögliche Werte der Parameter:

distanz	Anzahl der Bytes, um die der aktuelle Lese-/Schreibzeiger verschoben werden soll, und zwar positive Zahl: Vorwärtspositionieren Richtung Dateiende negative Zahl: Rückwärtspositionieren Richtung Dateianfang 0L: absolut Positionieren auf Dateianfang bzw. -ende.
ort	Bei absoluter Positionierung auf Dateianfang oder -ende, wohin der Lese-/Schreibzeiger verschoben werden soll und bei relativer Positionierung, von wo aus der Lese-/Schreibzeiger um *distanz* Bytes verschoben werden soll: SEEK_SET (Dateianfang) SEEK_CUR (aktuelle Position) SEEK_END (Dateiende)

Sinnvolle Kombinationsmöglichkeiten und Wirkung:

distanz	ort	Wirkung
0L bzw. 0LL	SEEK_SET	Positionieren auf Dateianfang.
0L bzw. 0LL	SEEK_CUR	Abfrage der aktuellen Position ohne Positionierung.
0L bzw. 0LL	SEEK_END	Positionieren auf Dateiende.
positive Zahl	SEEK_SET SEEK_CUR SEEK_END	Vorwärtspositionieren ab Dateianfang, ab aktueller Position, ab Dateiende (über das Dateiende hinaus).
negative Zahl	SEEK_CUR SEEK_END	Rückwärtspositionieren ab aktueller Position, ab Dateiende.
tell/lseek-Wert bzw. lseek64-Wert	SEEK_SET	Positionieren auf die durch einen `tell` oder `lseek`/`lseek64`-Aufruf gemerkte Position.

Returnwert Position in der Datei bei Erfolg, und zwar
bei Binärdateien die Anzahl Bytes, die der Lese-/Schreibzeiger vom Dateianfang entfernt ist,
bei Textdateien die absolute Position des Lese-/Schreibzeigers.

-1 im Fehlerfall. Zusätzlich wird in `errno` eine entsprechende Fehlerinformation abgelegt, und zwar
EBADF: Unzulässige Dateikennzahl
ESPIPE: Unzulässige Positionierung
EINVAL: Unzulässiges Argument.
EMDS: Bei nur zum Lesen geöffneter Binärdatei hinter das Dateiende positioniert.

Hinweise Die Aufrufe `lseek(dk, 0L, SEEK_CUR)` und `tell(dk)` sind äquivalent, d.h. sie rufen beide die aktuelle Position in der Datei ab, ohne zu positionieren.

Werden in eine Textdatei neue Sätze geschrieben (geöffnet zum Neuerstellen oder Anhängen) und erfolgt ein `lseek`/`lseek64`-Aufruf, dann werden zunächst ggf. restliche Daten aus dem C-internen Puffer in die Datei geschrieben und mit Zeilenende (\n) abgeschlossen.
Ausnahme bei ANSI-Funktionalität:
Wenn die Daten einer ISAM-Datei im Puffer nicht mit einem Neue-Zeile-Zeichen enden, bewirkt `lseek`/`lseek64` keinen Zeilenwechsel (bzw. Satzwechsel). D.h., die Daten werden beim Schreiben aus dem Puffer nicht automatisch mit einem Neue-Zeile-Zeichen abgeschlossen. Nachfolgende Daten verlängern den Satz in der Datei. Beim Lesen einer ISAM-Datei werden daher nur Neue-Zeile-Zeichen eingelesen, die vom Programm explizit geschrieben wurden.

Wenn Sie bei einer zum Schreiben geöffneten Binärdatei hinter das Dateiende positionieren, entsteht ein „Loch“ zwischen den letzten physisch gespeicherten Daten und den neu geschriebenen Daten. Lesen aus diesem „Loch“ liefert binäre Nullen.
Wenn Sie bei einer nur zum Lesen geöffneten Binärdatei hinter das Dateiende positionieren, führt das zu einem Fehler (EMDS).

Auf Systemdateien (SYSDTA, SYSLST, SYSOUT) kann nicht positioniert werden.

Da die Informationen über die Dateiposition in einem 4 Byte langen Feld abgelegt werden, ergeben sich für die Größe von SAM- und ISAM-Dateien folgende Einschränkungen bei der Bearbeitung mit `tell/lseek`:

1. SAM-Datei

Satzlänge	≤ 2048 Byte
Satzanzahl/Block	≤ 256
Blockanzahl	≤ 2048

2. ISAM-Datei

Satzlänge	≤ 32 KByte
Satzanzahl	≤ 32 K

Beispiel Folgendes Programm liest ab Position 10 aus der Datei, die als erstes Argument beim Aufruf übergeben wird, fügt den Inhalt ans Ende der Datei an, falls ein zweites Argument angegeben wird, oder schreibt auf Standardausgabe (funktioniert nur mit Binärdateien, d.h. in diesem Fall nur mit PAM-Dateien):

```
#include <stdio.h>
#include <stdlib>

int fd1, fd2;
long result;
char c;

int main(int argc, char *argv[])
{
  if((fd1 = open (argv[1],0)) < 0) exit(1);
  if(argc < 3)
          fd2 = 1;
  else
          fd2 = open(argv[2], 1);

  result = lseek(fd1, 10L, SEEK_SET);
  printf("aktuelle Position in Datei1 : %ld\n", tell(fd1));

  /* Weitere mögliche Positionsabfragen:
```

```
  printf("aktuelle Position in Datei1: %ld\n, result);
  printf("aktuelle Position in Datei1: %ld\n, lseek(fd1, OL, SEEK_CUR)); */

  while(read(fd1, &c, 1) > 0)
          write(fd2, &c, 1);
  close(fd1);
  close(fd2);
  return 0;
}
```

Siehe auch tell, fseek, fseek64, ftell, ftell64

malloc - Speicherplatz reservieren

Definition

```
#include <stdlib.h>

void *malloc(size_t anz);
```

`malloc` beschafft zur Ausführungszeit zusammenhängenden Speicherplatz in der Größe von *anz* Bytes.

`malloc` ist Teil eines C-spezifischen Speicherverwaltungspaketes, das angeforderte und wieder freigegebene Speicherbereiche intern verwaltet. Neue Anforderungen werden zuerst aus bereits verwalteten Bereichen zu erfüllen versucht, dann erst vom Betriebs-system (vgl. Funktion `garbcoll`).

Returnwert

Zeiger auf den neuen Speicherbereich
: falls `malloc` neuen Speicherplatz zuweisen konnte. Dieser Zeiger kann für beliebige Datentypen verwendet werden.

NULL-Zeiger
: falls `malloc` den Speicherplatz nicht beschaffen konnte, z.B. weil der noch vorhandene Speicherplatz nicht ausreicht oder ein Fehler auftrat.

Hinweise

Der neue Datenbereich beginnt auf Doppelwortgrenze.

Die tatsächliche Länge des Datenbereichs beträgt:
Angeforderte Länge *anz* + 8 Bytes für interne Verwaltungsdaten. Diese Summe wird ggf. auf die nächste Zweierpotenz aufgerundet.

Wenn `malloc` in der Liste der freien Blöcke nicht genug Speicherplatz findet, wird intern die Funktion `memalloc` aufgerufen, um vom System mehr Speicherplatz zu bekommen.

Um sicherzugehen, dass Sie ausreichend Platz für eine Variable anfordern, sollten Sie die Funktion `sizeof` verwenden.

Wird die Länge des zur Verfügung gestellten Speicherbereiches beim Schreiben überschritten, führt dies zu schwerwiegenden Fehlern im Arbeitsspeicher.

Falls *anz* den Wert 0 besitzt, liefert `malloc` eine eindeutige Adresse, die auch an `free` übergeben werden kann.

Beispiel 1

Folgender Programmausschnitt fordert Speicherplatz für 30 integer-Elemente an.

```
#include <stdlib.h>

int *int_array;
    .
    .
    .
int_array = (int *)malloc(30 * sizeof(int));
```

Beispiel 2 Dynamische Speicherplatzreservierung für Gebrauchtwagen-Informationen:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 20;

struct pkw {
       char   *typ;
       int    alter;
       long   kilometer;
       char   tuev[6];
       int    zust;
       int    preis;
       struct pkw *n;
       } *list;

int main(void)
{
  int mark;
  if((list = (struct pkw *)malloc(sizeof(*list))) == NULL)
    {
      printf("speicherplatz aufgebraucht\n");
      exit(1);
    }
        /* achtung! Beim vorigen malloc-Aufruf wurde für die Komponente typ
           lediglich der Platz für einen Zeiger (4 Bytes) bereitgestellt
        Der Platz für die Typbezeichnung selbst muss noch beschafft werden:
*/
  if((list->typ = (char *)calloc(1,20)) == NULL)
      exit(1);             /* Fehler */
                           /* Gebrauchtwagen einlesen */
  scanf("%20s %d", list->typ, &list->alter);
  scanf("%d %6s %d %d", &list->kilometer, list->tuev, &list->zust,
   &list->preis);
  list->n = NULL;

                           /* eingelesene Werte ausdrucken */
  printf("%s\n%d\n", list->typ, list->alter);
  printf("%d\n%.6s\n%d\n%d", list->kilometer, list->tuev, list->zust,
   list->preis);

                           /* Speicherplatz wieder freigeben */
  free(list);
  return 0;
}
```

Siehe auch calloc, realloc, free, garbcoll, memalloc, memfree

mblen - Anzahl Bytes eines Multibyte-Zeichens ermitteln

Definition
```
#include <stdlib.h>

int mblen(const char *s, size_t n);
```

`mblen` liefert die Anzahl Bytes eines Multibyte-Zeichens, auf das *s* zeigt. Dabei werden maximal *n* Bytes in *s* ausgewertet.

Returnwert

-1	falls *n* = 0 ist.
0	falls *s* ein NULL-Zeiger ist oder auf ein Nullbyte (\0) zeigt.
1	sonst.

Hinweis In dieser Implementierung sind Zeichen, die aus mehreren Bytes bestehen, nicht realisiert. Multibyte-Zeichen haben immer die Länge 1.

Siehe auch mbstowcs, mbtowc, wcstombs, wctomb

mbrlen - Restlänge eines Multibyte-Zeichen ermitteln

Definition
```
#include <wchar.h>

size_t mbrlen(const char *s, size_t n, mbstate_t *ps);
```

`mbrlen` ermittelt die Anzahl Bytes ab der Position **s*, die zur Vervollständigung eines Multibyte-Zeichens benötigt werden. Es werden maximal *n* Bytes überprüft.

`mbrlen` entspricht dem Aufruf
`mbrtowc(NULL, s, n, ps!= NULL ? ps: internal)`
wobei *internal* das `mbstate_t`-Objekt für die Funktion ist.

Beschreibung siehe `mbrtowc`.

mbrtowc - Multibyte-Zeichen vervollständigen und in Langzeichen konvertieren

Definition

```
#include <wchar.h>

size_t mbrtowc(wchar_t *pwc, const char *s, size_t n, mbstate_t *ps);
```

Falls *s* kein NULL-Zeiger ist, ermittelt `mbrtowc`, wie viele Bytes ab der Position, auf die **s*, zeigt, zur Vervollständigung des nächsten Multibyte-Zeichens benötigt werden. Berücksichtigt werden auch eventuelle Umschalt-Sequenzen (Shift-Sequenzen). Es werden maximal die nächsten *n* Bytes überprüft. Wenn `mbrtowc` das Multibyte-Zeichen vervollständigen kann, wird das zugehörige Langzeichen ermittelt und unter **pwc* gespeichert, sofern *pwc* kein NULL-Zeiger ist.
Ist das zugehörige Langzeichen das Nullzeichen, entspricht der Ergebniszustand dem „initial conversion“ Zustand.
Ist *s* ein NULL-Zeiger, entspricht `mbrtowc` dem Aufruf

```
mbrtowc(NULL, "", 1, ps)
```

In diesem Fall werden die Werte der Parameter *pwc* und *n* ignoriert.

Returnwert Abhängig vom aktuellen Konvertierungs-Zustand gibt `mbrtowc` den Wert der ersten zutreffenden Bedingung zurück:

`0` wenn die nächsten (maximal *n*) Bytes ein gültiges Multibyte-Zeichen ergeben, das dem Langzeichen Null entspricht.

Anzahl der zur Vervollständigung des Multibyte-Zeichens benötigten Bytes
falls die nächsten (maximal *n*) Bytes ein gültiges Multibyte-Zeichen ergeben. Gespeichert wird das diesem Multibyte-Zeichen entsprechende Langzeichen.

`(size_t)-2` wenn die nächsten *n* Bytes ein unvollständiges, aber potenziell gültiges Multibyte-Zeichen ergeben. Es wird kein Wert gespeichert.

`(size_t)-1` wenn ein Kodierfehler auftritt, das heißt die nächsten (maximal *n*) Bytes ergeben kein vollständiges und gültiges Multibyte-Zeichen. Es wird kein Wert gespeichert und in `errno` wird der Wert des Makros `EILSEQ` geschrieben. Der Konversions-Zustand ist undefiniert.

Hinweis In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Siehe auch mblen, mbtowc, wcstombs, wctomb

mbsinit - auf „initial conversion“ Zustand überprüfen

Definition

```
#include <wchar.h>

int mbsinit(const mbstate_t *ps);
```

Wenn *ps* kein NULL-Zeiger ist, überprüft `mbsinit`, ob das `mbstate_t`-Objekt, auf das *ps* zeigt, einen „initial conversion“ Zustand beschreibt.

Returnwert

Wert ≠ 0	wenn *ps* ein NULL-Zeiger ist oder auf ein Objekt zeigt, das einen „initial conversion“-Zustand beschreibt
0	sonst.

mbsrtowcs - Multibyte-Zeichenkette in Langzeichenkette umwandeln

Definition

```
#include <wchar.h>

size_t mbsrtowcs(wchar_t *dst, const char **src, size_t len, mbstate_t *ps);
```

`mbsrtowcs` konvertiert eine Folge von Multibyte-Zeichen aus dem Feld, auf das *src* indirekt zeigt, in Langzeichen. `mbsrtowcs` beginnt die Umwandlung mit dem Konvertierungszustand, der in **ps* beschrieben wird. Die konvertierten Zeichen werden in das Feld geschrieben, auf das *dst* zeigt, sofern *dst* kein NULL-Zeiger ist. Jedes einzelne Zeichen wird so konvertiert, als sei die Funktion `mbrtowc` aufgerufen worden.

Die Umwandlung ist beendet, wenn ein abschließendes Nullzeichen auftritt. Das Nullzeichen wird ebenfalls umgewandelt und in das Feld geschrieben.

Die Umwandlung wird vorher abgebrochen, wenn

- eine Bytefolge auftritt, die kein gültiges Multibyte-Zeichen darstellt oder
- *dst* kein NULL-Zeiger ist und *len* Zeichen in das Feld, auf das *dst* zeigt, geschrieben worden sind.

Wenn *dst* kein NULL-Zeiger ist, wird dem Zeigerobjekt, auf das *src* zeigt, einer der beiden folgenden Werte zugewiesen:

- ein NULL-Zeiger, falls die Umwandlung mit dem Erreichen eines Nullzeichens beendet wurde
- die Adresse direkt hinter dem letzten umgewandelten Multibyte-Zeichen.

Wenn *dst* kein NULL-Zeiger ist und die Umwandlung mit dem Erreichen eines Nullzeichens beendet wurde, entspricht der Ergebniszustand dem „initial conversion“ Zustand.

Returnwert

`(size_t)-1` wenn ein Konvertierungsfehler auftritt, das heißt eine Folge von Bytes, die kein gültiges Multibyte-Zeichen ergeben. In `errno` wird der Wert des Makros `EILSEQ` geschrieben. Der Konversions-Zustand ist undefiniert.

Anzahl der erfolgreich konvertierten Multibyte-Zeichen
sonst. Das abschließende Nullzeichen (falls vorhanden) wird nicht mitgezählt.

Siehe auch mblen, mbtowc, wcstombs, wctomb

mbstowcs - Multibyte-Zeichenkette in Langzeichenkette umwandeln

Definition

```
#include <stdlib.h>

size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
```

`mbstowcs` wandelt eine Folge von Multibyte-Zeichen in der Zeichenkette *s* in die entsprechenden Langzeichen (Typ `wchar_t`) um und schreibt maximal *n* Langzeichen in den Bereich *pwcs*.
`mbstowcs` wandelt um, bis entweder *n* Werte konvertiert sind oder der Wert Null auftritt (Null wird in den `wchar_t`-Wert 0 konvertiert).

Ist *pwcs* ein NULL-Zeiger, gibt `mbstowcs` unabhängig vom Wert *n* die Länge zurück, die benötigt wird, um die gesamte Zeichenkette umzuwandeln, aber speichert keine Werte.

Wenn ein ungültiges Zeichen auftritt, liefert `mbstowcs` den Wert (`size_t`)-1 zurück.

Die von `mbstowcs` im Bereich *pwcs* abgespeicherten Langzeichen entsprechen den Werten der einzelnen Bytes in der Zeichenkette *s*.

Returnwert

Anzahl der in *pwcs* abgespeicherten Langzeichen (ohne das abschließende Nullbyte), wenn *pwcs* kein NULL-Zeiger ist.
Wenn der Returnwert dem Wert *n* entspricht, ist der Ergebnisbereich *pwcs* nicht mit dem Nullbyte abgeschlossen.

Länge, die benötigt wird, um die gesamte Zeichenkette umzuwandeln, wenn *pwcs* ein NULL-Zeiger ist. Es werden keine Werte gespeichert.

(`size_t`)-1 bei Fehler.

Hinweise

Bei sich überlappenden Speicherbereichen ist das Verhalten undefiniert.

In dieser Version sind Zeichen, die aus mehreren Bytes bestehen, nicht realisiert. Multibyte-Zeichen und Langzeichen haben immer die Länge 1 Byte.
Der Shift-Zustand des Multibyte-Zeichens wird ignoriert.

Siehe auch

mblen, mbtowc, wcstombs, wctomb

mbtowc - Multibyte-Zeichen in Langzeichen umwandeln

Definition

```
#include <stdlib.h>

int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

`mbtowc` wandelt ein Multibyte-Zeichen in *s* in das entsprechende Langzeichen um (`wchar_t`-Wert) und speichert dieses in den Bereich *pwc*. Dabei werden maximal *n* Bytes in *s* ausgewertet.

Das von `mbtowc` im Bereich *pwc* abgespeicherte Langzeichen enspricht dem Wert des Bytes in *s*.

Keine Zuweisung erfolgt, wenn
– *pwc* oder *s* ein NULL-Zeiger ist,
– *n* = 0 ist.

Returnwert

-1	falls *n* = 0 ist.
0	falls *s* ein NULL-Zeiger ist oder auf ein Nullbyte zeigt.
1	sonst.

Hinweis In dieser Version des C-Laufzeitsystems sind Zeichen, die aus mehreren Bytes bestehen, nicht realisiert. Langzeichen und Multibyte-Zeichen haben immer die Länge 1 Byte .

Siehe auch mblen, mbstowcs, wcstombs, wctomb

memalloc - Speicherplatz reservieren

Definition

```
#include <stdlib.h>

void *memalloc(size_t anz);
```

`memalloc` beschafft zur Ausführungszeit zusammenhängenden Speicherplatz in der Größe von *anz* Bytes.

`memalloc` reicht die Speicheranforderung direkt an den entsprechenden Betriebssystemaufruf durch. Die Funktion eignet sich vor allem für Speicherbereiche mit einer Größe von mehr als 2 KByte (siehe auch `memfree`).

Returnwert

Zeiger auf den neuen Speicherbereich, falls `memalloc` neuen Speicherplatz zuweisen konnte. Dieser Zeiger kann für beliebige Datentypen verwendet werden.

NULL-Zeiger falls `memalloc` den Speicherplatz nicht beschaffen konnte, z.B. weil der noch vorhandene Speicherplatz nicht ausreicht.

Hinweise

Der neue Speicherbereich beginnt auf Doppelwortgrenze.

Die angeforderte Länge *anz* wird auf das nächste Vielfache von 2 KByte aufgerundet.

Wird die Länge dieses Speicherbereiches beim Schreiben überschritten, führt dies zu schwerwiegenden Fehlern im Arbeitsspeicher.

Der mit `memalloc` angeforderte Speicherbereich kann mit `memfree` wieder freigegeben werden.

Siehe auch memfree

memchr - Zeichen in Speicherbereich suchen

Definition #include <string.h>

void *memchr(const void *s, int c, size_t n);

`memchr` sucht das erste Vorkommen des Zeichens *c* in den ersten *n* Bytes des Speicherbereiches, auf den *s* zeigt.

Returnwert Zeiger auf die Position von *c* im Bereich *s*
bei Erfolg.

NULL-Zeiger wenn *c* in dem angegebenen Bereich nicht enthalten ist.

Hinweise Die Funktion eignet sich für die Bearbeitung von Zeichenvektoren, die das Nullbyte (\0) enthalten, da `memchr` das Nullbyte nicht als 'Textende' interpretiert.

Für C++ gelten die beiden folgenden Prototypen für die Funktion `memstr`:
const void *memchr(const void *s, int c, size_t n);
void *memchr(void *s, int c, size_t n);

Siehe auch memcmp, memcpy, memset

memcmp - Speicherbereiche vergleichen

Definition

```
#include <string.h>

int memcmp(const void *s1, const void *s2, size_t n);
```

`memcmp` vergleicht die Inhalte der Speicherbereiche, auf die *s1* und *s2* zeigen, in den ersten *n* Bytes.

Returnwert

< 0	Der Inhalt von *s1* ist in den ersten *n* Bytes lexikalisch kleiner als der Inhalt von *s2*.
0	Die Inhalte von *s1* und *s2* sind in den ersten *n* Bytes lexikalisch gleich groß (d.h. identisch).
> 0	Der Inhalt von *s1* ist in den ersten *n* Bytes lexikalisch größer als der Inhalt von *s2*.

Hinweis Die Funktion eignet sich für die Bearbeitung von Zeichenvektoren, die das Nullbyte (\0) enthalten, da `memcmp` das Nullbyte nicht als 'Textende' interpretiert.

Siehe auch memchr, memcpy, memset

memcpy - Speicherbereich kopieren

Definition #include <string.h>

void *memcpy(void *s1, const void *s2, size_t n);

`memcpy` kopiert die ersten *n* Bytes des Speicherbereiches, auf den *s2* zeigt, in den Speicherbereich, auf den *s1* zeigt.

Returnwert Zeiger auf den Speicherbereich *s1*.

Hinweise Die Funktion eignet sich für die Bearbeitung von Zeichenvektoren, die das Nullbyte (\0) enthalten, da `memcpy` das Nullbyte nicht als 'Textende' interpretiert.

`memcpy` überprüft nicht, ob im Ergebnisbereich *s1* ein Überschreiben droht.

Bei sich überlappenden Speicherbereichen ist das Verhalten undefiniert.

Siehe auch memchr, memcmp, memset

memfree - Speicherbereich freigeben

Definition #include <stdlib.h>

void memfree(const void *zg, size_t anz);

`memfree` gibt den Speicherbereich, auf den *zg* zeigt, in der Größe von *anz* Bytes frei. *zg* muss das Ergebnis eines vorangegangenen `memalloc`-Aufrufs sein!

`memfree` reicht die Freigabeanforderung direkt an den entsprechenden Betriebssystemaufruf durch. `memfree` kann nur in Zusammenhang mit `memalloc` benutzt werden. Beide Funktionen eignen sich vor allem für Speicherbereiche, die größer sind als 2 KByte.

Hinweise Mit `memfree` kann nur ein mit `memalloc` angeforderter Speicherbereich freigegeben werden.

Die an `memfree` übergebenen Werte müssen mit denen vom entsprechenden `memalloc`-Aufruf übereinstimmen. Zufällige Werte führen zu schwerwiegenden Fehlern im Arbeitsspeicher!

Siehe auch memalloc

memmove - Speicherbereich kopieren

Definition #include <string.h>

void *memmove(void *s1, const void *s2, size_t n);

`memmove` kopiert die ersten *n* Bytes des Speicherbereiches, auf den *s2* zeigt, in den Speicherbereich, auf den *s1* zeigt.
`memmove` kopiert die *n* Bytes zunächst in ein temporäres Feld, das die Speicherbereiche *s1* und *s2* nicht überlappt und anschließend erst in den Speicherbereich *s1*.

Returnwert Zeiger auf den Speicherbereich *s1*.

Hinweise Die Funktion eignet sich für die Bearbeitung von Zeichenvektoren, die das Nullbyte (\0) enthalten, da `memmove` das Nullbyte nicht als 'Textende' interpretiert.

Im Unterschied zu `memcpy` funktioniert `memmove` auch mit Speicherbereichen, die sich überlappen.

Siehe auch memcpy

memset - Speicherbereich initialisieren

Definition #include <string.h>

void *memset(void *s, int c, size_t n);

`memset` kopiert den Wert des Zeichens *c* in die ersten *n* Bytes des Speicherbereiches, auf den *s* zeigt.

Returnwert Zeiger auf den Speicherbereich *s*.

Hinweise Die Funktion eignet sich für die Bearbeitung von Zeichenvektoren, die das Nullbyte (\0) enthalten, da `memset` das Nullbyte nicht als 'Textende' interpretiert.

`memset` überprüft nicht, ob im Ergebnisbereich *s* ein Überschreiben droht.

Siehe auch memchr, memcmp, memcpy

mktemp - Eindeutigen temporären Dateinamen erzeugen

Definition

```
#include <stdio.h>

char *mktemp(char *vorlage);
```

`mktemp` erzeugt aus einer Zeichenkette *vorlage*, die mindestens 8 Zeichen enthalten muss, eindeutige Namen für temporäre SAM-Dateien. Der Name wird aus den Zeichen in *vorlage* folgendermaßen gebildet:

- Die ersten drei Zeichen werden ersetzt durch "#T.",
- das vierte Zeichen wird ersetzt durch ein Zeichen, das sich pro `mktemp` Aufruf ändert (Buchstaben A-Z, Ziffern 0-9),
- die letzten vier Zeichen werden ersetzt durch die TSN-Nummer der aktuellen Task (seit LOGON),
- Zeichen zwischen den ersten und letzten vier Zeichen bleiben unverändert.

Hätte *vorlage* z.B den Inhalt "XXXX.ABC.XXXX" und die TSN-Nummer der aktuellen Task wäre 6082, dann erzeugt `mktemp` beim ersten Aufruf den temporären Namen

#T.A.ABC.6082

Returnwert

Zeiger auf die Ergebniszeichenkette, die den neuen Namen enthält	bei Erfolg.
NULL-Zeiger	wenn ein Fehler auftrat, z.B. weil *vorlage* weniger als 8 Zeichen enthält oder weil die zulässige maximale Anzahl (36) der `mktemp`-Aufrufe überschritten ist (siehe auch Hinweis).

Hinweise

Da für die Bildung eines eindeutigen Namens die Buchstaben A-Z und die Ziffern 0-9 verwendet werden, ist die Anzahl der `mktemp`-Aufrufe pro Programmablauf auf 36 begrenzt.

Temporäre Dateien werden automatisch bei Beendigung einer Task (LOGOFF) gelöscht. Wenn allerdings bei der Systemgenerierung das standardmäßige Präfix (#) für temporäre Dateien geändert wurde, bleiben die Dateien erhalten.

Beispiel Folgendes Programm erzeugt drei eindeutige temporäre Dateinamen und öffnet die Dateien zum Schreiben und Lesen.

```
#include <stdio.h>
FILE *fp1, *fp2, *fp3;
char s[] = "XXXX.temp.XXXX";

int main(void)
{
   mktemp(s);
   fp1 = fopen(s,"w+r");
   printf("%s\n",s);              /* erzeugter Name: #T.A.TEMP.6082 */

   mktemp(s);
   fp2 = fopen (s,"w+r");
   printf("%s\n",s);              /* erzeugter Name: #T.B.TEMP.6082 */

   mktemp(s);
   fp3 = fopen (s,"w+r");
   printf("%s\n",s);              /* erzeugter Name: #T.C.TEMP.6082 */
   return 0;
}
```

mktime, mktime64 - Umwandlung von Datum und Uhrzeit (Kalenderfunktion)

Definition

```
#include <time.h>

time_t mktime(struct tm *tm_zg);
time64_t mktime64(struct tm *tm_zg);
```

`mktime` und `mktime64` wandeln Datum und Uhrzeit, die der Benutzer in einer Struktur vom Typ `tm` angibt, in eine Zeitangabe um, die als Anzahl der seit dem Stichtag (Epoche) vergangenen Sekunden dargestellt ist. Daten vor dem Stichtag werden als negative Werte zurückgegeben. Das kleinste darstellbare Datum ist der 01.01.1900 00:00:00 lokale Zeit.

Bei der Berechnung vervollständigen `mktime` und `mktime64` die `tm`-Struktur mit den Werten für Wochentag (0-6) und Tag seit Jahresbeginn (0-365) und passen die Werte der übrigen Komponenten an die standardmäßig vorgesehenen Wertebereiche an (siehe auch Parameterbeschreibung).

Bei `mktime` hängt der Stichtag von der Verwendung des TIMESHIFT-Bindeschalters ab (siehe Abschnitt „Zeitfunktionen“ auf Seite 41):

– ohne TIMESHIFT-Bindeschalter (Standard): 1.1.1950 00:00:00.
– mit TIMESHIFT-Bindeschalter: 1.1.1970 00:00:00.

Bei `mktime64` ist der Stichtag immer der 1.1.1970 00:00:00.

Falls die Kalenderzeit wegen fehlender Angaben im Eingabeparameter nicht dargestellt werden kann, liefern `mktime/mktime64` den Rückgabewert -1. Dies gilt bei `mktime` auch für Daten nach dem 19.01.2018 03:14:07 (ohne TIMESHIFT-Bindeschalter) bzw. nach dem 19.01.2038 03:14:07 (mit TIMESHIFT-Bindeschalter).

`mktime64` kann unabhängig von der Verwendung des TIMESHIFT-Bindeschalters Daten bis zum 18.3.4317 02:44:48 darstellen.

Parameter struct tm *tm_zg

Zeiger auf eine Struktur vom Typ `tm`, die vom Benutzer mit den Datums- und Uhrzeitangaben versorgt und anschließend von `mktime/mktime64` aktualisiert wird. In Klammern sind die Wertebereiche angegeben, die standardmäßig vorgesehen sind.

```
int   tm_sec;          Sekunden (0-59)
int   tm_min;          Minuten (0-59)
int   tm_hour;         Stunden (0-23)
int   tm_mday;         Tag des Monats (1-31)
int   tm_mon;          Monate ab Jahresbeginn (0-11)
int   tm_year;         Jahre seit 1900
int   tm_wday;         Wochentag (0-6, Sonntag=0)
int   tm_yday;         Tage seit dem 1. Januar (0-365)
int   tm_isdst;        Sommerzeitanzeige:
                       0    Sommerzeit ist nicht aktiv
                       >0   Sommerzeit ist aktiv
                       <0   Information ist nicht verfügbar
```

1. Datums- und Uhrzeitangaben des Benutzers

Die Komponenten `tm_wday` und `tm_yday` brauchen nicht versorgt zu zu werden, da `mktime` diese bei der Berechnung von `time_t` ignoriert und anschließend selbst mit den passenden Werten versorgt.
Alle anderen Komponenten müssen mit einem Wert belegt sein. Diese Werte sind nicht auf die o.g. standardmäßigen Wertebereiche begrenzt, d.h. sie können auch größer oder kleiner sein.

Beispiele dafür:

-1 in `tm_hour` bedeutet 1 Stunde vor Mitternacht,
0 in `tm_mday` bedeutet den letzten Tag des Vormonats,
-2 in `tm_mon` bedeutet 2 Monate vor dem Januar im Jahre `tm_year`.

2. Aktualisierung der Struktur durch `mktime`

Die Komponenten `tm_wday` und `tm_yday` erhalten die zu den Benutzerangaben passenden Werte.
Die anderen Komponenten werden so belegt, dass ihre Werte den o.g. Standardbereichen entsprechen.
Der Wert von `tm_mday` wird nicht belegt, bevor nicht `tm_mon` und `tm_year` bestimmt sind.

Returnwert	Integer > 0	Bei Ortszeiten nach dem Stichtag (Epoche): Anzahl der Sekunden, die seit dem Stichtag vergangen sind.
	Integer < 0	Bei Ortszeiten vor dem Stichtag (Epoche): Anzahl der Sekunden, die bis dahin vergangen sind.
	(`time_t`) - 1	falls die Zeit nicht darstellbar ist.

Siehe auch asctime, ctime, ctime64, difftime, difftime64, ftime, ftime64, gmtime, gmtime64, localtime, localtime64, time, time64

modf - Zahl in ganzzahligen und gebrochenen Teil aufspalten

Definition

```
#include <math.h>

double modf(double z, double *g_zg);
```

`modf` zerlegt eine Gleitkommazahl *z* in ihren ganzzahligen und gebrochenen Teil. Das Ergebnis von `modf` ist der Bruchteil mit Vorzeichen und indirekt über einen Ergebnisparameter *g_zg* der ganzzahlige Anteil.

Returnwert Bruchteil von *z* mit Vorzeichen.

Hinweis Beachten Sie, dass das Argument *g_zg* ein Zeiger sein muss!

Beispiel Folgendes Programm zerlegt die Zahl -456.789 in ihren ganzzahligen und gebrochenen Teil.

```
#include <math.h>
#include <stdio.h>

int main(void)
{
  double x, g;
  x = modf(-456.789, &g);
  printf("Bruchteil : %g\nGanzzahliger Teil : %g\n", x, g);
  return 0;
}
```

Siehe auch frexp, ldexp

offsetof - Abstand einer Strukturkomponente zum Strukturbeginn

Definition #include <stddef.h>

size_t offsetof(typ, komponente);

`offsetof` liefert den Abstand in Bytes, die die Strukturkomponente *komponente* vom Beginn der Struktur vom Typ *typ* (Etikett) entfernt ist.
`offsetof` ist ein Makro.

Returnwert Abstand der Strukturkomponente vom Strukturbeginn in Bytes.

Hinweis Ist die angegebene Strukturkomponente ein Bitfeld, ist das Verhalten undefiniert.

Beispiel

```
#include <stdio.h>
#include <stddef.h>

struct S1 {
   char c;
   int  i;
   double d;
};

int main(void)
{
  typedef struct S1 t_s1;

  printf("offsetof(struct S1, c) = %d\n", offsetof(struct S1, c) );
  printf("offsetof(struct S1, i) = %d\n", offsetof(struct S1, i) );
  printf("offsetof(struct S1, d) = %d\n", offsetof(struct S1, d) );
  printf("\n");

  printf("offsetof(t_s1, c) = %d\n", offsetof(t_s1, c) );
  printf("offsetof(t_s1, i) = %d\n", offsetof(t_s1, i) );
  printf("offsetof(t_s1, d) = %d\n", offsetof(t_s1, d) );
  printf("\n");
  return 0;
}
```

open, open64 - Datei öffnen (elementar)

Definition

```
#include <stdio.h>

int open(const char *d_name, int modus);
int open64(const char *d_name, int modus);
```

`open` und `open64` öffnen die Datei *d_name*. Die Zugriffsart hängt von dem (oktalen) Wert von *modus* ab. `open` und `open64` geben eine gültige Dateikennzahl zurück, die später in elementaren Zugriffsoperationen (`read`, `write`) zur Bezeichnung der Datei benutzt wird.

Es besteht kein funktionaler Unterschied zwischen `open` und `open64`, außer dass `open64` im File Status Flag implizit das Bit O_LARGEFILE setzt. Die Funktion `open64` entspricht der Verwendung der Funktion `open`, bei der O_LARGEFILE in *oflag* gesetzt ist.

Für die Bearbeitung von Dateien > 2 GB verfahren Sie wie folgt:

– Falls das Define `_FILE_OFFSET_BITS 64` (siehe Seite 71) gesetzt ist, rufen Sie `open` auf. Implizit wird dann `open64` mit den passenden Parametern verwendet.

– Andernfalls müssen Sie `open64` aufrufen.

Parameter

const char *d_name
Zeichenkette, die die zu öffnende Datei angibt. *d_name* kann sein:

– jeder gültige BS2000-Dateiname

– "link=*linkname*"
linkname bezeichnet einen BS2000-Linknamen

– "(SYSDTA)", "(SYSOUT)", "(SYSLST)"
die entsprechende Systemdatei

– "(SYSTERM)"
Terminal-Ein-/Ausgabe

– "(INCORE)"
temporäre Binärdatei, die nur im virtuellen Speicher angelegt wird

int modus

Konstante, die im Header <stdio.h> definiert ist und die gewünschte Zugriffsart angibt, (oder die entsprechende Oktalzahl) und zwar:

`O_RDONLY`
`0000`
: Öffnen zum Lesen. Die Datei muss bereits vorhanden sein.

`O_WRONLY`
`0001`
: Öffnen zum Schreiben. Die Datei muss bereits vorhanden sein. Der alte Inhalt bleibt erhalten.

`O_TRUNC|O_WRONLY`
`01001`
: Öffnen zum Schreiben. Ist die Datei vorhanden, wird der alte Inhalt gelöscht. Ist die Datei nicht vorhanden, wird sie neu erstellt.

`O_RDWR`
`0002`
: Öffnen zum Lesen und Schreiben. Die Datei muss bereits vorhanden sein. Der alte Inhalt bleibt erhalten.

`O_TRUNC|O_RDWR`
`01002`
: Öffnen zum Lesen und Schreiben. Ist die Datei vorhanden, wird der alte Inhalt gelöscht. Ist die Datei nicht vorhanden, wird sie neu erstellt.

`O_WRRD`
`0003`
: Öffnen zum Neuschreiben und Lesen. Ist die Datei vorhanden, wird der alte Inhalt gelöscht. Ist die Datei nicht vorhanden, wird sie neu erstellt.

`O_APPEND_OLD|O_TRUNC|O_WRONLY`
`0401`
: Öffnen zum Anfügen ans Ende der Datei. Die Datei muss bereits vorhanden sein. Es wird auf das Dateiende positioniert, d.h. der alte Inhalt bleibt erhalten und der neue Text wird ans Ende der Datei angehängt.

`O_APPEND_OLD|O_RDWR`
`0402`
: Öffnen zum Anfügen ans Ende der Datei und zum Lesen. Die Datei muss bereits vorhanden sein. Der alte Inhalt bleibt erhalten und der neue Text wird ans Ende der Datei angehängt. Nach dem Öffnen ist die Datei bei KR-Funktionalität (nur bei C/C++ Versionen kleiner V3.0 vorhanden) auf das Dateiende positioniert, bei ANSI-Funktionalität auf den Dateianfang.

lbp-Schalter

Der *lbp*-Schalter steuert die Behandlung des Last Byte Pointers (LBP). Er ist nur für Binärdateien mit Zugriffsart PAM relevant und kann mit jeder der oben angegebenen Konstanten kombiniert werden. Er wirkt sich erst aus, wenn die Datei geschlossen wird.

Beim Öffnen und Lesen einer bestehenden Datei wird der LBP unabhängig vom *lbp*-Schalter immer berücksichtigt:

– Ist der LBP der Datei ungleich 0, wird er ausgewertet. Ein eventuell vorhandener Marker wird ignoriert.

– Ist der LBP = 0, wird nach einem Marker gesucht und die Dateilänge daraus ermittelt. Falls kein Marker gefunden wird, wird das Ende des letzten vollständigen Blocks als Dateiende betrachtet.

`O_LBP`
: Beim Schließen einer Datei, die verändert oder neu erstellt wurde, wird kein Marker geschrieben (auch wenn einer vorhanden war) und ein gültiger LBP gesetzt. Auf diese Weise können Dateien mit Marker auf LBP ohne Marker umgestellt werden.

`O_NOLBP`
: Beim Schließen einer Datei, die verändert oder neu erstellt wurde, wird der LBP auf Null gesetzt. Für eine neu erstellte Datei wird immer ein Marker geschrieben, für eine veränderte nur dann, wenn vorher bereits ein Marker vorhanden war. War kein Marker vorhanden, wird auch keiner geschrieben und die Datei endet mit dem vollständigen letzten Block.

Wird der *lbp*-Schalter in beiden Varianten angegeben (O_LBP und O_NOLBP), so schlägt die Funktion `open`, `open64` fehl und `errno` wird auf `EINVAL` gesetzt.

Wird der *lbp*-Schalter nicht angegeben, hängt das Verhalten von der Umgebungsvariablen LAST_BYTE_POINTER ab (siehe auch „Umgebungsvariable LAST_BYTE_POINTER“ auf Seite 90):

`LAST_BYTE_POINTER=YES`
: Die Funktion verhält sich so, als ob `O_LBP` angegeben wäre.

`LAST_BYTE_POINTER=NO`
: Die Funktion verhält sich so, als ob `O_NOLBP` angegeben wäre.

Returnwert

Dateikennzahl	positive Zahl, die später bei den elementaren Zugriffsoperationen (`read`, `write`) zum Bezeichnen der Datei benutzt wird.
-1	wenn die Datei nicht geöffnet werden konnte, z.B. wegen fehlender Zugriffsberechtigung, falschem Datei- oder Linknamen etc.

Hinweise

Der BS2000-Dateiname bzw. -Linkname kann in Klein- und Großbuchstaben geschrieben werden, er wird automatisch in Großbuchstaben umgesetzt.

Wird eine nicht vorhandene Datei neu angelegt, wird standardmäßig folgende Datei erzeugt:
Bei KR-Funktionalität (nur bei C/C++ Versionen kleiner V3.0 vorhanden) eine SAM-Datei mit variabler Satzlänge und Standardblocklänge, bei ANSI-Funktionalität eine ISAM-Datei mit variabler Satzlänge und Standardblocklänge.
SAM-Dateien sind beim Öffnen mit `open` bzw. open64 immer Textdateien.

Bei Verwendung eines Linknamens lassen sich mit dem ADD-FILE-LINK-Kommando folgende Dateiattribute ändern: Zugriffsmethode, Satzlänge, Satzformat, Blocklänge und Blockformat (siehe Abschnitt „Systemdateien (SYSDTA, SYSOUT, SYSLST)“ auf Seite 73).

In allen Fällen, in denen der alte Inhalt einer bereits existierenden Datei gelöscht wird (0003, 01001), bleiben die Katalogeigenschaften dieser Datei erhalten.

Position des Schreib-/Lesezeigers im Anfügemodus:
Wenn der Schreib-/Lesezeiger in einer Datei, die im Anfügemodus eröffnet wurde (0401, 0402), explizit vom Dateiende wegpositioniert wurde (`lseek`/`lseek64`), wird er je nach KR- oder ANSI-Funktionalität unterschiedlich behandelt.
KR-Funktionalität (nur bei C/C++ Versionen kleiner V3.0 vorhanden): Der aktuelle Schreib-/Lesezeiger wird nur beim Schreiben mit der Elementarfunktion `write` ignoriert und automatisch ans Ende der Datei positioniert.
ANSI-Funktionalität: Der aktuelle Schreib-/Lesezeiger wird bei allen Schreibfunktionen ignoriert und automatisch ans Ende der Datei positioniert.

Der Versuch, eine nicht existierende Datei zum Lesen zu öffnen (0000, 0002), zum Ändern (0001) sowie zum Anfügen (0401, 0402), endet mit Fehler.

Sie können eine Datei gleichzeitig für verschiedene Zugriffsmodi eröffnen, sofern diese Modi im BS2000-Datenverwaltungssystem miteinander verträglich sind.

(INCORE)-Dateien können nur zum Neuschreiben (01001) oder zum Neuschreiben und Lesen (0003) eröffnet werden. Es müssen zuerst Daten geschrieben werden. Um die geschriebenen Daten wieder einlesen zu können, muss die Datei mit der Funktion `lseek`/`lseek64` auf Dateianfang positioniert werden.

Wenn ein Programm startet, werden die Standarddateien für Eingabe, Ausgabe und Fehlerausgabe automatisch mit folgenden Dateikennzahlen geöffnet:

`stdin`:	0
`stdout`:	1
`stderr`:	2

Es können maximal _NFILE Dateien gleichzeitig geöffnet sein. _NFILE ist in <stdio.h> mit 2048 definiert.

Beispiel Folgendes Programm öffnet die Datei *spass* zweimal zum Lesen und verarbeitet sie mit unterschiedlichen Dateikennzahlen (*fd1*, *fd2*).

```
#include <stdio.h>

int fd1,fd2;
char c;
int n;

int main(void)
{
               /* Datei "spass" das erste Mal
                  zum Lesen öffnen */
   if((fd1=open("spass",0)) == -1)
               /* Fehler beim ersten Öffnen */
      printf("fehler1\n");

               /* Datei "spass"
                  zum zweiten Mal zum Lesen öffnen */
   if((fd2=open("spass",0)) == -1)
               /* Fehler beim zweiten Öffnen */
      printf("fehler2\n");

               /* bis zum ersten 'a' wird über fd1 gelesen */
   while((n=read(fd1,&c,1)) > 0 && (c != 'a'))
               /* Ausgabe des gelesenen Zeichens auf
                  Standardausgabe */
         write(1,&c,n);

               /* jetzt wird über fd2 von Dateianfang!!
                  bis Dateiende gelesen */

   while((n=read(fd2,&c,1)) > 0)
               /* Ausgabe des gelesenen Zeichens auf
                  Standardausgabe */
         write(1,&c,n);

               /* über fd1 wird nach dem ersten 'a'
                  weitergelesen bis Dateiende */
   while((n=read(fd1,&c,1)) > 0)
               /* Ausgabe des gelesenen Zeichens auf
                  Standardausgabe */
         write(1,&c,n);
   return 0;
}
```

Siehe auch creat, creat64, fdopen, read, write, close

perror - Fehlermeldung ausgeben

Definition #include <stdio.h>

void *perror(const char *s);

`perror` schreibt auf die Standardfehlerausgabe eine Fehlermeldung, entsprechend dem Fehlercode in der C-internen Variablen `errno`. Zunächst wird eine als Argument übergebene Zeichenkette *s* ausgegeben, gefolgt von einem Doppelpunkt und dem kurzen Fehlertext aus <errno.h>; die Meldung wird mit einem Neue-Zeile-Zeichen abgeschlossen:

s : <kurze Fehlermeldung>\n

Als Fehlerinformationen erhalten Sie:

– einen Text, der den Fehler kurz beschreibt,
– den Namen der fehlerhaften Funktion und
– ggf. den DVS-Fehlercode (sedezimal).

Hinweise

Die `errno`-Fehlertexte enthalten z.B. bei Ein-/Ausgabefehlern oder bei der Ausführung von Systemkommandos als zusätzliche Information den entsprechenden DVS-Fehlercode. Eine Auflistung aller `errno`-Fehlercodes und Fehlertexte finden Sie in der Include-Datei <errno.h>.

Wird als Argument *s* ein NULL-Zeiger übergeben, wird lediglich der `errno`-Fehlertext ausgegeben.

Der Inhalt des Bereichs, in dem der Fehlercode und der Fehlertext abgespeichert sind, wird nicht explizit gelöscht. D.h. der alte Inhalt bleibt solange erhalten, bis er bei neuerlichem Auftritt eines Fehlers mit den entsprechenden Informationen überschrieben wird. `perror`-Aufrufe sind daher nur sinnvoll, unmittelbar nachdem eine Funktion einen Fehler-Returnwert geliefert hat.

Bei KR-Funktionalität (nur bei C/C++ Versionen kleiner V3.0 vorhanden) wird ein Returnwert vom Typ `char *` geliefert. Er enthält einen Zeiger auf einen C-internen Speicherbereich, in der die Fehlermeldung steht. Der Inhalt wird bei jedem `perror`-Aufruf überschrieben.

Beispiel

Folgendes Programm öffnet die Datei *dat* zum Lesen. Ist die Datei nicht vorhanden, erhält man folgende Fehlermeldung auf die Standardausgabe:

```
Programm fopen: dataset not found (cmd: OPEN), errorcode=DD33
```

DD33 ist dabei der DVS-Fehlercode.

```
#include <stdio.h>

int main(void)
{
  FILE *fp;
  if((fp = fopen("dat", "r")) == NULL)
    perror("Programm fopen");
  return 0;
}
```

pow - Allgemeine Exponentialfunktion

Definition

#include <math.h>

double pow(double x, double y);

`pow` berechnet x^y.
Falls x gleich 0 ist, muss y positiv sein,
falls x negativ ist, muss y ganzzahlig sein.

Returnwert

x^y	falls x, y und das Ergebnis im zulässigen Gleitkommaintervall liegen.
+/-HUGE_VAL	bei Überlauf (Vorzeichen von x). Zusätzlich wird `errno` auf ERANGE gesetzt (Resultat zu groß).
1.0	wenn x und y gleich 0 sind.
-HUGE_VAL	wenn x gleich 0 und y kleiner 0 ist. Zusätzlich wird `errno` auf EDOM gesetzt.
undefiniert	wenn x kleiner 0 und y nicht ganzzahlig. Zusätzlich wird `errno` auf EDOM gesetzt.

Beispiel

Folgendes Programm berechnet x^y für eingelesene Argumente x und y

```
#include <math.h>
#include <stdio.h>

int main(void)
{
  double x, y;
  scanf("%lf %lf", &x, &y);
  printf("%g**%g : %g\n", x, y, pow(x,y));
  return 0;
}
```

Siehe auch

exp, hypot, log, log10, sinh, sqrt

printf - Formatierte Ausgabe auf Standardausgabe

Definition

```
#include <stdio.h>

int printf(const char *format, argumentenliste);
```

`printf` bereitet Daten (Zeichen, Zeichenketten, numerische Werte) gemäß den Angaben in der Zeichenkette *format* für die Ausgabe auf und schreibt sie auf die Standardausgabe `stdout`. Numerische Werte werden dabei von ihrer internen Darstellung in abdruckbare Zeichen umgewandelt.

Parameter Die Formatzeichenkette kann folgende Angaben enthalten:

– Einfache Zeichen (char)

  Diese werden 1 : 1 ausgegeben.

– Steuerzeichen für Zwischenraum, beginnend mit einem Gegenschrägstrich (\)

– Formatanweisungen, beginnend mit dem Prozentzeichen (%)

  Gemäß den Angaben in den Formatanweisungen werden die in einer Argumentenliste übergebenen Daten formatiert und umgewandelt. Zu jedem Argument muss es eine Formatanweisung geben, wobei die erste Formatanweisung mit dem ersten Argument korrespondiert usw.

Einfache Zeichen

Alle Zeichen, die nicht Element einer Formatanweisung sind und kein spezielles Steuerzeichen darstellen (mit Gegenschrägstrich beginnend), werden unverändert ausgegeben. Soll das Prozent-Zeichen (%) geschrieben werden, muss es doppelt angegeben werden (%%).

Steuerzeichen für Zwischenraum

\n	Zeilenvorschub
\t	Tabulator
\f	Seitenwechsel
\v	vertikaler Tabulator
\b	Zeichen rücksetzen
\r	Wagenrücklauf

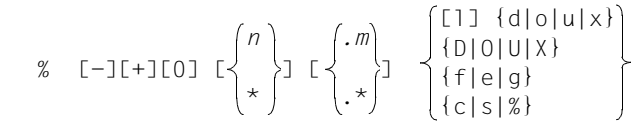
Nähere Informationen zur Umsetzung dieser Steuerzeichen finden Sie in Abschnitt „Zwischenraum“ auf Seite 68.

argumentenliste
Variablen oder Konstanten, deren Werte gemäß den Angaben in den Formatanweisungen für die Ausgabe umgewandelt und formatiert werden sollen.
Wenn die Anzahl der Formatanweisungen nicht mit der Anzahl der Argumente übereinstimmt, gilt Folgendes:
Gibt es mehr Argumente, werden die überzähligen ignoriert.
Gibt es weniger Argumente, führt dies zu undefinierten Ergebnissen.

Im Folgenden wird die Formatanweisung getrennt nach KR-Funktionalität und ANSI-Funktionalität beschrieben.

Formatanweisung (KR-Funktionalität, nur bei C/C++ Versionen kleiner V3.0)

Formatanweisungen können folgendermaßen aufgebaut sein:



1. Jede Formatanweisung muss mit einem Prozentzeichen (%) beginnen.
2. Allgemeine Formatierungszeichen, z.B. zur Steuerung der Vorzeichenausgabe, links- oder rechtsbündigen Ausrichtung, Breite des Ausgabefeldes etc.
3. Zeichen, die die eigentliche Umwandlung festlegen.

Bedeutung der Formatierungszeichen:

- Linksbündige Ausrichtung des Ausgabefeldes.
  Standard: Rechtsbündige Ausrichtung.

+ Das Ergebnis einer Umwandlung mit Vorzeichen wird immer mit Vorzeichen ausgegeben.
  Standard: Nur ein ggf. negatives Vorzeichen wird ausgegeben.

0 Mit Nullen auffüllen.
  Bei allen Umwandlungen wird das Ausgabefeld mit Nullen aufgefüllt.
  Standard: Das Ausgabefeld wird mit Leerzeichen aufgefüllt.
  Die Auffüllung mit Nullen wird auch bei linksbündiger Ausrichtung (Formatierungszeichen −) durchgeführt.

n Minimale Gesamtfeldbreite (inklusive Dezimalpunkt). Falls für die Umwandlung einer Zahl mehr Stellen benötigt werden, hat diese Angabe keine Bedeutung. Ist die Ausgabe kürzer als die angegebene Feldbreite, wird sie bis zur Feldbreite mit Leerzeichen bzw. Nullen aufgefüllt (vgl. Formatierungszeichen − und 0).

* Die Gesamtfeldbreite (siehe *n*) wird statt in der Formatanweisung mit einem Argument festgelegt. Der aktuelle Wert (ganzzahlig) muss unmittelbar vor dem umzuwandelnden Argument oder unmittelbar vor dem Wert der Genauigkeitsangabe (Formatierungszeichen *.m*) in der Argumentenliste stehen (durch ein Komma getrennt).

.m Genauigkeitsangabe.
  e-, f-, g-Umwandlung: Genaue Anzahl der Stellen nach dem Dezimalpunkt (maximal 20). Standard: 6 Stellen.
  s-Umwandlung: Maximale Anzahl der auszugebenden Zeichen. Standard: Alle Zeichen bis zum abschließenden Nullbyte (\0).
  Bei allen anderen Umwandlungen wird die Genauigkeitsangabe ignoriert.

.* Die Genauigkeitsangabe (siehe *.m*) wird statt in der Formatanweisung mit einem Argument festgelegt. Der aktuelle Wert (ganzzahlig) muss unmittelbar vor dem umzuwandelnden Argument in der Argumentenliste stehen (durch ein Komma getrennt).

Bedeutung der Umwandlungszeichen:

l l vor d, o, u, x:
Umwandlung eines Arguments vom Typ `long int`.
Die Angabe ist identisch mit den Großbuchstaben D, O, U, X.

d, o, u, x
Darstellung einer ganzen Zahl (`int`) als
Dezimalzahl mit Vorzeichen (d),
Oktalzahl ohne Vorzeichen (o),
Dezimalzahl ohne Vorzeichen (u),
Sedezimalzahl ohne Vorzeichen (x).

f Darstellung einer Gleitkommazahl (`float` oder `double`) im Format
[−]ddd.ddd
Das Dezimalpunktzeichen wird durch die Lokalität (Kategorie LC_NUMERIC) beeinflusst. Voreingestellt ist der Punkt.
Die Anzahl der Stellen nach dem Dezimalpunkt hängt von der Genauigkeitsangabe in *.m* ab;
Standard (keine Angabe): 6 Stellen.
Bei Genauigkeit 0 erfolgt die Ausgabe ohne Dezimalpunkt.

e Darstellung einer Gleitkommazahl (`float` oder `double`) im Format
[−]d.ddde{+|−}dd.
Das Dezimalpunktzeichen wird durch die Lokalität (Kategorie LC_NUMERIC) beeinflusst. Voreingestellt ist der Punkt.
Die Anzahl der Stellen nach dem Dezimalpunkt hängt von der Genauigkeitsangabe *.m* ab;
Standard (keine Angabe): 6 Stellen.
Bei Genauigkeit 0 wird ein Dezimalpunkt ohne nachfolgende Ziffern ausgegeben.

g Darstellung einer Gleitkommazahl (`float` oder `double`) im f- oder e-Format.
Die Anzahl der Stellen nach dem Dezimalpunkt hängt von der Genauigkeitsangabe *.m* ab.
Es wird jeweils die Darstellung gewählt, die unter Wahrung der Genauigkeit am wenigsten Platz beansprucht.

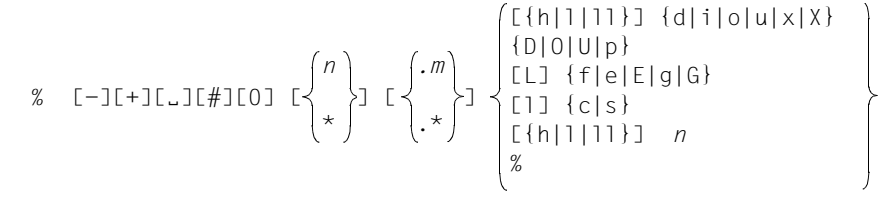
c Format für die Ausgabe eines einzelnen Zeichens (`char`). Das Zeichen '\0' wird ignoriert.

s Format für die Ausgabe von Zeichenketten.
Die Zeichenkette sollte mit '\0' abgeschlossen sein. `printf` schreibt so viele Zeichen der Zeichenkette wie mit der Genauigkeit *.m* angegeben ist.
Standard (keine Angabe): `printf` schreibt alle Zeichen bis '\0'.

% Ausgabe des Zeichens %, keine Umwandlung.

Formatanweisung (ANSI-Funktionalität)

Formatanweisungen können folgendermaßen aufgebaut sein:



1. Jede Formatanweisung muss mit einem Prozentzeichen (%) beginnen.
2. Allgemeine Formatierungszeichen, z.B. zur Steuerung der Vorzeichenausgabe, links- oder rechtsbündigen Ausrichtung, Breite des Ausgabefeldes etc.
3. Zeichen, die die eigentliche Umwandlung festlegen.

Bedeutung der Formatierungszeichen:

- Linksbündige Ausrichtung des Ausgabefeldes.
  Standard: Rechtsbündige Ausrichtung.

+ Das Ergebnis einer Umwandlung mit Vorzeichen wird immer mit Vorzeichen ausgegeben.
  Standard: Nur ein ggf. negatives Vorzeichen wird ausgegeben.

␣ (Leerzeichen)
  Wenn das erste Zeichen einer mit Vorzeichen umzuwandelnden Zeichenfolge kein Vorzeichen ist, wird dem Ergebnis ein Leerzeichen vorangestellt.
  Das Formatierungszeichen ␣ wird ignoriert, wenn gleichzeitig + angegeben wird.

Umwandlung des Ergebnisses in ein alternatives Format.
o-Umwandlung: Die Genauigkeit wird so erhöht, dass die erste Ziffer des Ergebnisses die Ziffer 0 ist.
x- bzw. X-Umwandlung: Einem Ergebnis ungleich 0 wird die Sequenz 0x bzw. 0X vorangestellt.
e-, E-, f-, g- bzw. G-Umwandlung: Das Ergebnis enthält immer einen Dezimalpunkt, auch wenn danach keine weiteren Ziffern folgen (normalerweise enthält das Ergebnis nur einen Dezimalpunkt, wenn danach mindestens eine Ziffer folgt). Außerdem werden bei g- bzw. G-Umwandlung abschließende Nullen nicht weggelassen.
Das Formatierungszeichen # hat keine Wirkung bei c-, s-, d-, i-, u-Umwandlung.

0 Mit Nullen auffüllen.
Bei der Umwandlung von ganzen Zahlen (d, i, o, u, x, X) und Gleitkommazahlen (e, E, f, g, G) wird das Ausgabefeld mit Nullen aufgefüllt.
Standard: Das Ausgabefeld wird mit Leerzeichen aufgefüllt.
0 wird ignoriert, wenn das Formatierungszeichen − oder bei der Umwandlung von ganzen Zahlen eine Genauigkeit *.m* angegeben wird.
Bei c-, p- und s-Umwandlung hat das Formatierungszeichen 0 keine Wirkung.

n Minimale Gesamtfeldbreite (inklusive Dezimalpunkt). Falls für die Umwandlung einer Zahl mehr Stellen benötigt werden, hat diese Angabe keine Bedeutung. Ist die Ausgabe kürzer als die angegebene Feldbreite, wird sie bis zur Feldbreite mit Leerzeichen bzw. Nullen aufgefüllt (vgl. Formatierungszeichen − und 0).

***** Die Gesamtfeldbreite (siehe *n*) wird statt in der Formatanweisung mit einem Argument festgelegt. Der aktuelle Wert (ganzzahlig) muss unmittelbar vor dem umzuwandelnden Argument oder unmittelbar vor dem Wert der Genauigkeitsangabe (Formatierungszeichen *.m*) in der Argumentenliste stehen (durch ein Komma getrennt).

.m Genauigkeitsangabe.
d-, i-, o-, u-, x- bzw. X-Umwandlung: Minimale Anzahl der auszugebenden Ziffern. Standard: 1.
e-, E-, f-Umwandlung: Genaue Anzahl der Stellen nach dem Dezimalpunkt (maximal 20). Standard: 6 Stellen.
g- bzw. G-Umwandlung: Maximale Anzahl der signifikanten Stellen.
s-Umwandlung: Maximale Anzahl der auszugebenden Zeichen. Standard: Alle Zeichen bis zum abschließenden Nullbyte (\0).

.* Die Genauigkeitsangabe (siehe *.m*) wird statt in der Formatanweisung mit einem Argument festgelegt. Der aktuelle Wert (ganzzahlig) muss unmittelbar vor dem umzuwandelnden Argument in der Argumentenliste stehen (durch ein Komma getrennt).

Bedeutung der Umwandlungszeichen:

h h vor d, i, o, u, x, X:
Umwandlung eines Arguments vom Typ `short`.

h vor n:
Das Argument ist vom Typ Zeiger auf `short int` (keine Umwandlung).

l l vor d, i, o, u, x, X:
Umwandlung eines Arguments vom Typ `long`.
l vor d, o, u ist gleichbedeutend mit den Großbuchstaben D, O, U.

l vor c:
Das Argument ist vom Typ `wint_t`.

l vor s:
Das Argument ist vom Typ Zeiger auf `wchar_t`.

l vor n:
Das Argument ist vom Typ Zeiger auf `long int` (keine Umwandlung).

ll ll vor d, i, o, u, x, X :
Umwandlung eines Arguments vom Typ `long long int` bzw. `unsigned long long int`.

ll vor n:
Das Argument ist vom Typ Zeiger auf `long long int`.

L L vor e, E, f, g, G:
Umwandlung eines Arguments vom Typ `long double`.

d, i, o, u, x, X
Darstellung einer ganzen Zahl (`int`) als
Dezimalzahl mit Vorzeichen (d, i),
Oktalzahl ohne Vorzeichen (o),
Dezimalzahl ohne Vorzeichen (u),
Sedezimalzahl ohne Vorzeichen (x, X). Bei x werden die Kleinbuchstaben abcdef benutzt, bei X die Großbuchstaben ABCDEF.
Die Genauigkeitsangabe *.m* gibt die minimale Anzahl der auszugebenden Ziffern an. Ist der Wert mit weniger Ziffern darstellbar, wird das Ergebnis mit führenden Nullen aufgefüllt. Standard ist Genauigkeit 1. Bei Genauigkeit 0 und Wert 0 erfolgt keine Ausgabe.

f Darstellung einer Gleitkommazahl (`float` oder `double`) im Format [−]ddd.ddd
Das Dezimalpunktzeichen wird durch die Lokalität (Kategorie LC_NUMERIC) beeinflusst. Voreingestellt ist der Punkt.
Die Anzahl der Stellen nach dem Dezimalpunkt hängt von der Genauigkeitsangabe

in *.m* ab;
Standard (keine Angabe): 6 Stellen.
Bei Genauigkeit 0 erfolgt die Ausgabe ohne Dezimalpunkt.

e, E Darstellung einer Gleitkommazahl (`float` oder `double`) im Format [−]d.ddde{+|−}dd.
Das Dezimalpunktzeichen wird durch die Lokalität (Kategorie LC_NUMERIC) beeinflusst. Voreingestellt ist der Punkt.
Bei E-Umwandlung wird dem Exponenten der Großbuchstabe E vorangestellt.
Die Anzahl der Stellen nach dem Dezimalpunkt hängt von der Genauigkeitsangabe *.m* ab;
Standard (keine Angabe): 6 Stellen.
Bei Genauigkeit 0 erfolgt die Ausgabe ohne Dezimalpunkt.

g, G Darstellung einer Gleitkommazahl (`float` oder `double`) im f- oder e-Format (bzw. bei G-Umwandlung im E-Format).
Die Anzahl der signifikanten Stellen hängt von der Genauigkeitsangabe *.m* ab.
Das e- bzw. E-Format wird nur dann verwendet, wenn der Exponent des Umwandlungsergebnisses kleiner -4 oder größer als die angegebene Genauigkeit ist.

c Ohne Umwandlungszeichen `l`: Das Argument vom Typ `int` wird umgewandelt in den Typ `unsigned char`, und das entsprechende Zeichen wird geschrieben.
Mit Umwandlungszeichen `l`:Das Argument vom Typ `wint_t` wird umgewandelt wie bei `ls` ohne Angabe einer Genauigkeit, wobei das Argument ein Zeiger auf ein 2-elementiges Feld ist, dessen erstes Element das `wint_t`-Argument enthält und das zweite das Null-Langzeichen.

p Umwandlung eines Arguments vom Typ Zeiger auf `void`.
Die Ausgabe erfolgt als 8stellige Sedezimalzahl (analog zu der Angabe %08.8x).

s Ohne Umwandlungszeichen `l`: Das Argument ist ein Zeiger auf eine Zeichenkette vom Typ `char`. Die Zeichenkette sollte mit '\0' abgeschlossen sein. `printf` schreibt so viele Zeichen der Zeichenkette wie mit der Genauigkeit *.m* angegeben ist.
Standard (keine Angabe): `printf` schreibt alle Zeichen bis '\0'.
Mit Umwandlungszeichen `l`:Das Argument ist vom Typ Zeiger auf eine Langzeichenkette (`wchar_t`). Alle Langzeichen bis zum abschließenden Null-Langzeichen inklusive werden in Multibyte-Zeichen konvertiert (wie durch `wcrtomb`; das zugehörige `mbstate_t`-Objekt wird vor Konvertierung des ersten Langzeichens mit Null initialisiert). Die entsprechenden Multibyte-Zeichen werden bis zum Null-Langzeichen (exklusive) geschrieben. `printf` schreibt so viele Zeichen der Zeichenkette (inklusive Umschaltsequenzen), wie mit der Genauigkeit *.m* angegeben ist.
Standard (keine Angabe): `printf` schreibt alle Zeichen bis '\0'.
Ein unvollständiges Multibyte-Zeichen wird in keinem Fall geschrieben.

n Es findet keine Umwandlung und Ausgabe des Arguments statt. Das Argument ist vom Typ Zeiger auf `int`. Dieser ganzzahligen Variablen wird die Anzahl der Zeichen zugewiesen, die printf bis zu diesem Zeitpunkt für die Ausgabe erzeugt hat.

% Ausgabe des Zeichens %, keine Umwandlung.

Returnwert Anzahl der ausgegebenen Zeichen
bei Erfolg.

Integer< 0 bei Fehler.

Hinweise Bei der Umwandlung von Gleitkommazahlen rundet `printf` auf die angegebene Genauigkeit.

`printf` nimmt keine Konvertierung von einem Datentyp in einen anderen vor. Soll ein Wert nicht entsprechend seinem Typ ausgegeben werden, muss er explizit konvertiert werden (z.B. mit dem cast-Operator).

Die Daten werden nicht sofort in die externe Datei geschrieben, sondern in einem C-internen Puffer zwischengespeichert (siehe Abschnitt „Pufferung“ auf Seite 65).

Maximale Anzahl der auszugebenden Zeichen:
Bei KR-Funktionalität (nur bei C/C++ Versionen kleiner V3.0 vorhanden) können pro `printf`-Aufruf maximal 1400 Zeichen ausgegeben werden,
bei ANSI-Funktionalität maximal 1400 Zeichen pro Konversionselement (z.B. %s).

Versuche, nicht initialisierte Variablen oder Variablen nicht entsprechend ihrem Datentyp auszugeben, können zu undefinierten Ergebnissen führen.

Das Verhalten ist undefiniert, wenn in einer Formatanweisung dem Prozentzeichen (%) ein nicht definiertes Formatierungs- bzw. Umwandlungszeichen folgt.

`printf` arbeitet wie `fprintf`, nur dass die Daten auf die Standardausgabe und nicht in eine Datei geschrieben werden.

Beispiel 1 Ausgabe des Datums und der Uhrzeit in der Form

```
Donnerstag, den 14. Februar, 12:05 Uhr
```

Die Argumente *wochentag* und *monat* sind Zeiger auf Zeichenketten, die mit ’\0’ abgeschlossen sind.

```
printf("%s, den %d. %s, %02d:%02d Uhr\n", wochentag, tag, monat, std, min);
```

Beispiel 2 Ausgabe der Zahl pi auf 5 Nachkommastellen.

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

Beispiel 3 Die gängigsten `printf`-Formate erklären sich durch ihre Anwendung in den übrigen Beispielprogrammen von selbst. In folgender Tabelle finden Sie einige weitere Formatangaben einschließlich ihrer Wirkung aufgelistet.
Zur Verdeutlichung ist das umgewandelte Ergebnis in > < eingerahmt.

Formatangabe	Argument(e)	Ergebnis
%.6s	"Konstanz"	>Konsta<
%10.5s	"Konstanz"	> Konst<
%-10.5s	"Konstanz"	>Konst <
%15.15s	"Konstanz"	> Konstanz<
%*.*s	20,7,"Konstanz"	> Konstan<
%-*.*s	15,10,"Konstanz"	>Konstanz <
%8d	721932	> 721932<
%-8d	721932	>721932 <
%-*.*f	3,2,27.31928	>27.32<
%-0*.*f	1,12,19.84	>19.840000000000<
%04.*f	12,10.60	>10.600000000000<
%-0*.*g	1,12,19.84	>19.84<
%e	1712.1961	>1.712196e+03<
%.10e	1712.1961	>1.7121961000e+03<
%10.10e	1712.1961	>1.7121961000e+03<

Siehe auch fprintf, sprintf, snprintf, putc, putchar, puts, scanf, fscanf

putc - Zeichen in eine Datei schreiben

Definition

```
#include <stdio.h>

int putc(int c, FILE *dz);
```

`putc` schreibt das Zeichen *c* in die Datei mit Dateizeiger *dz* an die aktuelle Lese-/Schreibposition.

Returnwert

Geschriebenes Zeichen *c*	bei Erfolg.
EOF	sonst.

Hinweise

`putc` ist sowohl als Makro als auch als Funktion realisiert (siehe Abschnitt „Funktionen und Makros“ auf Seite 19).

Die Zeichen werden nicht sofort in die externe Datei geschrieben, sondern in einem C-internen Puffer zwischengespeichert (siehe Abschnitt „Pufferung“ auf Seite 65).

Bei der Ausgabe in Textdateien werden die Steuerzeichen für Zwischenraum (\n, \t, etc.) je nach Art der Textdatei in ihre entsprechende Wirkung umgesetzt (siehe Abschnitt „Zwischenraum“ auf Seite 68).

Beispiel

Folgendes Programm liest Zeichen von der Standardeingabe ein und schreibt sie in die Datei *dat*.

```
#include <stdio.h>

FILE *fp;
int c;

int main(void)
{
  fp = fopen("dat","w");
  while((c=getchar()) != EOF)
        putc((char)c,fp);
  fclose(fp);
  return 0;
}
```

Siehe auch

fputc, printf, putchar, fopen, fopen64, putwc

putchar - Zeichen auf Standardausgabe ausgeben

Definition

```
#include <stdio.h>

int putchar(int c);
```

`putchar` schreibt das Zeichen *c* auf die Standardausgabe.

Returnwert

Geschriebenes Zeichen *c*	bei Erfolg.
EOF	sonst.

Hinweise

`putchar` ist sowohl als Makro als auch als Funktion realisiert (siehe Abschnitt „Funktionen und Makros“ auf Seite 19).

Die Zeichen werden nicht sofort in die externe Datei geschrieben, sondern in einem C-internen Puffer zwischengespeichert (siehe Abschnitt „Pufferung“ auf Seite 65).

Weitere Informationen zur Ausgabe in Textdateien, v.a. zur Umsetzung der Steuerzeichen für Zwischenraum (\n, \t, etc.), finden Sie in Abschnitt „Zwischenraum“ auf Seite 68.

Siehe auch putc, fputc, putwchar

putenv - Umgebungsvariable ändern oder hinzufügen

Definition

```
#include <stdlib.h>

int putenv (const char *string);
```

`putenv()` ändert den Wert einer vorhandenen Umgebungsvariablen oder definiert eine neue Umgebungsvariable. *string* muß auf eine Zeichenkette der Form "*name*=*value*" zeigen. *name* steht für den Namen einer Umgebungsvariablen, *value* für den ihr zugewiesenen Wert. Wenn *name* mit einer existierenden Umgebungsvariablen identisch ist, wird der zugehörige Wert *value* mit der neuen Angabe überschrieben. Wenn *name* eine neue Umgebungsvariable ist, wird die Umgebung um diese erweitert. In jedem Fall wird *string* Teil der Umgebung und ändert damit die Umgebung.

Der von *string* belegte Speicherplatz wird nicht mehr verwendet, wenn `putenv()` einer vorhandenen Umgebungsvariablen einen neuen Wert zuweist.

Returnwert

0	bei Erfolg.
≠ 0	bei Fehler, z.B. wenn nicht genügend Speicherplatz vorhanden ist. `errno` wird gesetzt, um den Fehler anzuzeigen.

Fehler

`putenv()` schlägt fehl, wenn gilt:

`ENOMEM`	Es steht nicht genügend Speicherplatz zur Verfügung.

Hinweise

`putenv()` verändert die Umgebung, auf die `environ` zeigt, und kann in Verbindung mit `getenv()` verwendet werden.

`putenv()` kann `malloc()` verwenden, um die Umgebung zu vergrößern.

Eine mögliche Fehlerquelle ist der Aufruf von `putenv()` mit einer automatischen Variablen als Argument und einer anschließenden Rückkehr von der aufrufenden Funktion, während *string* noch immer Teil der Umgebung ist.

Beim Start eines Programms wird neben den Voreinstellungen für die Umgebung auch die S-Variable `SYSPOSIX` als Umgebungsdefinition ausgewertet (siehe Abschnitt „Umgebungsvariablen“ auf Seite 118). `putenv()` verändert die S-Variable jedoch nicht, sondern modifiziert die Umgebung nur für den aktuellen Programmlauf.

Siehe auch

environ, getenv, malloc, stdlib.h.

puts - Zeichenkette auf Standardausgabe ausgeben

Definition

#include <stdio.h>

int puts(const char *s);

`puts` schreibt die Zeichenkette *s* auf die Standardausgabe `stdout` und fügt ein abschließendes Neue-Zeile-Zeichen hinzu.
Die Zeichenkette *s* muss mit einem Nullbyte (\0) abgeschlossen sein.

Returnwert

0	bei Erfolg
EOF	sonst

Hinweise

Im Gegensatz zu `fputs` schließt `puts` die Ausgabe automatisch mit einem Neue-Zeile-Zeichen ab. Enthält die auszugebende Zeichenkette bereits ein abschließendes Neue-Zeile-Zeichen (z.B. ein Satz in SAM- oder ISAM-Dateien), führt dies zu einer zusätzlichen Leerzeile bei der Ausgabe.

Das abschließende Nullbyte von *s* wird nicht mitausgegeben.

Weitere Informationen zur Ausgabe in Textdateien, v.a. zur Umsetzung der Steuerzeichen für Zwischenraum (\n, \t, etc.), finden Sie im Abschnitt „Zwischenraum“ auf Seite 68.

Beispiel

Dieses Beispiel zeigt den unterschiedlichen Abschluss der Ausgabe bei `puts` und `fputs`.

```
#include <stdio.h>
int main(void)
{
 FILE *fp;
 char s[BUFSIZ];
 fp=fopen("dat","w");
 while(gets(s) != NULL)
      {
        fputs(s,fp);
        puts(s);
      }
 return 0;
}
```

Wenn Sie nach Ablauf dieses Programmes *dat* anschauen, stellen Sie fest, dass die Zeichenketten aus der Eingabe (`gets` löscht ein ggf. vorhandenes Neue-Zeile-Zeichen) von `fputs` hintereinander und nicht zeilenweise geschrieben wurden. Die Ausgabe durch `puts` erfolgt dagegen zeilenweise, da in jede eingelesene Zeichenkette automatisch ein Neue-Zeile-Zeichen angefügt wird.

Siehe auch

fputs, gets, fgets, putws, sprintf, snprintf

putw - Wortweise in eine Datei schreiben

Definition

#include <stdio.h>

int putw(int w, FILE *dz)

`putw` schreibt das Maschinenwort *w* in die Datei mit Dateizeiger *dz* an die aktuelle Lese-/Schreibposition.

Returnwert

Das geschriebene Wort *w*	bei Erfolg.
EOF	sonst.

Hinweise

Weil Wortlänge und Anordnung der Bytes systemabhängig sind, können unter Umständen Dateien, die mit `putw` auf einem Betriebssystem ungleich BS2000 beschrieben wurden, nicht mit `getw` im BS2000 gelesen werden.

Da `putw` Fehler nicht explizit anzeigt (-1 ist ein gültiger Integerwert), sollten Sie zusätzlich `ferror` verwenden, um abzuprüfen, ob vor oder nach dem Schreiben ein Fehler auftrat.

Die Zeichen werden nicht sofort in die externe Datei geschrieben, sondern in einem C-internen Puffer zwischengespeichert (siehe Abschnitt „Pufferung“ auf Seite 65).

Bei der Ausgabe in Textdateien werden die Steuerzeichen für Zwischenraum (\n, \t, etc.) je nach Art der Textdatei in ihre entsprechende Wirkung umgesetzt (siehe Abschnitt „Zwischenraum“ auf Seite 68).

Beispiel

Folgendes Programm überträgt den Inhalt der Datei *input* wortweise auf Datei *output*.

```
#include <stdio.h>

FILE *fp_in, *fp_out; int w;
int main(void)
{
  fp_in = fopen("input","r");
  fp_out = fopen("output","w");
  while(!feof(fp_in) && !ferror(fp_in) && !ferror(fp_out))
       {
         w = getw(fp_in);
         putw(w,fp_out);
       }
  fclose(fp_in); fclose(fp_out);
  return 0;
}
```

Siehe auch

getw

putwc - Langzeichen in Datei schreiben

Definition

```
#include <wchar.h>
#include <stdio.h>

wint_t putwc(wchar_t wc, FILE *dz);
```

`putwc` entspricht der Funktion `fputwc` mit folgendem Unterschied: Wenn sie als Makro implementiert ist, kann sie *dz* mehrmals auswerten. *dz* sollte daher niemals ein Ausdruck mit Seiteneffekten sein.
Es wird empfohlen, statt `putwc`, insbesondere statt `putwc(`*wc, *f*`++)`, die Funktion `fputwc` zu verwenden.

Beschreibung siehe `fputwc`.

putwchar - Langzeichen auf Standardausgabe schreiben

Definition

```
#include <wchar.h>

wint_t putwchar(wchar_t wc);
```

Der Funktionsaufruf `putwchar(`*wc*`)` entspricht dem von `putwc(`*wc*`, stdout)`.
Beschreibung siehe `fputwc`.

qsort - Datenfeld sortieren (Quicksort)

Definition

```
#include <stdlib.h>

void qsort(void *feld, size_t anz, size_t elgroesse,
           int (*vergl)(const void *, const void *);
```

`qsort` sortiert *anz* Elemente eines Vektors *feld* nach dem Quicksort-Algorithmus. Jedes Vektorelement ist *elgroesse* Bytes lang.
Um das Feld sortieren zu können, benötigt `qsort` eine vom Benutzer bereitzustellende Funktion *vergl*, die zwei Elemente miteinander vergleicht.

Parameter

void *feld
: Zeiger auf das erste Element des zu sortierenden Vektors.

size_t anz
: Anzahl der zu sortierenden Elemente.

size_t elgroesse
: Größe eines Elementes in Bytes.

int (*vergl)(const void *, const void *)
: Zeiger auf eine Funktion, die zwei Elemente vergleicht und eine ganze Zahl als Ergebnis liefert. Das Ergebnis wird wie folgt gedeutet:

< 0	Argument1 ist kleiner als Argument2
= 0	Argument1 und Argument2 sind gleich
> 0	Argument1 ist größer als Argument2

Die Funktion hat zwei Parameter, und zwar zwei Zeiger auf den Typ der Vektorelemente.

Die Funktion kann etwa wie folgt definiert sein:

Beispiel 1

```
/*Vergleicht zwei char-Werte*/
int comp(const void *a, const void *b)
   {
     if(*((const char *)a) < *((const char *) b) )
       return(-1);
     else if(*((const char *)a) > *((const char *) b ) )
            return(1);
     return(0);
   }
```

Beispiel 2

```
/*Vergleicht zwei integer-Werte*/
int compare(const void *a, const void *b)
   {
     return  ( *((const int *) a) - *((const int *) b) );
   }
```

Hinweis Die Reihenfolge von Vektorelementen, für die die Vergleichsfunktion Gleichheit feststellt, wird nicht verändert.

Beispiel Folgendes Programm sortiert eine Zahlenfeld und gibt es auf Standardausgabe aus.

```
#include <stdio.h>
#include <stdlib.h>

int comp   (const void *s, const void *t)
{
    return  ( *((const int *) s) - *((const int *) t) );
}

int main(void)
{
  int j;
  static int array[] = {4,7,2,1,54,9,2,3,1,23};

  qsort (array, 10, sizeof(int), comp);

  for (j=0; j<10; j++)
     printf("%d\n", array[j]);
  return 0;
}
```

Siehe auch bsearch

raise - Signal an eigenes Programm senden

Definition

```
#include <signal.h>

int raise(int sig);
```

`raise` sendet das Signal *sig* an das eigene Programm.

Mit `raise` lassen sich STXIT-Ereignisse simulieren sowie STXIT-unabhängige Signale senden (selbst definierte und vom C-Laufzeitsystem vordefinierte).

Parameter

int sig

Signal, das an das eigene Programm geschickt werden soll. Für *sig* können symbolische Konstanten eingesetzt werden, die in der folgenden Übersicht unter „SIGNR“ aufgelistet sind. Diese Konstanten sind in der Include-Datei <signal.h> definiert.

SIGNR	STXIT-Klasse	Bedeutung
SIGHUP	ABEND	Abbruch der Dialogstationsleitung
SIGINT	ESCPBRK	Unterbrechung von der Dialogstation (K2)
SIGILL	PROCHK	Ausführung einer ungültigen Instruktion
SIGABRT	-	raise-Signal für Programmbeendigung mit `_exit(-1)`
SIGFPE	PROCHK	fehlerhafte Gleitkommaoperation
SIGKILL	-	raise-Signal für Programmbeendigung mit `exit(-1)`
SIGSEGV	ERROR	Speicherzugriff mit unerlaubtem Segmentzugriff
SIGALRM	RTIMER	ein Zeitintervall ist abgelaufen (Realzeit)
SIGTERM	TERM	Signal bei Programmbeendigung
SIGUSR1	-	vom Benutzer definiert
SIGUSR2	-	vom Benutzer definiert
SIGDVZ	PROCHK	Division durch 0
SIGXCPU	RUNOUT	CPU-Zeit ist aufgebraucht
SIGBPT +	SVC	Haltepunkt (derzeit nicht unterstützt)
SIGTIM	TIMER	ein Zeit-Intervall ist abgelaufen (CPU-Zeit)
SIGINTR	INTR	SEND-MESSAGE-Kommando
SIGSVC +	SVC	SVC-Aufruf (derzeit nicht unterstützt)

Die mit + gekennzeichneten Signale werden derzeit nicht unterstützt.

Returnwert

0 Das Signal wurde erfolgreich geschickt.

-1 Das Signal konnte nicht gesendet werden, weil *sig* keine gültige Signalnummer ist. Zusätzlich wird `errno` auf EINVAL gesetzt (ungültige Signalnummer).

Hinweise Mit Ausnahme von SIGKILL können die `raise`-Signale mit der Funktion `signal` abgefangen werden. Ausführliche Informationen hierzu finden Sie bei `signal`.

Wenn das Programm keine Behandlung von `raise`-Signalen vorsieht, wird das Programm bei Eintritt eines Signals mit `exit(-1)` beendet und es werden folgende Meldungen ausgegeben:

```
"CCM0101 signal occured: signal"
"CCM0999 Exit -1"
```

Signal SIGABRT:
SIGABRT führt zu einer Programmbeendigung mit `_exit(-1)`. Im Unterschied zu `exit(-1)` werden die mit `atexit` registrierten Beendigungsroutinen nicht aufgerufen und geöffnete Dateien nicht geschlossen.

Signal SIGKILL:
SIGKILL führt zu einer Programmbeendigung mit `exit(-1)`. Im Unterschied zu SIGABRT kann SIGKILL nicht abgefangen werden, d.h. `signal`-Aufrufe, die als Argument den Namen einer selbst geschriebenen Funktion oder SIG_IGN angeben, sind für SIGKILL nicht zulässig.

Beispiel Ein Programm, das sich selbst abbricht.

```
#include <signal.h>

int main(void)
{
  for(;;)
    raise(SIGKILL);
  return 0;
}
```

Siehe auch alarm, atexit, exit, _exit, signal

rand - Zufallsgenerator

Definition #include <stdlib.h>

int rand(void);

`rand` liefert eine positive ganze Zufallszahl aus dem Bereich [0, 2^{15}-1].

Ein `rand`-Aufruf wählt Werte aus einer Folge von Pseudo-Zufallszahlen aus, unter Verwendung eines multiplikativen, kongruenten Zufallsgenerators. Der Generator hat eine Periode von 2^{32}.

Returnwert Zufallszahl aus [0, 2^{15}-1]

Hinweis Der Zufallsgenerator lässt sich mit `srand` initialisieren bzw. rücksetzen. Unterbleibt die Initialisierung, beginnt der Zufallsgenerator mit seinem voreingestellten Wert, wie bei `srand(1)`.

Beispiel 1 Generiere zweimal dieselben fünf Zufallszahlen:

```
#include <stdlib.h>
#include <stdio.h>

int i;

int main(void)
{
   for(i=1; i <= 10; ++i)
      {
        printf("%d\n", rand());
        if(i == 5)
          srand(1);
      }
   return 0;
}
```

Beispiel 2 Simulation eines Würfels.

```
#include <stdio.h>
#include <stdlib.h>
#define A 32767              /* 2**15 - 1 */

int cpu_t;                   /* Abfragevariable fuer die CPU-Zeit */
int i,x;

int main(void)
{
cpu_t = cputime();
srand(cpu_t);                /* Startwert fuer Zufallsgenerator       */
for(i=1; i<= 6; ++i)         /* Simulation von sechs mal wuerfeln     */
  {
  x =  rand()/(A/6)+1;       /* Zufallszahl im Bereich 1-6 ermitteln  */
  printf("Augenzahl = %d\n",x);
  }
return 0;
}
```

Siehe auch srand

read - Aus einer Datei einlesen (elementar)

Definition

```
#include <stdio.h>

int read(int dk, char *puf, int anz);
```

`read` ist die elementare Einleseoperation.

`read` liest aus der Datei mit Dateikennzahl *dk* maximal *anz* Zeichen in den Bereich, auf den *puf* zeigt.

Bei Textdateien liest `read` pro Aufruf nur die Zeichen innerhalb einer Zeile. Ein Zeilenende beendet die Eingabe.
Bei Binärdateien liest `read` über Neue-Zeile-Zeichen (\n) hinweg.
SAM-Dateien werden mit elementaren Funktionen stets als Textdateien verarbeitet.

Parameter

int dk
: Dateikennzahl der Eingabedatei.
Eine Dateikennzahl (positive ganze Zahl) ist das Ergebnis eines erfolgreichen Aufrufs von `open`/`open64` oder `creat`/`creat64`.
Die Dateikennzahlen für `stdin` (0), `stdout` (1) und `stderr` (2) sind nach Programmstart automatisch zugeordnet.

char *puf
: Zeiger auf den Bereich, in den die gelesenen Daten geschrieben werden sollen. Er sollte mindestens *anz* Bytes groß sein.

int anz
: Maximale Anzahl der Bytes, die gelesen werden sollen. Wird vorher das Zeilenende erreicht, werden weniger als *anz* Bytes eingelesen.

Returnwert

Anzahl der tatsächlich gelesenen Bytes	bei Erfolg.
0	bei Dateiende.
-1	`read` hat nichts gelesen, weil einer der folgenden Fehler vorliegt: – physikalischer Ein-/Ausgabefehler – *dk* ist keine gültige Dateikennzahl – die Datei ist nicht vorhanden – es besteht kein Zugriffsrecht auf die Datei – *anz* ist nicht möglich

Hinweise Die Anzahl der tatsächlich gelesenen Bytes kann kleiner sein als die Angabe in *anz*, wenn vorher Zeilenende erreicht wird (nur bei Textdateien), sowie bei Dateiende oder Fehler.

Um sicherzugehen, dass nicht mehr Bytes gelesen werden, als der Puffer aufnehmen kann, sollten Sie `sizeof` verwenden.

Beispiel Folgendes Programm kopiert die Standardeingabe (Dateikennzahl 0) auf die Standardausgabe (Dateikennzahl 1). Wenn Sie den Umlenkmechanismus für `stdin` und `stdout` ausnutzen (PARAMETER-PROMPTING in der RUNTIME-Option), können Sie damit von einer beliebigen Quelle auf ein beliebiges Ziel kopieren. BUFSIZ (8192 Bytes) ist in der Include-Datei <stdio.h> definiert.

```
#include <stdio.h>

int main(void)
{
  char buf[BUFSIZ];
  int n;

  while((n = read(0, buf, sizeof(buf))) > 0)
        write(1, buf, n);
  return 0;
}
```

Siehe auch write, open, open64, creat, creat64

realloc - Speicherplatz verändern

Definition

```
#include <stdlib.h>

void *realloc(void *zg, size_t anz);
```

`realloc` verändert die Größe des Speicherbereiches, auf den *zg* zeigt, in *anz* Bytes. Der Speicherbereich muss zuvor durch `malloc` oder `calloc` zugewiesen worden sein.

`realloc` ist Teil eines C-spezifischen Speicherverwaltungspaketes, das angeforderte und wieder freigegebene Speicherbereiche intern verwaltet. Neue Anforderungen werden zuerst aus bereits verwalteten Bereichen zu erfüllen versucht, dann erst vom Betriebssystem (vgl. Funktion `garbcoll`).

Returnwert Zeiger auf den Anfang des geänderten Speicherbereiches, bei Erfolg.

NULL-Zeiger falls `realloc` den Speicherplatz nicht verändern konnte, z.B. weil der noch vorhandene Speicherplatz nicht ausreicht oder ein Fehler auftrat.

Hinweise Wenn `realloc` die Größe eines Speicherbereiches ändert, kann u.U. der zugewiesene Block verschoben sein. In solchen Fällen ist der Inhalt des als Argument übergebenen Zeigers nicht identisch mit dem Rückgabewert.
Der Inhalt des Blocks bleibt bis zum Minimum der alten (beim Vergrößern) bzw. neuen Größe (beim Verkleinern) erhalten.

Liefert `realloc` den NULL-Zeiger, kann evtl. der Block, auf den *zg* zeigt, zerstört worden sein!

Ist *zg* ein NULL-Zeiger, funktioniert `realloc` wie ein `malloc`-Aufruf für die angegebene Größe.

Falls *anz* den Wert 0 besitzt, liefert `realloc` eine eindeutige Adresse, die auch an `free` übergeben werden kann.

Beispiel Folgender Programmausschnitt fordert zunächst Speicherplatz für 20 Zeichen an und erweitert diesen Bereich dann für die Aufnahme weiterer 80 Zeichen (also insgesamt auf 100 Bytes).

```
#include <stdlib.h>

char *char_array;
char_array = (char *)malloc(20 * sizeof(char));
     .
     .
char_array = (char *)realloc(char_array, 100 * sizeof(char));
```

Siehe auch malloc, calloc, free, garbcoll

remove - Datei löschen

Definition

```
#include <stdio.h>

int remove(const char *d_name);
```

`remove` löscht die Datei *d_name*. *d_name* kann ein voll- oder teilqualifizierter Dateiname sein.

Returnwert

0	bei Erfolg.
-1	wenn die Datei nicht gelöscht werden kann, z.B. wenn keine Datei mit dem Namen *d_name* existiert oder die Datei durch eine andere Task geöffnet ist. Zusätzlich wird `errno` auf EDMS gesetzt.

Hinweise

Wird ein teilqualifizierter Dateiname angegeben, löscht `remove` alle entsprechenden Dateien ohne vorherige Abfrage (Y/N). Es wird von der Antwort „Y“ ausgegangen.

`remove` löscht die Dateien nur logisch, d.h. der Katalogeintrag wird gelöscht und der zugewiesene Speicherplatz freigegeben.

Wenn eine Datei durch irgendein Programm geöffnet ist, wird sie nicht gelöscht.

Satz-E/A

`remove` ist auch auf Dateien mit Satz-E/A unverändert anwendbar.

rename - Datei umbenennen

Definition

```
#include <stdio.h>

int rename(const char *name_alt, const char *name_neu);
```

`rename` benennt die Datei mit dem Namen *name_alt* auf den neuen Namen *name_neu* um.

Returnwert

0	bei Erfolg.
-1	wenn die Datei nicht umbenannt werden konnte. Wenn z.B. – keine Datei mit dem Namen *name_alt* exisitiert, – bereits eine Datei unter dem Namen *name_neu* katalogisiert ist oder – die umzubenennende Datei durch ein Programm geöffnet ist. Zusätzlich wird `errno` auf EMACRO gesetzt.

Satz-E/A `rename` ist auch auf Dateien mit Satz-E/A unverändert anwendbar.

rewind - Lese-/Schreibzeiger auf Dateianfang positionieren

Definition

```
#include <stdio.h>

void rewind(FILE *dz);
```

`rewind` positioniert den Lese-/Schreibzeiger der Datei mit Dateizeiger *dz* auf Dateianfang.

Hinweise

Die Aufrufe `rewind(dz)` und `fseek(dz,0L,SEEK_SET)` bzw. `fseek64(dz,0LL,SEEK_SET)` sind äquivalent, außer dass `rewind` kein Ergebnis zurückliefert.

Auf Systemdateien kann nicht positioniert werden (SYSDTA, SYSOUT, SYSLST).

Werden in eine Textdatei neue Sätze geschrieben (geöffnet zum Neuerstellen oder Anhängen) und erfolgt ein `rewind`-Aufruf, dann werden zunächst ggf. restliche Daten aus dem Puffer in die Datei geschrieben und mit Zeilenende (\n) abgeschlossen.
Ausnahme bei ANSI-Funktionalität:
Wenn die Daten einer ISAM-Datei im Puffer nicht mit einem Neue-Zeile-Zeichen enden, bewirkt `rewind` keinen Zeilenwechsel (bzw. Satzwechsel). D.h., die Daten werden beim Schreiben aus dem Puffer nicht automatisch mit einem Neue-Zeile-Zeichen abgeschlossen. Nachfolgende Daten verlängern den Satz in der Datei. Beim Lesen einer ISAM-Datei werden daher nur Neue-Zeile-Zeichen eingelesen, die vom Programm explizit geschrieben wurden.

Ein erfolgreicher Aufruf der Funktion `rewind` löscht das EOF-Flag der Datei und hebt alle Effekte der vorangegangenen `ungetc`-Aufrufe für diese Datei auf.

Satz-E/A

`rewind` ist auch auf Dateien mit Satz-E/A unverändert anwendbar.

Beispiel Folgendes Programm verarbeitet eine Datei zuerst ab dem 11. Zeichen bis Dateiende und anschließend ab Dateianfang (funktioniert nur mit Binärdateien, in diesem Fall nur mit SAM- und PAM-Dateien).

```
#include <stdio.h>

int main(void)
{
   FILE *fp;
   int c,i;

   fp = fopen("input","rb");
               /* die ersten 10 Zeichen überlesen */
   fseek(fp,10L,SEEK_SET);
   while((c=getc(fp)) != EOF)
         putc((char)c,stdout);
               /* auf Dateianfang positionieren */
   rewind(fp);
   for(i=0; i<10; i++)
      {
        c=getc(fp);
        putc((char)c,stdout);
      }
   fclose(fp);
   return 0;
}
```

Siehe auch fseek, fseek64, fsetpos, fsetpos64

rindex - Letztes Vorkommen eines Zeichens in einer Zeichenkette

Definition #include <string.h>

char *rindex(const char *s, int c);

`rindex` sucht das letzte Vorkommen des Zeichens *c* in der Zeichenkette *s* und liefert bei Erfolg einen Zeiger auf die gesuchte Position in *s*.

Das abschließende Nullbyte (\0) wird als Zeichen mitberücksichtigt.

Returnwert Zeiger auf die Position von *c* in der Zeichenkette *s* bei Erfolg.

NULL-Zeiger wenn *c* in der Zeichenkette *s* nicht enthalten ist.

Hinweis Die Funktionen `rindex` und `strrchr` sind äquivalent.

Beispiel Finde das letzte 'k':

```
#include <string.h>
#include <stdio.h>

int main(void)
{
   char *s = "was fuer ein Spass im kkuehlen Nass!";
   printf("%s\n", s);
   printf("Wo steckt der Fehler? %s\n", rindex(s, 'k'));
   return 0;
}
```

Siehe auch index, strchr, strrchr

rint, rintf, rintl - auf nächste ganze Zahl runden

Definition

```
#include <math.h>

double rint(double x);

float rintf (float x);

long double rintl (long double x);
```

Die Funktionen geben in Gleitpunktdarstellung jeweils die ganze Zahl zurück, die x am nächsten liegt.
`rint` stellt das Ergebnis dar als Zahl vom Typ `double`, `rintf` als Zahl vom Typ `float` und `rintl` als Zahl vom Typ `long double`.

Der zurückgegebene Wert ist entsprechend dem aktuell gesetzten Rundungsmodus des Rechners gerundet. Wenn der Rundungsmodus 'round-to-nearest' gesetzt ist und die Differenz zwischen x und dem gerundeten Ergebnis genau 0.5 ist, wird die nächste gerade Ganzzahl zurückgegeben.

Wenn der aktuell eingestellte Rundungsmodus in Richtung positiv unendlich rundet, ist `rint` identisch zu `ceil`. Wenn der aktuell eingestellte Rundungsmodus in Richtung negativ unendlich rundet, ist `rint` identisch zu `floor`.
In dieser Version ist der Rundungsmodus fest auf Richtung positiv unendlich eingestellt.

Returnwert

ganze Zahl	dargestellt als Zahl vom Typ `double`, `float` bzw. `long double` bei Erfolg.
HUGE_VAL	bei Überlauf. `errno` wird auf ERANGE gesetzt, um den Fehler anzuzeigen.

Hinweis In dieser Version ist der Rundungsmodus fest auf Richtung positiv unendlich eingestellt.

Siehe auch abs, ceil, floor, llrint, llround, lrint, lround, round

round, roundf, roundl - auf nächste ganze Zahl runden

Definition

```
#include <math.h>

double round(double x);

float roundf (float x);

long double roundl (long double x);
```

Die Funktionen geben in Gleitpunktdarstellung jeweils die ganze Zahl zurück, die x am nächsten liegt.
`round` stellt das Ergebnis dar als Zahl vom Typ `double`, `roundf` als Zahl vom Typ `float` und `roundl` als Zahl vom Typ `long double`.

Der zurückgegebene Wert ist unabhängig vom eingestellten Rundungsmodus. Wenn die Differenz zwischen x und dem gerundeten Ergebnis genau 0.5 ist, wird die betragsmäßig größere ganze Zahl zurückgegeben.

Returnwert

ganze Zahl	dargestellt als Zahl vom Typ `double`, `float` bzw. `long double` bei Erfolg.
undefiniert	bei Über- oder Unterlauf. `errno` wird auf ERANGE gesetzt, um den Fehler anzuzeigen.

Siehe auch abs, ceil, floor, llrint, llround, lrint, lround, rint

scanf - Formatierte Eingabe von Standardeingabe

Definition #include <stdio.h>

int scanf(const char *format, argumentenliste);

`scanf` liest Daten (Eingabefelder) von der Standardeingabe `stdin`, wandelt diese gemäß den Angaben in der Formatzeichenkette *format* um und speichert die Ergebnisse in den Bereichen ab, die Sie mit den Ergebniszeigern in der Argumentenliste angeben. Jedes Argument muss ein Zeiger auf eine Variable sein, deren Datentyp mit einer Typangabe in der Formatzeichenkette *format* korrespondiert. Die Formatzeichenkette steuert, wie das Eingabefeld zu interpretieren und umzuwandeln ist.

Parameter const char *format

Die Formatzeichenkette kann drei Klassen von Zeichen bzw. Angaben enthalten:

1. Zwischenraumzeichen
2. Beliebige Zeichen, außer Zwischenraum- und dem Prozentzeichen (%)
3. Formatanweisungen, beginnend mit dem Prozentzeichen (%)

Zwischenraum (KR-Funktionalität, nur bei C/C++ Versionen kleiner V3.0)

␣	Leerzeichen
\n	Zeilenvorschub
\t	Tabulator

Zwischenraum (ANSI-Funktionalität)

␣	Leerzeichen
\n	Zeilenvorschub
\t	Tabulator
\f	Seitenwechsel
\v	vertikaler Tabulator
\r	Wagenrücklauf

Die Formatzeichenkette kann beliebig viele oder keine Zwischenraumzeichen enthalten. Diese Zeichen haben keine Steuerfunktion.

Irgendwelche Zwischenraumzeichen in der Eingabe werden als Trennzeichen zwischen Eingabefeldern behandelt und nicht mit umgewandelt (Ausnahme siehe %c und %[ ]).

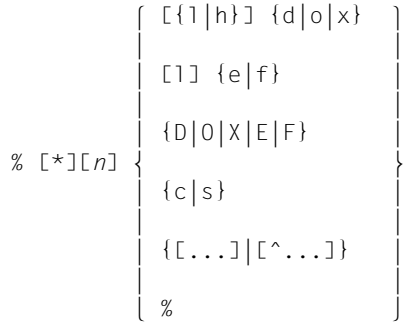
Beliebiges Zeichen außer % und Zwischenraum

Das Zeichen muss mit dem nächsten Zeichen aus der Eingabe übereinstimmen. `scanf` liest das Eingabezeichen, jedoch ohne es umzuwandeln und in einer Variablen abzuspeichern. Stimmt das Eingabezeichen nicht mit dem hier angegebenen Zeichen überein, wird die Eingabebearbeitung abgebrochen.

Im Folgenden wird die Formatanweisung getrennt nach KR-Funktionalität und ANSI-Funktionalität beschrieben.

Formatanweisung (KR-Funktionalität, nur bei C/C++ Versionen kleiner V3.0)

Formatanweisungen enthalten Angaben, wie die Eingabefelder zu interpretieren und umzuwandeln sind. Sie können folgendermaßen aufgebaut sein:



Zu einer Formatanweisung gehört ein Eingabefeld. Ein Eingabefeld ist eine Folge von Zeichen, die beendet wird

- durch das erste Zwischenraumzeichen,
- durch ein Zeichen, das nicht zur Formatanweisung (Typangabe) passt,
- wenn die explizit angegebene Feldlänge *n* erreicht ist.

Führende Zwischenraumzeichen werden bei der Eingabe ignoriert.

Jede Formatanweisung muss mit einem Prozentzeichen (%) beginnen. Die restlichen Zeichen werden wie folgt interpretiert:

* Überspringen einer Zuweisung.
Das nächste Eingabefeld wird zwar gelesen und umgewandelt, aber in keiner Variablen abgespeichert.

n Maximale Länge des umzuwandelnden Eingabefeldes.
Tritt vorher ein Zwischenraumzeichen oder ein Zeichen auf, das nicht zur Typangabe in der Formatanweisung passt, wird die Länge entsprechend gekürzt.

l l vor d, o, x:
Umwandlung eines Arguments vom Typ Zeiger auf `long int` (d) bzw. `unsigned long int` (o, x). Die Angabe ist identisch mit den Großbuchstaben D, O, X.

l vor e, f:
Umwandlung eines Arguments vom Typ Zeiger auf `double`.
Die Angabe ist identisch mit den Großbuchstaben E, F.

h h vor d, o, x:
Umwandlung eines Arguments vom Typ Zeiger auf `short int` (d) bzw. `unsigned short int` (o, x).

d Ein dezimaler Integerwert wird erwartet. Das entsprechende Argument muss ein Zeiger auf `int` sein.

o Ein oktaler Integerwert wird erwartet. Das entsprechende Argument kann ein Zeiger auf `unsigned int` oder `int` sein. Intern wird der Wert `unsigned` dargestellt.

x Ein sedezimaler Integerwert wird erwartet. Das entsprechende Argument kann ein Zeiger auf `unsigned int` oder `int` sein. Intern wird der Wert `unsigned` dargestellt.

e, f Eine Gleitkommazahl wird erwartet. Das entsprechende Argument muss ein Zeiger auf `float` sein.
Die Gleitkommazahl kann ein Vorzeichen enthalten sowie einen Exponenten (E bzw. e, gefolgt von einem ganzahligen Wert). Das Dezimalpunktzeichen wird durch die Lokalität (Kategorie LC_NUMERIC) beeinflusst. Voreingestellt ist der Punkt.

c Ein Zeichen wird erwartet. Das entsprechende Argument sollte ein Zeiger auf `character` sein.
`scanf` liest in diesem Fall auch Leerzeichen ein. Um das nächste Zeichen ungleich Leerzeichen einzulesen, ist "%1s" zu verwenden. c eignet sich dazu, Zeichenketten einzulesen, die auch Leerzeichen enthalten: Dazu ist als Argument ein Zeiger auf einen `char`-Vektor zu übergeben und eine Feldlänge n anzugeben (z.B. "%10c"). `scanf` schließt in diesem Fall die Zeichenkette nicht automatisch mit dem Nullbyte ab.

s Eine Zeichenkette wird erwartet. Das entsprechende Argument muss ein Zeiger auf einen `char`-Vektor sein und groß genug, um die Zeichenkette und ein abschließendes Nullbyte (\0) aufnehmen zu können. `scanf` schließt die Zeichenkette automatisch mit dem Nullbyte ab. Führende Zwischenraumzeichen in der Eingabe werden ignoriert, ein nachfolgendes Zwischenraumzeichen wird als Trennzeichen (Ende der Zeichenkette) interpretiert.

[ ] Eine Zeichenkette wird erwartet. Das entsprechende Argument muss ein Zeiger auf einen `char`-Vektor sein und groß genug, um die Zeichenkette inklusive des automatisch angefügten Nullbytes aufnehmen zu können. Im Unterschied zu %s fungieren bei dieser Angabe Leerzeichen nicht automatisch als Trennzeichen.

[...] Bei dieser Angabe werden solange Zeichen eingelesen, bis das erste Zeichen auftritt, das nicht in den eckigen Klammern angegeben ist. D.h., die Zeichenkette darf nur aus den Zeichen in [ ] bestehen, alle anderen Zeichen gelten als Trennzeichen.

[^...] Bei dieser Angabe werden solange Zeichen eingelesen, bis eines von den Zeichen auftritt, die in den eckigen Klammern nach ^ angegeben sind. Nur die in [ ] angegebenen Zeichen gelten als Trennzeichen.

% Eingabe des Zeichens %, keine Umwandlung.

Formatanweisung (ANSI-Funktionalität)

Formatanweisungen enthalten Angaben, wie die Eingabefelder zu interpretieren und umzuwandeln sind. Sie können folgendermaßen aufgebaut sein:

```
                ⎧ [{l|ll|h}] {d|i|o|u|x|X  }     ⎫
                ⎪                                 ⎪
                ⎪ [{l|L}] {e|E|f|g|G}             ⎪
                ⎪                                 ⎪
                ⎪ p                               ⎪
% [*][n] {      ⎪                                 ⎪ }
                ⎪ [] {[...]|[^...] |c|s }         ⎪
                ⎪                                 ⎪
                ⎪ [{l|ll|h}] n                    ⎪
                ⎪                                 ⎪
                ⎩ %                               ⎭
```

Zu einer Formatanweisung gehört ein Eingabefeld. Ein Eingabefeld ist eine Folge von Zeichen, die beendet wird

- durch das erste Zwischenraumzeichen,
- durch ein Zeichen, das nicht zur Formatanweisung (Typangabe) passt,
- wenn die explizit angegebene Feldlänge *n* erreicht ist.

Führende Zwischenraumzeichen werden bei der Eingabe ignoriert.

Jede Formatanweisung muss mit einem Prozentzeichen (%) beginnen. Die restlichen Zeichen werden wie folgt interpretiert:

* Überspringen einer Zuweisung.
Das nächste Eingabefeld wird zwar gelesen und umgewandelt, aber in keiner Variablen abgespeichert.

n Maximale Länge des umzuwandelnden Eingabefeldes.
Tritt vorher ein Zwischenraumzeichen oder ein Zeichen auf, das nicht zur Typangabe in der Formatanweisung passt, wird die Länge entsprechend gekürzt.

l l vor d, i, o, u, x, X:
Umwandlung eines Arguments vom Typ Zeiger auf `long int` (d, i) bzw. `unsigned long int` (o, u, x, X).

l vor e, E, f, g, G:
Umwandlung eines Arguments vom Typ Zeiger auf `double`.

l vor c, s, oder `[`:
Umwandlung eines Arguments vom Typ Zeiger auf `wchar_t`.

l vor n:
Das Argument ist vom Typ Zeiger auf `long int` (keine Umwandlung).

ll ll vor d, i, o, u, x, X:
Umwandlung eines Arguments vom Typ Zeiger auf `long long int` (d, i) bzw. `unsigned long long int` (o, u, x, X).

ll vor n:
Das Argument ist vom Typ Zeiger auf `long long int`.

h h vor d, i, o, u, x, X:
Umwandlung eines Arguments vom Typ Zeiger auf `short int` (d, i) bzw. `unsigned short int` (o, u, x, X).

h vor n:
Das Argument ist vom Typ Zeiger auf `short int` (keine Umwandlung).

L L vor e, E, f, g, G:
Umwandlung eines Arguments vom Typ Zeiger auf `long double`.

d Ein dezimaler Integerwert wird erwartet. Das entsprechende Argument muss ein Zeiger auf `int` sein.

i Ein Integerwert wird erwartet. Die Basis (sedezimal, oktal, dezimal) wird aus dem Aufbau des Eingabefeldes ermittelt.
Führendes 0x oder 0X: sedezimal
Führende 0: oktal
Sonst: dezimal
Das entsprechende Argument muss ein Zeiger auf `int` sein.

o Ein oktaler Integerwert wird erwartet. Das entsprechende Argument kann ein Zeiger auf `unsigned int` oder `int` sein. Intern wird der Wert `unsigned` dargestellt.

u Ein dezimaler Integerwert wird erwartet. Das entsprechende Argument muss ein Zeiger auf `unsigned int` sein.

x, X Ein sedezimaler Integerwert wird erwartet. Das entsprechende Argument kann ein Zeiger auf `unsigned int` oder `int` sein. Intern wird der Wert `unsigned` dargestellt.

e, E, f, g, G
Eine Gleitkommazahl wird erwartet. Das entsprechende Argument muss ein Zeiger auf `float` sein.
Die Gleitkommazahl kann ein Vorzeichen enthalten sowie einen Exponenten (E bzw. e, gefolgt von einem ganzahligen Wert).
Das Dezimalpunktzeichen wird durch die Lokalität (Kategorie LC_NUMERIC) beeinflusst. Voreingestellt ist der Punkt.

c Ohne Umwandlungszeichen `l`: Ein Zeichen wird erwartet. Das entsprechende Argument sollte ein Zeiger auf `character` sein. `scanf` liest in diesem Fall auch Leerzeichen ein. Um das nächste Zeichen ungleich Leerzeichen einzulesen, ist "%1s" zu verwenden. c eignet sich dazu, Zeichenketten einzulesen, die auch Leerzeichen enthalten: Dazu ist als Argument ein Zeiger auf einen `char`-Vektor zu übergeben und eine Feldlänge n anzugeben (z.B. "%10c"). `scanf` schließt in diesem Fall die Zeichenkette nicht automatisch mit dem Nullbyte ab.

Mit Umwandlungszeichen `l`: Es wird eine Multibyte-Zeichenkette erwartet, die im „initial shift“-Zustand beginnt. Die Multibyte-Zeichen werden in Langzeichen konvertiert (wie durch `mbrtowc`; das zugehörige `mbstate_t`-Objekt wird vor Konvertierung des ersten Multibyte-Zeichens mit Null initialisiert). Das entsprechende Argument muss ein Zeiger auf das erste Element eines Feldes vom Typ `wchar_t` sein, das groß genug ist, um die resultierende Folge aufzunehmen. Es wird kein abschließendes Null-Langzeichen angefügt.

p Ein 8stelliger Zeigerwert wird erwartet, analog dem Format %08.8x. Das entsprechende Argument muss vom Typ Zeiger auf einen Zeiger auf `void` sein.

s Ohne Umwandlungszeichen `l`: Eine Zeichenkette wird erwartet. Das entsprechende Argument muss ein Zeiger auf einen `char`-Vektor sein und groß genug, um die Zeichenkette und ein abschließendes Nullbyte (\0) aufnehmen zu können. `scanf` schließt die Zeichenkette automatisch mit dem Nullbyte ab. Führende Zwischenraumzeichen in der Eingabe werden ignoriert, ein nachfolgendes Zwischenraumzeichen wird als Trennzeichen (Ende der Zeichenkette) interpretiert.

Mit Umwandlungszeichen `l`: Es wird eine Multibyte-Zeichenkette erwartet, die im „initial shift“-Zustand beginnt. Die Multibyte-Zeichen werden in Langzeichen konvertiert (wie durch `mbrtowc`; das zugehörige `mbstate_t`-Objekt wird vor Konvertierung des ersten Multibyte-Zeichens mit Null initialisiert). Das entsprechende Argument muss ein Zeiger auf einen `wchar_t`-Vektor sein und groß genug, um die resultierende Langzeichenkette und ein abschließendes Null-Langzeichen aufnehmen zu können: `scanf` schließt die Langzeichenkette automatisch mit dem

Null-Langzeichen ab. Führende Zwischenraumzeichen in der Eingabe werden ignoriert, ein nachfolgendes Zwischenraumzeichen wird als Trennzeichen (Ende der Langzeichenkette) interpretiert.

[ ] Ohne Umwandlungszeichen `l`: Eine Zeichenkette wird erwartet. Das entsprechende Argument muss ein Zeiger auf einen `char`-Vektor sein und groß genug, um die Zeichenkette inklusive des automatisch angefügten Nullbytes aufnehmen zu können. Im Unterschied zu %s fungieren bei dieser Angabe Leerzeichen nicht automatisch als Trennzeichen.

[...] Bei dieser Angabe werden solange Zeichen eingelesen, bis das erste Zeichen auftritt, das nicht in den eckigen Klammern angegeben ist. D.h., die Zeichenkette darf nur aus den Zeichen in [ ] bestehen, alle nicht angegebenen Zeichen gelten als Trennzeichen.
Die schließende Klammer] kann in die Liste der einzulesenden Zeichen aufgenommen werden, wenn sie als erstes Zeichen unmittelbar nach der öffnenden Klammer angegeben wird: [ ]...].

[^...] Bei dieser Angabe werden solange Zeichen eingelesen, bis eines von den Zeichen auftritt, die in den eckigen Klammern nach ^ angegeben sind. Nur die in [ ] angegebenen Zeichen gelten als Trennzeichen.
Die schließende Klammer] kann in die Liste der Trennzeichen aufgenommen werden, wenn sie als erstes Zeichen unmittelbar nach dem Zeichen ^ angegeben wird: [^]...].

Mit Umwandlungszeichen `l`: Es wird eine Multibyte-Zeichenkette erwartet, die im „initial shift“-Zustand beginnt. Die Multibyte-Zeichen werden in Langzeichen konvertiert (wie durch `mbrtowc`; das zugehörige `mbstate_t`-Objekt wird vor Konvertierung des ersten Multibyte-Zeichens mit Null initialisiert). Das entsprechende Argument muss ein Zeiger auf einen `wchar_t`-Vektor sein und groß genug, um die resultierende Langzeichenkette und ein abschließendes Null-Langzeichen aufnehmen zu können: `scanf` schließt die Langzeichenkette automatisch mit dem Null-Langzeichen ab.

n Es werden keine Zeichen vom Eingabefeld gelesen. Das Argument ist vom Typ Zeiger auf `int`. Dieser ganzzahligen Variablen wird die Anzahl der Zeichen zugewiesen, die `scanf` bis zu diesem Zeitpunkt verarbeitet hat.

% Eingabe des Zeichens %, keine Umwandlung.

argumentenliste
: Zeiger auf Variablen, in die `scanf` die umgewandelten Ergebnisse abspeichern soll. Für %*-Anweisungen (Zuweisung überspringen) in *format* dürfen keine Zeigerargumente angegeben werden. Für alle anderen %-Anweisungen muss es jeweils ein Zeigerargument geben. Der Datentyp des Zeigerargumentes richtet sich nach der Typangabe in der zugehörigen Formatanweisung.

Returnwert
: Anzahl der eingelesenen und erfolgreich umgewandelten Eingabefelder.
Dazu zählen weder die Eingabefelder, für die %* (Zuweisung überspringen) angegeben wurde noch die mit %n (Anzahl verarbeitete Zeichen) erfolgten Zuweisungen.

EOF
: bei einem Fehler vor Beginn der Umwandlung.

Hinweise
: Bei der Umwandlung von Integerwerten in `unsigend int` (o, u, x, X) wird aus einem Wert mit negativem Vorzeichen das Zweierkomplement gebildet.
Z.B. liefert Format %u bei der Eingabe -1 X'FFFFFFFF'.

Das Ergebnis eines `scanf`-Aufrufes sollten Sie immer abfragen, um sicher zu sein, dass kein Fehler passiert ist!

Der nächste `scanf`-Aufruf startet mit dem Lesen unmittelbar nach dem Zeichen, das als letztes vom vorherigen Aufruf verarbeitet wurde.

Wenn ein Eingabezeichen nicht dem angegebenen Format entspricht, wird es in den Eingabepuffer zurückgeschrieben. Es muss dort mit `getc` abgeholt werden, sonst erhält der nächste `scanf`-Aufruf dasselbe Zeichen noch einmal.

Gibt es mehr Zeigerargumente als Formatanweisungen (exkl. den %*-Angaben), werden die überzähligen Argumente ignoriert. Gibt es weniger Argumente, führt dies zu undefinierten Ergebnissen.

Beispiel 1

```
int i
float x;
char name[20];
scanf("%2d %f %*d %6s", &i, &x, name);

Eingabedaten: 234567 678 Hubertxy

Inhalt der Variablen nach scanf:

i:     23
x:     4567.0
name:  Hubert\0
```

Die Zuweisung von 678 wird wegen der %*d Angabe nicht ausgeführt. Der Aufruf der nächsten Einleseoperation beginnt mit dem Zeichen 'x'.

Beispiel 2

```
int i;
float x;
char name[50];
scanf("%2d %f %*d %6s", &i, &x, name);

Eingabedaten: 25 54.32E-1 thomson

Inhalt der Variablen nach scanf:

i:     25
x:     5.432
name:  thomso\0
```

Beispiel 3

```
char string1[20];
char string2[20];
scanf("%[1234567890] %[^,!:]", string1, string2);

Eingabedaten: 234567ab  c,de

Inhalt der Variablen nach scanf:

string1: 234567
string2: ab  c
```

Siehe auch fscanf, sscanf

setbuf - Ein-/Ausgabe-Puffer einrichten

Definition #include <stdio.h>

void setbuf(FILE *dz, char *puffer);

`setbuf` legt einen Speicherbereich für die Datei mit Dateizeiger *dz* an. Dieser Speicherbereich wird dann an Stelle des vom System zugewiesenen Bereichs zur Pufferung der Ein-/Ausgabedaten verwendet.

Der Dateizeiger *dz* muss auf eine bereits geöffnete Datei zeigen, für die noch keine Lese- bzw. Schreibfunktion durchgeführt wurde.

Parameter FILE *dz
Dateizeiger der Datei, für die ein Ein-/Ausgabepuffer bereitgestellt werden soll.

char *puffer
Zeiger auf den Bereich, der als Puffer verwendet werden soll oder NULL.
Ist das Argument ein NULL-Zeiger, wird der vom System zugewiesene Puffer verwendet.

Hinweis Der Zeiger *puffer* muss für eine Datei mit Standardattributen auf einen Bereich der Größe BUFSIZ zeigen. BUFSIZ ist in <stdio.h> definiert.
Wird der Blockungsfaktor mit dem BUFFER-LENGTH-Parameter des ADD-FILE-LINK-Kommandos explizit vereinbart, muss die Größe des Bereichs dieser vereinbarten Blockungsgröße entsprechen.

Siehe auch setvbuf

setjmp - Marke für nicht lokale Sprünge setzen

Definition

```
#include <setjmp.h>

int setjmp(jmp_buf env);
```

`setjmp` ist nur zusammen mit der Funktion `longjmp` sinnvoll: Mit diesen Funktionen lassen sich nicht lokale Sprünge realisieren, d.h. Sprünge von einer beliebigen Funktion in eine andere, noch aktive Funktion.

`setjmp` speichert den aktuellen Programmzustand (Adresse im C-Laufzeitstack, Befehlszähler, Registerinhalte) in einem Feld vom Typ `jmp_buf`. Der Typ `jmp_buf` ist in <setjmp.h> definiert.
Ein späterer `longjmp`-Aufruf richtet den von `setjmp` gespeicherten Programmzustand wieder ein und setzt die Programmausführung von diesem Zustand aus fort.

Eine ausführliche Beschreibung und Hinweise zu `setjmp/longjmp` finden Sie bei der Funktion `longjmp`.

Returnwert

0	Normale Rückkehr, d.h. an die Stelle nach dem `setjmp`-Aufruf wurde nicht durch einen `longjmp`-Aufruf verzweigt.
≠ 0	Wenn an die Stelle nach dem `setjmp`-Aufruf mit `longjmp` verzweigt wurde, und zwar das Argument *wert* von `longjmp` (falls *wert* gleich 0 ist, liefert `setjmp` 1).

Beispiel siehe Beispiel bei `longjmp`

Siehe auch longjmp, signal

setlocale - Lokalität auswählen/abfragen

Definition #include <locale.h>

char *setlocale(int category, const char *locale);

Mit `setlocale` lässt sich die aktuelle Lokalität abfragen oder eine neue Lokalität auswählen. Die Lokalität kann sich dabei auf alle Lokalitätsvariablen des Programms beziehen oder auf Teile davon.
Die Lokalitätsvariablen sind in <locale.h> definiert.

Parameter int category

Kategorie der Lokalitätsvariablen, auf die sich die mit *locale* ausgewählte Lokalität beziehen soll. *category* kann folgende vordefinierte Werte enthalten:

LC_ALL	Lokalitätsvariablen aller Kategorien.
LC_COLLATE	Die Sortierreihenfolge beeinflusst das Verhalten der Funktionen `strcoll` und `strxfrm`.
LC_CTYPE	Die Zeichenart beeinflusst das Verhalten der Makros zur Zeichenbearbeitung is... (nicht `isdigit` und `isxdigit`), `tolower`, `toupper`, `strlower` und `strupper`.
LC_MONETARY	Die Konventionen zur Darstellung von monetären Werten beeinflussen die von `localeconv` gelieferten Werte.
LC_NUMERIC	Die Konventionen zur Darstellung von nicht-monetären numerischen Werten beeinflussen die Art des Dezimalpunktes bei formatierter Ein-/Ausgabe und bei der Umwandlung von Zeichenketten (`atof`, `strtod`), sowie die von `localeconv` gelieferten Werte.
LC_TIME	Die Konventionen zur Darstellung von Datum und Uhrzeit beeinflussen das Verhalten von `strftime`.

const char *locale

Zeichenkette, die die Lokalität auswählt. Folgende vordefinierte Lokalitäten stehen zur Verfügung (ausführliche Beschreibung siehe Seite 100ff):

"C"	Definiert die minimale Umgebung für die Übersetzung eines C-Programms und ist bei Programmstart voreingestellt (Ausnahme: siehe Lokalität "V1CTYPE").
""	Standard-Lokalität. Sie entspricht in dieser Version der Lokalität "C".
"V1CTYPE"	Kompatible Lokalität zum C-Laufzeitsystem V1.0. "V1CTYPE" wird bei Programmstart automatisch eingestellt, wenn die main-Routine ein C-V1.0-Objekt ist.
"V2CTYPE"	Kompatible Lokalität zum C-Laufzeitsystem V2.0 und V2.1.
"GERMANY"	Länderspezifische Lokalität, die den landesüblichen Konventionen entspricht.
"De.EDF04F"	Länderspezifische Lokalität, deren Konvertierungstabellen auf ASCII-Code ISO 8859-15 bzw. EBCDIC-Code EDF04F basieren und die in der Kategorie LC_MONETARY die Währung “DM“ unterstützt.
"De.EDF04F@euro"	Länderspezifische Lokalität, deren Konvertierungstabellen auf ASCII-Code ISO 8859-15 bzw. EBCDIC-Code EDF04F basieren und die in der Kategorie LC_MONETARY die Währung “Euro“ unterstützt.

Die Zeichenketten sind in der Include-Datei <locale.h> folgendermaßen vordefiniert:

LC_C_C	"C"
LC_C_DEFAULT	""
LC_C_V1CTYPE	"V1CTYPE"
LC_C_V2CTYPE	"V2CTYPE"
LC_C_GERMANY	"GERMANY"
LC_C_De.EDF04F	"De.EDF04F"
LC_C_De.EDF04F@euro	"De.EDF04F@euro"

Wird für *locale* ein NULL-Zeiger übergeben, wird die aktuelle Lokalität für die Kategorie *category* nicht verändert.

Returnwert Zeiger auf eine Zeichenkette, die die aktuelle Lokalität für die angegebene Kategorie *category* angibt.
Diese Zeichenkette kann bei `setlocale`-Aufrufen als Parameter *locale* verwendet werden.
Die Zeichenkette kann folgende Werte annehmen:
"C", "V1CTYPE", "V2CTYPE", "GERMANY", "De.EDF04F", "De.EDF04F@euro".
Bei Angabe der Kategorie LC_ALL enthält die Zeichenkette den Wert "C", sofern für alle Kategorien dieser Wert eingestellt ist.

Sobald für eine Kategorie eine Lokalität ungleich "C" eingestellt ist, enthält die Zeichenkette die Lokalitäten für alle Kategorien. Den Werten für die einzelnen Kategorien wird dann jeweils ein Schrägstrich (/) vorangestellt, der den Beginn eines neuen Wertes anzeigt. Die Reihenfolge der Lokalitäten entspricht der Reihenfolge der oben aufgeführten Kategorien (siehe Parameterbeschreibung *int category*).

Die letzte (sechste) Lokalität in der Zeichenkette bezieht sich auf die Kategorie LC_MESSAGES, die derzeit für die Lokalitäten “C“, “GERMANY“, “VC1TYPE“ und “VC2TYPE“ nicht unterstützt wird und für diese Lokalitäten immer auf "C" gesetzt ist.
Geben Sie hier die Lokalität "De.EDF04F" oder "De.EDF04F@euro" an, wird in der Kategorie LC_MESSAGES der entsprechende Wert eingetragen.

Wird eine Zeichenkette, die die Lokalitäten für alle Kategorien enthält, als Parameter *locale* bei einem `setlocale`-Aufruf verwendet und eine andere Kategorie als LC_ALL angegeben, dann wird dieser Zeichenkette nur die Lokalität für die angegebene Kategorie entnommen (ohne den führenden Schrägstrich).
Beispiel für den Returnwert bei LC_ALL:
"/V2CTYPE/C/GERMANY/C/GERMANY/C"
1 2 3 4 5 6

Position	Kategorie	Lokalität
1	LC_COLLATE	V2CTYPE
2	LC_CTYPE	C
3	LC_MONATARY	GERMANY
4	LC_NUMERIC	C
5	LC_TIME	GERMANY
6	LC_MESSAGES	C

NULL-Zeiger falls die ausgewählte Kategorie nicht hergestellt werden kann. Die bisher aktuelle Lokalität bleibt unverändert.

Hinweise

Die verfügbaren Lokalitäten sind im Kapitel „Lokalität“ auf Seite 99 ausführlich beschrieben.

Anwenderspezifische Lokalitäten:
Zusätzlich zu den o.g. vordefinierten Lokalitäten lassen sich auch eigene Lokalitäten implementieren und durch `setlocale` auswählen (siehe Abschnitt „Benutzerspezifische Lokalitäten“ auf Seite 116).

Die Zeichenkette, auf die der Returnwert von `setlocale` zeigt, darf nicht explizit durch das Programm verändert werden. Sie kann nur durch `setlocale`-Aufrufe überschrieben werden.

Soll die aktuelle Lokaltität lediglich abgefragt, aber nicht verändert werden, ist für *locale* ein NULL-Zeiger zu übergeben.

Siehe auch

localeconv

setvbuf - Ein-/Ausgabe-Puffer einrichten

Definition

```
#include <stdio.h>

int setvbuf(FILE *dz, char *puffer, int typ, size_t anz);
```

`setvbuf` legt einen Speicherbereich für die Datei mit Dateizeiger *dz* an. Dieser Speicherbereich wird dann an Stelle des vom System zugewiesenen Bereichs zur Pufferung der Ein-/Ausgabedaten verwendet.

Der Dateizeiger *dz* muss auf eine bereits geöffnete Datei zeigen, für die noch keine Lese- bzw. Schreibfunktion durchgeführt wurde.

Parameter

FILE *dz
: Dateizeiger der Datei, für die ein Ein-/Ausgabepuffer bereitgestellt werden soll.

char *puffer
: Zeiger auf den Bereich, der als Puffer verwendet werden soll oder NULL.

  Ist das Argument ein NULL-Zeiger, wird der vom System zugewiesene Puffer verwendet.

int typ
: Pufferungstyp der Datei. Dieser Parameter wird nur syntaktisch überprüft und ansonsten ignoriert. Er muss einen der folgenden vordefinierten Werte enthalten:

  _IOFBF (Vollpufferung)
  _IOLBF (zeilenweise Pufferung)
  _IONBF (keine Pufferung, nicht unterstützt)

  Der Pufferungstyp ist durch die Art der Datei festgelegt und kann nicht vom Benutzer verändert werden:
  Textdateien sind zeilengepuffert, d.h. die Daten werden bei Auftritt eines Neue-Zeile-Zeichens (\n) in die Datei geschrieben.
  Binärdateien sind vollgepuffert, d.h. die Daten werden in die Datei geschrieben, wenn der Puffer voll ist.
  Eine ungepufferte Ein-/Ausgabe wird nicht unterstützt.

size_t size
: Größe des bereitgestellten Puffers in Bytes.

Returnwert

0	wenn die Funktion `setvbuf` erfolgreich ausgeführt wurde.
≠ 0	wenn ein (syntaktisch) ungültiger Wert für *typ* übergeben wurde oder die Funktion nicht ausgeführt werden kann.

Hinweis Der Zeiger *puffer* muss für eine Datei mit Standardattributen auf einen Bereich der Größe BUFSIZ zeigen. BUFSIZ ist in <stdio.h> definiert.
Wird der Blockungsfaktor mit dem BUFFER-LENGTH-Parameter des ADD-FILE-LINK-Kommandos explizit vereinbart, muss die Größe des Bereichs dieser vereinbarten Blokkungsgröße entsprechen.

Siehe auch setbuf

signal - Signalbearbeitung steuern

Definition

```
#include <signal.h>

void (*signal(int sig, void (*fkt) (int))) (int);
```

Die Funktion `signal` steht zur Behandlung von Signalen zur Verfügung.

Man muss zwei Arten von Signalen unterscheiden, die ein Programm erhalten und behandeln kann. Sie unterscheiden sich in der Art der Auslösung. In der Folge davon wird ihre Behandlung intern unterschiedlich realisiert:

1. STXIT-Ereignisse

   STXIT-Ereignisse werden ausgelöst

   – durch Programmfehler, z.B. Adressfehler, Ausführung von ungültigen Instruktionen, Division durch Null etc.,

   – durch die Funktion `alarm`,

   – von außen, z.B. durch Drücken der K2-Taste, bestimmte Kommandos (ABEND, INFORM-PROGRAM etc.)

   Die Behandlung dieser Ereignisse wird intern über den BS2000-spezifischen STXIT-Contingency-Mechanismus realisiert. Dieser Mechanismus sowie die STXIT-Ereignisklassen sind ausführlich im Handbuch „Makroaufrufe an den Ablaufteil“ dargestellt.

2. `raise`-Signale

   Hierunter sind alle Ereignisse zu verstehen, die mit der Funktion `raise` ausgelöst werden können. Mit `raise` lassen sich STXIT-Ereignisse simulieren und von STXIT-Ereignissen unabhängige Signale (benutzereigene sowie vordefinierte) senden.

   Die Behandlung dieser Art Signale wird C-spezifisch, d.h. nicht über den o.g. Mechanismus realisiert.

Hat ein Programm keine Behandlung von Signalen vorgesehen, wird bei Eintritt eines Signals das Programm abgebrochen.

Das Programm kann ein Signal aber auch abfangen. Dazu muss man die Funktion `signal` aufrufen und als Argument eine Funktion *fkt* übergeben.
Es gibt dann folgende Möglichkeiten, auf ein Signal zu reagieren:

– *fkt* ist die voreingestellte Funktion SIG_DFL: Das Programm wird abgebrochen.

– *fkt* ist die vordefinierte Funktion SIG_IGN: Das Signal wird ignoriert.

– *fkt* ist eine selbst geschriebene Routine: Das Signal wird gemäß dieser Routine behandelt.

Im Folgenden werden diese drei Möglichkeiten der Signalbehandlung etwas ausführlicher erläutert, um dabei vor allem auf die unterschiedliche Behandlung von STXIT-Ereignissen und `raise`-Signalen einzugehen.

Programmabbruch

Programmabbruch erfolgt dann, wenn das Programm keine Signalbehandlung vorsieht oder wenn `signal` mit der Funktion SIG_DFL aufgerufen wird.

STXIT-Ereignis:

Es erfolgt die Standard-Abbruchreaktion durch das Betriebssystem. Das Programm wird abgebrochen, und es werden Informationen ausgegeben zur Abbruchadresse und zum Fehlergewicht sowie eine DUMP-Meldung:

```
... PROCESSING INTERRUPTED AT adresse ..., EC=gewicht
... DUMP DESIRED? REPLY(Y=YES,N=NO)?
```

`raise`-Signal:

Es erfolgt eine C-spezifische Programmbeendigung durch exit(-1) und es werden folgende Meldungen ausgegeben:

```
CCM0101 signal occured: signal
CCM0999 Exit -1
```

Signal ignorieren

Ein Signal wird ignoriert, wenn `signal` mit der vordefinierten Funktion SIG_IGN aufgerufen wird. Der Programmablauf wird fortgesetzt, so als wenn kein Signal eingetreten wäre. Hier gibt es keinen Unterschied zwischen der Behandlung von STXIT-Ereignissen und `raise`-Signalen.

Signal gemäß eigener Funktion fkt behandeln

Ein Signal wird gemäß einer selbst geschriebenen Funktion *fkt* behandelt, wenn `signal` mit dem Namen dieser Funktion aufgerufen wird. Bei Eintritt eines Signals wird das aufrufende Programm unterbrochen und die Funktion *fkt* ausgeführt. Nach Beendigung der Signalbehandlung wird das Programm an der Stelle fortgesetzt, an der es unterbrochen wurde (Ausnahme: in *fkt* wurden die Funktionen `exit` bzw. `longjmp` aufgerufen).

STXIT-Ereignis:

fkt wird intern als eigener STXIT-Contingency-Prozess realisiert, das übrige Programm als sog. „Basisprozess“. Die Steuerung wird durch das Betriebssystem vorgenommen.

raise-Signal:

fkt wird intern als „normale“ C-Funktion behandelt und nicht über den Contingency-Mechanismus realisiert. Die Steuerung obliegt dem C-Laufzeitsystem.

Weitere Einzelheiten bzgl. der unterschiedlichen Realisierung und der damit verbundenen unterschiedlichen Möglichkeiten von `signal`-Aufrufen entnehmen Sie bitte den „Hinweisen“.

Parameter int sig

Signal, das behandelt werden soll.

Für *sig* können symbolische Konstanten eingesetzt werden, die in der folgenden Tabelle unter „SIGNR“ aufgelistet sind. Diese Konstanten sind in der Include-Datei <signal.h> definiert.
Die Tabelle zeigt außerdem in der letzten Spalte die möglichen Arten der Signalauslösung (STXIT-Ereignis / raise / alarm). Bei STXIT-Ereignissen ist die jeweilige STXIT-Ereignisklasse angegeben.

SIGNR	Bedeutung	Signalauslösung durch
SIGHUP	Abbruch der Dialogstationsleitung	ABEND / raise
SIGINT	Unterbrechung von der Dialogstation (K2)	ESCPBRK / raise
SIGILL	Ausführung einer ungültigen Instruktion	PROCHK / raise
SIGABRT	raise-Signal für Programmbeendigung mit `_exit(-1); abort`	raise / abort
SIGFPE	fehlerhafte Gleitkommaoperation	PROCHK / raise
SIGKILL	raise-Signal für Programmbeendigung mit `exit(-1)`	raise
SIGSEGV	Speicherzugriff mit unerlaubtem Segmentzugriff	ERROR / raise
SIGALARM	Zeitintervall abgelaufen (Realzeit)	RTIMER / raise / alarm
SIGTERM	Programmbeendigung	TERM / raise
SIGUSR1	vom Benutzer definiert	raise
SIGUSR2	vom Benutzer definiert	raise

SIGNR	Bedeutung	Signalauslösung durch
SIGDVZ	Division durch 0	PROCHK / raise
SIGXCPU	CPU-Zeit aufgebraucht	RUNOUT / raise
SIGBPT	Haltepunkt (nicht unterstützt)	SVC
SIGTIM	Zeitintervall abgelaufen (CPU-Zeit, SETIC)	TIMER / raise
SIGINTR	SEND-MESSAGE-Kommando	INTR / raise
SIGSVC	SVC-Aufruf (nicht unterstützt)	SVC

Die symbolische Konstante für die Signalnummer kann durch einen weiteren symbolischen Namen ergänzt werden, z.B. `signal(SIGDVZ + SIG_PSK, fkt)`. Mit diesem Zusatz (im Beispiel "+ SIG_PSK") wird gesteuert, ob die Funktion *fkt* nur auf Grund ei-

nes STXIT-Ereignisses oder zusätzlich auf Grund eines `raise`-Signals aktiviert werden soll und ob *fkt* dem entsprechenden Signal temporär oder permanent zugeordnet werden soll. Die technischen Hintergründe hierfür entnehmen Sie bitte den „Hinweisen“. Die symbolischen Namen sind in <signal.h> definiert.

Ohne Zusatz ist SIG_TSK voreingestellt.

Symbolischer Name	Zuordnung	Aktivierung durch
SIG_TSK	temporär	STXIT / raise (Standard)
SIG_TS	temporär	STXIT
SIG_PSK	permanent	STXIT / raise
SIG_PS	permanent	STXIT

void (*fkt)(int)
Name der Funktion, die bei Eintritt eines Signals aufgerufen werden soll. Diese Funktion erhält als einziges Argument die Signalnummer vom Typ `int`.
Die Funktion muss **vor** dem entsprechenden `signal`-Aufruf definiert werden!

In <signal.h> gibt es zwei vordefinierte Funktionen:

SIG_DFL Diese Funktion ist voreingestellt und bewirkt einen Programmabbruch. Die Art des Abbruchs hängt davon ab, ob es sich um ein STXIT-Ereignis oder um ein `raise`-Signal handelt (siehe oben).

SIG_IGN Das Signal wird ignoriert.

Returnwert Vor dem `signal`-Aufruf gültige Funktion zur Signalbehandlung
bei Erfolg. `signal` liefert die letzte Einstellung der Signalbehandlung, und zwar SIG_DFL, SIG_IGN oder eine vom Benutzer geschriebene Funktion *fkt*.

SIG_ERR (= 1)bei Fehler, z.B. wenn *sig* keine gültige Signalnummer ist oder
fkt auf eine unzulässige Adresse zeigt.
Zusätzlich wird `errno` auf den entsprechenden Fehlercode gesetzt:
EINVAL (unzulässiges Argument)
EFAULT (unzulässige Adresse).

Hinweise Das Signal SIGKILL kann nicht abgefangen werden, d.h. ihm kann weder eine selbst geschriebene Funktion noch SIG_IGN zugeordnet werden.

Wenn für ein Signal, dem bereits eine Signalbehandlung zugeordnet ist, eine zweite Funktion zur Signalbehandlung angemeldet wird, wird zunächst die erste Funktion abgemeldet, bevor die neue Funktion angemeldet wird. Aus diesem Grunde ist für eine kurze Zeitspanne *keine* Signalbehandlung für das betroffene Signal angemeldet.

Aus einer Funktion, die dem Signal SIGTERM zugeordnet ist, kann nicht mit einem `longjmp`-Aufruf zurückgekehrt werden. Zum Zeitpunkt der Signalauslösung sind die Einträge im C-Laufzeitstack für alle Funktionen einschließlich der main-Funktion bereits abgebaut.

Bei Programmstart ist für alle Signale SIG_DFL voreingestellt (vgl. nächsten Absatz).

Temporäre/permanente Zuordnung:
In vielen Implementierungen (z.B. UNIX) und auch im ANSI-Standard ist die temporäre Zuordnung eines Signals zu einer Funktion vorgesehen. Das bedeutet: Die vom Anwender vorgenommene Zuordnung einer Funktion zu einer Signalnummer gilt nur temporär für ein einziges Auftreten dieses Signals. Die Zuordnung wird nach Auftritt des Signals aufgehoben und auf SIG_DFL (Programmabbruch) zurückgesetzt.
Lediglich die Zuordnung SIG_IGN (Signal ignorieren) gilt permanent für mehrmaliges Auftreten des entsprechenden Signals.

– Im BS2000 ist die Signalbehandlung für Signale vom Typ „STXIT-Ereignisse“ über den STXIT-Contingency-Mechanismus realisiert. Dieser Mechanismus beruht auf einer permanenten Zuordnung eines STXIT-Ereignisses zu einer STXIT-Contingency-Routine, d.h. eine temporäre Zuordnung kann nur durch explizites Abmelden erreicht werden.

– Um einerseits die temporäre Zuordnung vieler Implementierungen zu erfüllen, andererseits den BS2000-üblichen permanenten Charakter performant zu unterstützen, werden beide Varianten angeboten. D.h., man kann auswählen, ob eine Signalroutine temporär oder permanent zugeordnet wird.

– Zusätzlich wird aus Performancegründen die Möglichkeit angeboten, zu entscheiden, ob eine Signalroutine nur durch STXIT-Ereignisse ausgelöst werden kann (performanter) oder zusätzlich auch durch `raise`-Signale.

– Die o.g. Auswahlmöglichkeiten sind realisiert durch symbolische Ergänzungen der eigentlichen Signalnummer: SIG_TSK, SIG_TS, SIG_PSK, SIG_PS (siehe Parameterbeschreibung *sig*).

– Wollen Sie ein Signal jedes Mal mit *fkt* abfangen, sind z.B. folgende `signal`-Aufrufe möglich:

```
signal(SIGDVZ + SIG_PSK, fkt);   /* fkt wird durch STXIT-Ereignis und */
                                 /* raise-Signal SIGDVZ aktiviert.  */

signal(SIGDVZ + SIG_PS, fkt);    /* fkt wird nur durch STXIT-Ereignis */
                                      /* SIGDVZ aktiviert. */
```

Folgende Aufrufe sind äquivalent, d.h. beide sehen eine temporäre Zuordnung vor, und die Signalroutine wird durch ein STXIT-Ereignis und ein `raise`-Signal aktiviert:

```
signal(SIGDVZ, fkt);
signal(SIGDVZ + SIG_TSK, fkt);
```

Probleme können sich bei den drei verschiedenen Signalnummern ergeben, die durch dieselbe STXIT-Ereignisklasse (PROCHK) abgebildet werden. Folgende `signal`-Aufrufe werden unterschiedlich behandelt, je nach Art der Auslösung des Signals:

```
signal(SIGILL, fkt1);
signal(SIGFPE, fkt2);
signal(SIGDVZ, fkt3);
```

STXIT-Ereignis:

Es wird in jedem Fall *fkt3* aufgerufen, wenn die Signale SIGILL und SIGFPE über den STXIT-Contingency-Mechanismus abgefangen werden. Aber selbst, wenn nur für ein Signal ein `signal`-Aufruf vorgesehen ist, wird die zugeordnete Routine bei Eintreten aller drei Signale aktiviert.

`raise`-Signal:

Werden die Signale jedoch mit der `raise`-Funktion ausgelöst, wird die jeweils zugeordnete Funktion aktiviert. Signale, für die kein `signal`-Aufruf vorgesehen ist, werden standardmäßig behandelt (SIG_DFL, Programmabbruch).

Eine Funktion, die einem Signal zugeordnet ist, benötigt für ihren Ablauf eine intakte C-Umgebung. Daher werden bei regulärer Programmbeendigung unmittelbar vor dem Abbau der C-Umgebung alle Signalroutinen abgemeldet. Danach eintretende Ereignisse werden nicht mehr abgefangen, auch nicht SIGTERM.

Beispiel Folgendes Programm fängt mit der Funktion *fkt* die STXIT-Ereignisse SIGDVZ (Division durch 0) und SIGINT (Unterbrechung mit der K2-Taste) ab und gibt eine entsprechende Fehlermeldung aus. Mittels der Funktionen `setjmp` und `longjmp` fährt das Programm nach Behandlung beider Unterbrechungsereignisse (sie finden an verschiedenen Stellen des Programmes statt) mit derselben Programmstelle fort (neue Eingabeaufforderung).

```
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>

jmp_buf env;

void fkt(int sig)
{
  if(sig == SIGDVZ + SIG_PS)
    printf("Divisionsfehler, Eingabe wiederholen\n");
  if(sig == SIGINT + SIG_PS)
    printf("K2-Taste gedrueckt, Eingabe wiederholen\n");
  longjmp(env, 1);
}

int main(void)
{
  float a;
  float b;
  double z;
  signal(SIGDVZ + SIG_PS, fkt);
  signal(SIGINT + SIG_PS, fkt);
  setjmp(env);
  printf("Bitte a und b eingeben\n");  /* Unterbrechung mit K2 möglich */
  scanf("%f %f", &a, &b);
  z = a / b;                           /* Division durch 0 möglich, */
                                       /* falls b = 0 */
  printf("z = %f\n", z);
  printf ("Programmende\n");
  return 0;
}
```

Siehe auch alarm, longjmp, raise, setjmp

sin - Sinus

Definition #include <math.h>

double sin(double x);

`sin` berechnet für eine Gleitkommazahl x die trigonometrische Funktion Sinus. x gibt den Winkel im Bogenmaß an.

Returnwert `sin(x)` eine Gleitkommazahl im Intervall [-1.0, +1.0].

Beispiel Folgendes Programm gibt Sinuswerte aus dem Bereich -pi bis +pi aus.

```
#include <math.h>
#include <stdio.h>

#define pi 3.14159265358979

int main(void)
{
   double x;
   for (x = -pi; x <= pi; x = x + pi/4.)
       printf(" sin(%1.2f) = %.4f \n ", x, sin(x));
   return 0;
}
```

Siehe auch cos, asin, sinh

sinh - Sinus hyperbolicus

Definition #include <math.h>

double sinh(double x);

`sinh` berechnet den Sinus hyperbolicus für die Gleitkommazahl x.

Returnwert `sinh(x)` für eine Gleitkommazahl x.

+/-HUGE_VAL bei Überlauf (Vorzeichen von x). Zusätzlich wird `errno` auf ERANGE gesetzt (Resultat zu groß).

Beispiel Folgendes Programm gibt die Sinushyperbolicuswerte aus dem Bereich von -1 bis 1 aus (Schrittweite 0.1).

```
#include <math.h>
#include <stdio.h>

int main(void)
{
   double x;
   for (x = -1.0; x < 1.0; x = x + 0.1)
       printf(" sinh(%.2f) = %.4f \n ", x, sinh(x));
   return 0;
}
```

Siehe auch cosh, asin, sin

sleep - Programm für festgesetzte Zeitspanne anhalten

Definition #include <signal.h>

int sleep(unsigned int sek);

`sleep` hält ein Programm für *sek* Sekunden an.

Returnwert Angeforderte Zeit minus tatsächliche Zeit.
Falls `sleep` früher beendet wurde als in *sek* angegeben, wird die noch übrige Zeit angezeigt (siehe auch Hinweis).

Hinweise `sleep` legt das Programm für *sek* Sekunden auf Eis, indem intern der VPASS-Makro mit dem Wert von einer Sekunde in einer Schleife aufgerufen wird.
Obwohl das Programm mit `sleep` für Sekunden angehalten wird, läuft die Zeit für eine zuvor gestellte Alarmuhr (mit `alarm`) weiter. Dies hat folgende Auswirkungen:

1. Die vorher eingestellte Alarmzeit sei kleiner als die `sleep`-Zeit, etwa
   ```
   alarm(2);
   sleep(30);
   ```
   Nach Ablauf von zwei „Schlaf“-Sekunden wird der Alarm ausgelöst und der `sleep`-Aufruf beendet.
2. Die vorher eingestellte Alarmzeit sei größer als die `sleep`-Zeit, etwa
   ```
   alarm(30);
   sleep(5);
   ```
   Die Zeit der Alarmuhr läuft um 5 „schlafende“ Sekunden weiter. Die Alarmuhr steht nach dem `sleep`-Aufruf auf 25.

Die Zeit, die das Programm tatsächlich angehalten wird, kann auch noch aus folgenden Gründen von *sek* abweichen:

– sie kann bis zu einer Sekunde kürzer sein, weil das „Aufwecken“ in festen 1-Sekunden-Intervallen stattfindet,

– sie kann aus Prioritätsgründen beliebig länger sein, weil das System Wichtigeres zu tun hat.

Siehe auch alarm, signal

snprintf - Formatierte Ausgabe in eine Zeichenkette

Definition

```
#include <stdio.h>

int snprintf(char *s, size_t n, const char *format, argumentenliste);
```

`snprintf` bereitet Daten (Zeichen, Zeichenketten, numerische Werte) gemäß den Angaben in der Zeichenkette *format* auf und schreibt sie in den Bereich, auf den *s* zeigt.

`snprintf` bricht die Ausgabe beim Erreichen der mit dem Parameter *n* spezifizierten Länge ab, wodurch ein Pufferüberlauf verhindert werden kann. Ansonsten ist die Funktionalität von `snprintf` identisch zu der von `sprintf`.

Parameter

char *s
: Zeiger auf die Ergebniszeichenkette. `snprintf` schließt die Zeichenkette mit dem Nullbyte (\0) ab. Die maximale Länge der Ausgabe beträgt daher *n*-1.

size_t n
: Länge des für die Ergebniszeichenkette reservierten Bereichs. *n* darf nicht größer sein als `INT_MAX`. Bei *n* = 0 erfolgt keine Ausgabe.

const char *format
: Formatzeichenkette wie bei `printf` mit KR- oder ANSI-Funktionalität (Beschreibung siehe dort).

  Es gibt nur bzgl. der Steuerzeichen für Zwischenraum (\n, \t, etc.) folgenden Unterschied: `snprintf` trägt in die Ergebniszeichenkette den EBCDIC-Wert des Steuerzeichens ein. Erst bei der Ausgabe in Textdateien werden die Steuerzeichen je nach Art der Textdatei in ihre entsprechende Wirkung umgesetzt (siehe Abschnitt „Zwischenraum“ auf Seite 68).

argumentenliste
: Variablen oder Konstanten, deren Werte gemäß den Angaben in den Formatanweisungen für die Ausgabe umgewandelt und formatiert werden sollen.
  Wenn die Anzahl der Formatanweisungen nicht mit der Anzahl der Argumente übereinstimmt, gilt Folgendes:
  Gibt es mehr Argumente, werden die überzähligen ignoriert.
  Gibt es weniger Argumente, führt dies zu undefinierten Ergebnissen.

Returnwert

< 0	*n* > `INT_MAX` oder Ausgabefehler.
= 0 .. n-1	Die Ausgabe konnte vollständig aufbereitet werden. Der Returnwert gibt die Länge der Ausgabe ohne das abschließende `NULL`-Zeichen an.
> n	Die Ausgabe konnte nicht vollständig aufbereitet werden. Der Returnwert gibt die Länge ohne das abschließende `NULL`-Zeichen an, die eine vollständige Ausgabe benötigen würde.

Hinweise

Sie müssen dafür sorgen, dass *n* nicht größer ist, als die Länge des Bereichs auf den *s* zeigt!

Bei der Umwandlung von Gleitkommazahlen rundet `snprintf` auf die angegebene Genauigkeit.

`snprintf` nimmt keine Konvertierung von einem Datentyp in einen anderen vor. Soll ein Wert nicht entsprechend seinem Typ ausgegeben werden, muss er explizit konvertiert werden (z.B. mit dem `cast`-Operator).

Maximale Anzahl der auszugebenden Zeichen
Bei KR-Funktionalität (nur bei C/C++ Versionen kleiner V3.0 vorhanden) können pro `snprintf`-Aufruf maximal 1400 Zeichen ausgegeben werden,
bei ANSI-Funktionalität maximal 1400 Zeichen pro Konversionselement (z.B. %s).

Bei sich überlappenden Speicherbereichen ist das Verhalten undefiniert.

Versuche, nicht initialisierte Variablen oder Variablen nicht entsprechend ihrem Datentyp auszugeben, können zu undefinierten Ergebnissen führen.

Das Verhalten ist undefiniert, wenn in einer Formatanweisung dem Prozentzeichen (%) ein nicht definiertes Formatierungs- bzw. Umwandlungszeichen folgt.

Siehe auch

printf, fprintf, sprintf, putc, putchar, puts, sscanf

sprintf - Formatierte Ausgabe in eine Zeichenkette

Definition

```
#include <stdio.h>

int sprintf(char *s, const char *format, argumentenliste);
```

`sprintf` bereitet Daten (Zeichen, Zeichenketten, numerische Werte) gemäß den Angaben in der Zeichenkette *format* auf und schreibt sie in den Bereich, auf den *s* zeigt.

`sprintf` arbeitet wie `printf`, außer dass die aufbereiteten Daten in eine Zeichenkette und nicht auf die Standardausgabe geschrieben werden.

Parameter

char *s
: Zeiger auf die Ergebniszeichenkette. `sprintf` schließt die Zeichenkette mit dem Nullbyte (\0) ab.

const char *format
: Formatzeichenkette wie bei `printf` mit KR- oder ANSI-Funktionalität (Beschreibung siehe dort).

  Es gibt nur bzgl. der Steuerzeichen für Zwischenraum (\n, \t, etc.) folgenden Unterschied: `sprintf` trägt in die Ergebniszeichenkette den EBCDIC-Wert des Steuerzeichens ein. Erst bei der Ausgabe in Textdateien werden die Steuerzeichen je nach Art der Textdatei in ihre entsprechende Wirkung umgesetzt (siehe Abschnitt „Zwischenraum“ auf Seite 68).

argumentenliste
: Variablen oder Konstanten, deren Werte gemäß den Angaben in den Formatanweisungen für die Ausgabe umgewandelt und formatiert werden sollen.
  Wenn die Anzahl der Formatanweisungen nicht mit der Anzahl der Argumente übereinstimmt, gilt Folgendes:
  Gibt es mehr Argumente, werden die überzähligen ignoriert.
  Gibt es weniger Argumente, führt dies zu undefinierten Ergebnissen.

Returnwert

Anzahl der in *s* gespeicherten Zeichen.
: Das durch `sprintf` generierte abschließende Nullbyte (\0) wird dabei nicht mitgezählt.

Hinweise

Sie müssen dafür sorgen, dass der Bereich, auf den *s* zeigt, groß genug für das Ergebnis ist!

Bei der Umwandlung von Gleitkommazahlen rundet `sprintf` auf die angegebene Genauigkeit.

`sprintf` nimmt keine Konvertierung von einem Datentyp in einen anderen vor. Soll ein Wert nicht entsprechend seinem Typ ausgegeben werden, muss er explizit konvertiert werden (z.B. mit dem `cast`-Operator).

Maximale Anzahl der auszugebenden Zeichen
Bei KR-Funktionalität (nur bei C/C++ Versionen kleiner V3.0 vorhanden) können pro `sprintf`-Aufruf maximal 1400 Zeichen ausgegeben werden,
bei ANSI-Funktionalität maximal 1400 Zeichen pro Konversionselement (z.B. %s).

Bei sich überlappenden Speicherbereichen ist das Verhalten undefiniert.

Versuche, nicht initialisierte Variablen oder Variablen nicht entsprechend ihrem Datentyp auszugeben, können zu undefinierten Ergebnissen führen.

Das Verhalten ist undefiniert, wenn in einer Formatanweisung dem Prozentzeichen (%) ein nicht definiertes Formatierungs- bzw. Umwandlungszeichen folgt.

Beispiel Sie können `sprintf` z.B. dazu benutzen, eine Zeichenkette zu kopieren. Damit kann man die Funktion `strncpy` implementieren. Das Beispiel bei `strncpy` würde dann wie folgt aussehen:

```
#include <stdio.h>

int main(void)
{
   int n;
   char *s2 = "Peter geht schwimmen !";
   char s1[BUFSIZ];
   printf("Der Satz lautet : %s \nWieviel Zeichen kopieren ?\n", s2);
   scanf("%d",&n);

               /* Alternativ könnte an dieser Stelle
                  folgender Aufruf stehen:
                  strncpy(s1,s2,n);        */

  sprintf(s1,"%.*s",n,s2);
  printf("%s \n",s1);
  return 0;
}
```

Siehe auch printf, fprintf, snprintf, putc, putchar, puts, sscanf

sqrt - Quadratwurzel

Definition #include <math.h>

double sqrt(double x);

`sqrt` berechnet die Quadratwurzel zu einer nicht-negativen Gleitkommazahl x.

Returnwert

`sqrt(x)`	falls x >= 0 ist.
0	falls x negativ ist. Zusätzlich wird `errno` auf EDOM gesetzt (domain error, d.h. Argument unzulässig).

Beispiel Folgendes Programm berechnet die Quadratwurzel für einen eingelesenen Wert x.

```
#include <math.h>
#include <stdio.h>

int main(void)
{
   double x;
   scanf("%lf", &x);
   printf("Wurzel aus %g : %g\n", x, sqrt(x));
   printf("%d\n", errno);
   return 0;
}
```

Siehe auch exp, pow, log, log10, hypot, sinh

srand - Initialisieren des Zufallsgenerators

Definition #include <stdlib.h>

void srand(unsigned int i);

Mit `srand` können Sie den Zufallsgenerator initialisieren, der von `rand` aufgerufen wird. Mit `i = 1` setzen Sie den Zufallsgenerator auf seine voreingestellte Startzahl.

Beispiel siehe Beispiel bei `rand`

Siehe auch rand

sscanf - Formatierte Eingabe aus einer Zeichenkette

Definition

```
#include <stdio.h>

int sscanf(const char *s, const char *format, argumentenliste);
```

`sscanf` liest Daten (Eingabefelder) aus einer Zeichenkette *s*, wandelt diese gemäß den Angaben in der Formatzeichenkette *format* um und speichert die Ergebnisse in den Bereichen ab, die Sie mit den Ergebniszeigern in der Argumentenliste angeben.

`sscanf` arbeitet wie `scanf`, nur dass die Eingabefelder nicht von der Standardeingabe (`stdin`), sondern aus einer Zeichenkette gelesen werden.

Parameter

const char *s
: Zeichenkette, in der die Eingabedaten stehen. Sie sollte mit dem Nullbyte (\0) abgeschlossen sein.

const char *format
: Formatzeichenkette wie bei `scanf` mit KR- oder ANSI-Funktionalität (Beschreibung siehe dort).

argumentenliste
: Zeiger auf Variablen, in die `sscanf` die umgewandelten Ergebnisse abspeichern soll. Für %*-Anweisungen (Zuweisung überspringen) in *format* dürfen keine Zeigerargumente angegeben werden. Für alle anderen %-Anweisungen muss es jeweils ein Zeigerargument geben. Der Datentyp des Zeigerargumentes richtet sich nach der Typangabe in der zugehörigen Formatanweisung.

Returnwert

Anzahl der eingelesenen und erfolgreich umgewandelten Eingabefelder.
: Dazu zählen nicht die Eingabefelder, für die %* (Zuweisung überspringen) angegeben wurde.

EOF
: bei einem Fehler vor Beginn der Umwandlung.

Hinweise

Bei sich überlappenden Speicherbereichen ist das Ergebnis undefiniert.

Die ausführliche Beschreibung, Hinweise und Beispiele zur formatierten Eingabe finden Sie bei `scanf`.

Siehe auch

scanf, fscanf

_ _STDC_ _ - ANSI-Sprachstandard?

Definition _ _STDC_ _

Dieses Makro generiert den Wert 1 bei einer Übersetzung mit SOURCE-PROPERTIES=PARAMETERS(LANGUAGE-STANDARD=ANSI). In allen anderen Sprachmodi des Compilers ist der Wert dieses Makros undefiniert.

Hinweis Das Makro muss in keiner Include-Datei definiert werden. Sein Name wird vom Compiler erkannt und ersetzt.

_ _STDC_ VERSION_ _ - Amendment 1 konform?

Definition _ _STDC _VERSION_ _

Dieses Makro wird zu der Dezimalkonstanten 199409L expandiert und zeigt damit an, dass die Implementierung Amendment 1-konform ist.

Hinweis Das Makro muss in keiner Include-Datei definiert werden. Sein Name wird vom Compiler erkannt und ersetzt.

strcat - Verketten von Zeichenketten

Definition

```
#include <string.h>

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

`strcat` hängt eine Kopie der Zeichenkette *s2* ans Ende der Zeichenkette *s1* und liefert einen Zeiger auf *s1* zurück.

Das Nullbyte (\0) am Ende der Zeichenkette *s1* wird vom ersten Zeichen der Zeichenkette *s2* überschrieben.
`strcat` schließt die Zeichenkette mit dem Nullbyte (\0) ab.

Returnwert Zeiger auf die Ergebniszeichenkette.

Hinweise Als Argumente werden Zeichenketten erwartet, die mit dem Nullbyte (\0) abgeschlossen sind.

`strcat` überprüft nicht, ob der Speicherbereich von *s1* groß genug für das Ergebnis ist!

Bei sich überlappenden Speicherbereichen ist das Verhalten undefiniert.

Beispiel

```
#include <string.h>
#include <stdio.h>
int main(void)
{
  char text1[BUFSIZ];
  char text2[BUFSIZ];
  printf("Beispiel strcat - bitte 2 Textzeilen eingeben!\n");
  if(scanf("%s %s", text1, text2) == 2)
  printf("%s\n", strcat(text1, text2));
  return 0;
}
```

Siehe auch strncat

strchr - Erstes Vorkommen eines Zeichens in einer Zeichenkette

Definition #include <string.h>

char *strchr(const char *s, int c);

`strchr` sucht das erste Vorkommen des Zeichens *c* in der Zeichenkette *s* und liefert bei Erfolg einen Zeiger auf die gesuchte Position in *s*.

Das abschließende Nullbyte (\0) wird als Zeichen mitberücksichtigt.

Returnwert Zeiger auf die Position von *c* in der Zeichenkette *s*
bei Erfolg.

NULL-Zeiger wenn *c* in der Zeichenkette *s* nicht enthalten ist.

Hinweise Die Funktionen `strchr` und `index` sind äquivalent.

Für C++ gelten die beiden folgenden Prototypen für die Funktion `strchr`:
const char *strchr(const char *s, int c);
char *strchr(char *s, int c);

Beispiel Finde das erste ’k’:

```
#include <string.h>
#include <stdio.h>

int main(void)
{
   char *s = "was fuer ein Spass im kkuehlen Nass!";
   printf("%s\n", s);
   printf("Wo steckt der Fehler? %s\n", strchr(s, 'k'));
   return 0;
}
```

Siehe auch index, rindex, strrchr

strcmp - Vergleich von zwei Zeichenketten

Definition #include <string.h>

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

`strcmp` vergleicht zwei Zeichenketten *s1* und *s2* lexikalisch, z.B.:

"Zirkel" ist lexikalisch kleiner als "Zirkus",
"Busse" ist lexikalisch größer als "Bus".

Returnwert

< 0	*s1* ist lexikalisch kleiner als *s2*
= 0	*s1* und *s2* sind lexikalisch gleich groß
> 0	*s1* ist lexikalisch größer als *s2*

Hinweis Als Argumente werden Zeichenketten erwartet, die mit dem Nullbyte (\0) abgeschlossen sind.

Beispiel Folgendes Programm sucht einen eingelesenen Namen in der Namensliste *list*.

```
#include <stdio.h>
#include <string.h>

char *list[] = {"anne", "peter", "walter", "hans" };

int main(int argc, char *argv[])
{
  int j, i = 0;
  while((i <= 3) && (j = strcmp(argv[1], list[i++])));
  if (j == 0)
     printf("Der Kandidat ist schon bekannt!\n");
  else
     printf("Das ist ein neuer Kandidat!\n");
     return 0;
}
```

Siehe auch strncmp

strcoll - Vergleich von zwei Zeichenketten

Definition

```
#include <string.h>

int strcoll(const char *s1, const char *s2);
```

`strcoll` vergleicht zwei Zeichenketten *s1* und *s2* lexikalisch. Die lexikalische Reihenfolge der einzelnen Zeichen wird entsprechend der LC_COLLATE-Kategorie der aktuellen Lokalität interpretiert.

Returnwert

< 0	*s1* ist lexikalisch kleiner als *s2*
= 0	*s1* und *s2* sind lexikalisch gleich groß
> 0	*s1* ist lexikalisch größer als *s2*

Hinweise

Als Argumente werden Zeichenketten erwartet, die mit dem Nullbyte (\0) abgeschlossen sind.

Das Lokalitätskonzept ist ausführlich im Kapitel „Lokalität“ auf Seite 99 beschrieben.

Beispiel

siehe bei `strxfrm`.

Siehe auch

setlocale, strxfrm

strcpy - Zeichenkette kopieren

Definition #include <string.h>

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

`strcpy` kopiert die Zeichenkette *s2* einschließlich des Nullbytes (\0) in die Zeichenkette *s1*. *s1* muss groß genug sein, um die Zeichenkette *s2* einschließlich des Nullbytes (\0) aufnehmen zu können.

Returnwert Zeiger auf die Ergebniszeichenkette *s1*.

Hinweise Als Argumente werden Zeichenketten erwartet, die mit dem Nullbyte (\0) abgeschlossen sind.

`strcpy` überprüft nicht, ob *s1* groß genug für das Ergebnis ist. Ist *s1* kleiner als *s2* (einschließlich des Nullbytes), so ist das Ergebnis eine Zeichenkette, die nicht mit dem Nullbyte abgeschlossen ist!

Bei sich überlappenden Speicherbereichen ist das Verhalten undefiniert.

Beispiel Folgendes Programm gibt die Inhalte von *s1* und *s2* aus, ruft dann `strcpy` auf und gibt nochmal beide Inhalte aus.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    char s1[] = "Anne hat es gut !";
    char s2[] = "Roland besser !";
    printf("Inhalt s1: %s\nInhalt s2: %s\n", s1, s2);

    strcpy(s1, s2);   /* s2 nach s1 kopieren */
    printf("Nach strcpy:\nInhalt s1: %s\nInhalt s2: %s\n", s1, s2);
    return 0;
}
```

Siehe auch strncpy

strcspn - Zeichenketten vergleichen und Segmentlänge berechnen

Definition #include <string.h>

size_t strcspn(const char *s1, const char *s2);

`strcspn` berechnet ab Beginn der Zeichenkette *s1* die Länge des Segmentes, das kein einziges Zeichen aus der Zeichenkette *s2* enthält. Das abschließende Nullbyte (\0) gilt nicht als Teil der Zeichenkette *s2*.

Sobald ein Zeichen in *s1* mit einem Zeichen in *s2* übereinstimmt, wird die Funktion beendet und die Segmentlänge zurückgeliefert.

Stimmt bereits das erste Zeichen in *s1* mit einem Zeichen in *s2* überein, ist die Segmentlänge gleich 0.

Returnwert Ganzzahliger Wert, der die Segmentlänge (Anzahl ungleicher Zeichen) ab Beginn der Zeichenkette *s1* angibt.

Hinweis Als Argumente werden Zeichenketten erwartet, die mit dem Nullbyte (\0) abgeschlossen sind.

Beispiel

```
#include <stdio.h>
#include <string.h>
int main(void)
{
 char text1[40];
 static char text2[] = "/*#$&";
 size_t n;
 printf("Beispiel fuer strcspn. Bitte eine Textzeile eingeben:\n");
 scanf("%s",text1);
 n = strcspn(text1, text2);
 printf("Laenge des Anfangssegments ohne /, *, #, $, &: %d\n", n);
 return 0;
}
```

Siehe auch strspn

strerror - Fehlermeldungstext ermitteln

Definition

```
#include <string.h>

char *strerror(int errnum);
```

`strerror` bildet die in `errnum` übergebene Fehlernummer auf einen sprachabhängigen Meldungstext ab und liefert einen Zeiger auf diese Zeichenkette zurück.

Der zurückgelieferte Meldungstext kann auch Inserts enthalten:

- Wenn die im Parameter *errnum* übergebene Fehlernummer mit der aktuellen Fehlernummer übereinstimmt, dann werden Inserts berücksichtigt und in den Fehlermeldungstext aufgenommen. Die aktuelle Fehlernummer ist die in der Variablen `errno` abgelegte Fehlernummer.
- Andernfalls wird ein Meldungstext ohne Inserts zurückgeliefert, der zur in *errnum* übergebenen Fehlernummer passt.

Returnwert Zeiger auf einen C-internen Speicherbereich, der eine Zeichenkette mit dem Fehlermeldungstext enthält.

Hinweis Der Bereich, auf den `strerror` zeigt, darf nicht durch das Programm verändert werden. Er lässt sich nur durch wiederholte Aufrufe von `strerror` überschreiben.

Beispiel

```
#include <stdio.h>
#include <errno.h>
#include <string.h>

int main(void)

{
     printf("Fehlertext zu EDOM: %s\n", strerror(EDOM));
     return 0;
}
```

Siehe auch perror

strfill - Teil einer Zeichenkette kopieren

Definition

```
#include <string.h>

char *strfill(char *s1, const char *s2, size_t n);
```

`strfill` kopiert maximal *n* Zeichen aus der Zeichenkette *s2* in die Zeichenkette *s1*.

Je nach Länge bzw. Inhalt der Zeichenketten *s1*, *s2* und der Anzahl *n*, wird folgendermaßen kopiert:

1. Unabhängig von der Länge der Zeichenkette *s1*, werden (mit Ausnahme von Fall 5.) immer *n* Zeichen nach *s1* kopiert. D.h.:
   - Enthält *s1* mehr als *n* Zeichen, bleiben die restlichen rechten Zeichen in *s1* erhalten.
   - Enthält *s1* weniger als *n* Zeichen, wird *s1* bis zur Länge *n* verlängert. In diesem Fall ist *s1* nach dem Kopiervorgang nicht automatisch mit einem Nullbyte abgeschlossen (siehe auch Hinweise).
2. *s2* enthält weniger als *n* Zeichen

   Zusätzlich zu den kopierten Zeichen aus *s2* werden noch soviele Leerzeichen hinzugefügt, bis *n* Zeichen geschrieben wurden.
3. *s2* enthält mehr als *n* Zeichen

   Es werden nur die führenden *n* Zeichen aus *s2* kopiert.
4. *s2* ist leer

   s1 wird mit *n* Leerzeichen aufgefüllt.
5. *s2* wird als NULL-Zeiger übergeben

   Es werden (*n* - `strlen(s1)`) Leerzeichen an die Zeichenkette *s1* angehängt. Ergibt diese Subtraktion ein negatives Ergebnis oder 0, d.h. ist die Anzahl der Zeichen in *s1* größer oder gleich *n*, bleibt der Inhalt von *s1* unverändert.

Returnwert Zeiger auf die Ergebniszeichenkette *s1*.

Hinweise Als Argumente werden Zeichenketten erwartet, die mit dem Nullbyte (\0) abgeschlossen sind.

`strfill` überprüft nicht, ob *s1* groß genug für das Ergebnis ist und schließt die Ergebniszeichenkette nicht automatisch mit dem Nullbyte (\0) ab! Um kein unvorhergesehenes Ergebnis zu erhalten, sollten Sie nach jedem Aufruf von `strfill` die Zeichenkette *s1* explizit mit dem Nullbyte abschließen (siehe auch Beispiel).

Bei sich überlappenden Speicherbereichen ist das Verhalten undefiniert.

Beispiel

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    size_t n;
    char s1[10];
    char s2[10];
    printf("Bitte 2 Zeichenketten eingeben!\n");
    scanf("%s %s", s1, s2);
    printf("wie viele Zeichen kopieren?\n");
    scanf("%d", &n);
    strfill(s1, s2, n);
    /* strfill(s1, NULL, n);       Beispiel für Übergabe von s2 als
                                   NULL-Zeiger */
    *(s1 + n) = '\0';  /*Ergebniszeichenkette mit Nullbyte abschließen*/
    printf("s1 nach strfill: %s\n", s1);
    printf("Aktuelle Laenge von s1: %d\n", strlen(s1));
    return 0;
}
```

Siehe auch strncpy

strftime - Lokalitätsspezifische Darstellung von Datum und Uhrzeit

Definition

```
#include <time.h>

size_t strftime(char *s, size_t max_anz, const char *format,
                const struct tm *tm_zg);
```

`strftime` schreibt maximal *max_anz* Zeichen entsprechend den Angaben in der Zeichenkette *format* in den Bereich, auf den *s* zeigt.
Die Formatzeichenkette besteht aus beliebigen einfachen Zeichen und Umwandlungszeichen (beginnend mit %). Alle einfachen Zeichen, inklusive des abschließenden Nullbytes (\0), werden 1:1 in die Zeichenkette übernommen. Die Umwandlungszeichen werden durch passende Datums-/Uhrzeit-Informationen ersetzt. Diese Informationen werden durch die aktuelle Lokalität (Kategorie LC_TIME) und die Werte der Struktur bestimmt, auf die *tm_zg* zeigt.

Parameter

char *s
: Ergebniszeichenkette. Sie muss groß genug sein, um *max_anz* Zeichen, einschließlich des Nullbytes, aufnehmen zu können.

size_t max_anz
: Maximale Anzahl der Zeichen, einschließlich des Nullbytes, die in die Ergebniszeichenkette geschrieben werden sollen.

const char *format
: Formatzeichenkette, die einfache Zeichen und Umwandlungszeichen enthält. Die Umwandlungszeichen werden gemäß folgender Beschreibung durch lokalitätsspezifische und aktuelle Daten ersetzt:

%a	Abgekürzter lokalitätsspezifischer Name des Wochentages.
%A	Ausgeschriebener lokalitätsspezifischer Name des Wochentages.
%b	Abgekürzter lokalitätsspezifischer Name des Monats.
%B	Ausgeschriebener lokalitätsspezifischer Name des Monats.
%c	Lokalitätsspezifische Darstellung von Uhrzeit und Datum.
%d	Tag des Monats als Dezimalzahl (01 - 31).
%H	Stunde als Dezimalzahl (00 - 23). 24-Stunden-Darstellung.
%I	Stunde als Dezimalzahl (01 - 12). 12-Stunden-Darstellung.
%j	Tag des Jahres als Dezimalzahl (001 - 366).
%m	Monat als Dezimalzahl (01 - 12).
%M	Minuten als Dezimalzahl (00 - 59).
%p	Lokalitätsspezifisches Äquivalent zu AM und PM.

%S	Sekunden als Dezimalzahl (00 - 59).
%U	Wochennummer des Jahres (00 - 53). Die erste Woche beginnt mit dem ersten Sonntag des Jahres.
%w	Wochentag als Dezimalzahl (0 - 6). Sonntag ist 0.
%W	Wochennummer des Jahres (00 - 53). Die erste Woche beginnt mit dem ersten Montag des Jahres.
%x	Lokalitätsspezifische Darstellung des Datums.
%X	Lokalitätsspezifische Darstellung der Uhrzeit.
%y	Jahreszahl ohne Jahrhundertangabe als Dezimalzahl (00 - 99).
%Y	Jahreszahl mit Jahrhundertangabe.
%z	Name der Zeitzone oder kein Zeichen, falls die Zeitzone nicht bestimmbar ist.
%%	Das Zeichen %.

const struct tm *tm_zg
: Zeiger auf eine Struktur vom Typ `tm`, aus der `strftime` die Uhrzeit und das Datum entnehmen kann. Eine Struktur vom Typ `tm` liefern die Funktionen `gmtime`, `gmtime64`, `localtime`, `localtime64`, `mktime` und `mktime64`.

Returnwert Anzahl der geschriebenen Zeichen exklusive des abschließenden Nullbytes.
bei Erfolg.

0 falls ein Fehler auftritt. Wenn z.B. die Umwandlung mehr als *max_anz* Zeichen (inklusive des Nullbytes) ergibt.

Hinweise Bei sich überlappenden Speicherbereichen ist das Verhalten undefiniert.

Die zur Verfügung stehenden Lokalitäten sind im Kapitel „Lokalität“ auf Seite 99 beschrieben.

Siehe auch gmtime, gmtime64, localtime, localtime64, mktime, mktime64, setlocale

strlen - Länge einer Zeichenkette ermitteln

Definition

```
#include <string.h>

size_t strlen(const char *s);
```

`strlen` bestimmt die Länge der Zeichenkette *s*, das abschließende Nullbyte (\0) ausgeschlossen.

Während der `sizeof`-Operator immer die definierte Länge liefert, berechnet `strlen` die aktuelle Anzahl von Zeichen in einer Zeichenkette. Ein Neue-Zeile-Zeichen (\n) wird mitgezählt.

Returnwert Länge der Zeichenkette *s*.
Das abschließende Nullbyte wird nicht mitgezählt.

Hinweis Als Argument wird eine Zeichenkette erwartet, die mit dem Nullbyte (\0) abgeschlossen ist.

Beispiel 1 Dieses Programm liest eine Zeichenkette ein und berechnet ihren aktuellen Speicherplatzbedarf unter Berücksichtigung des Nullbytes (strlen + 1) sowie ihre definierte Länge (`sizeof(s)` = 8192 Bytes).

```
#include <stdio.h>
#include <string.h>

int main(void)
{
  char s[BUFSIZ];
  printf("Geben Sie bitte Ihre Zeichenkette ein.\n");
  scanf("%s", s);
  printf("Benoetigter Speicherplatz f. die Zeichenkette: %d\n", strlen(s)+1);
  printf("Definierter Speicherplatz f. die Zeichenkette: %d\n", sizeof(s));
  return 0;
}
```

Beispiel 2 Dieses Programm berechnet jeweils die aktuelle Satzlänge einer Datei (das Neue-Zeile-Zeichen '\n' mit eingeschlossen).

```
#include <stdio.h>
#include <string.h>

int main(void)
{
  FILE *fp;
  int n = 200, z = 0;
  char string[BUFSIZ];
  fp = fopen("input", "r");

  while (fgets(string, n, fp) != NULL)
  {
    z++;
    printf("der %d. Satz enthaelt %d Zeichen \n", z, strlen(string));
  }
  return 0;
}
```

strlower - Zeichenkette kopieren mit Umwandlung in Kleinbuchstaben

Definition

#include <string.h>

char *strlower(char *s1, const char *s2);

`strlower` kopiert die Zeichenkette *s2* einschließlich des Nullbytes (\0) in die Zeichenkette *s1* und wandelt dabei die Großbuchstaben in Kleinbuchstaben um.

Wird die Zeichenkette *s2* als NULL-Zeiger übergeben, entfällt der Kopiervorgang und es werden in *s1* die Großbuchstaben in Kleinbuchstaben umgewandelt.

Returnwert Zeiger auf die Ergebniszeichenkette *s1*.

Hinweise Als Argumente werden Zeichenketten erwartet, die mit dem Nullbyte (\0) abgeschlossen sind.

`strlower` überprüft nicht, ob *s1* groß genug für das Ergebnis ist. Ist *s1* kürzer als *s2* (einschließlich des Nullbytes), so wird der Speicherplatz hinter *s1* überschrieben!

Bei sich überlappenden Speicherbereichen ist das Verhalten undefiniert.

Beispiel Folgendes Programm kopiert den Inhalt von *s2* nach *s1* und wandelt dabei die Groß- in Kleinbuchstaben um.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    char s1[] = "                ";
    char s2[] = "GROSSBUCHSTABEN!";
    printf("Inhalt s2: %s\n", s2);

                /* s2 nach s1 kopieren mit Umwandlung in Kleinbuchstaben*/
    strlower(s1, s2);

    printf("Nach strlower:\nInhalt s1: %s\n", s1);
    return 0;
}
```

Siehe auch strupper, tolower, toupper

strncat - Verketten von Zeichenketten

Definition

```
#include <string.h>

char *strncat(char *s1, const char *s2, size_t n);
```

`strncat` hängt maximal *n* Zeichen der Zeichenkette *s2* ans Ende der Zeichenkette *s1* und liefert einen Zeiger auf *s1* zurück.

Das Nullbyte (\0) am Ende der Zeichenkette *s1* wird vom ersten Zeichen der Zeichenkette *s2* überschrieben.

Wenn die Zeichenkette *s2* weniger als *n* Zeichen enthält, werden nur die Zeichen aus *s2* an *s1* angehängt.
Wenn die Zeichenkette *s2* mehr als *n* Zeichen enthält, werden nur die führenden *n* Zeichen von *s2* an *s1* angehängt.

Returnwert Zeiger auf die Ergebniszeichenkette. `strncat` schließt die Zeichenkette mit dem Nullbyte (\0) ab.

Hinweise Als Argumente werden Zeichenketten erwartet, die mit dem Nullbyte (\0) abgeschlossen sind.

`strncat` überprüft nicht, ob *s1* groß genug für das Ergebnis ist!

Bei sich überlappenden Speicherbereichen ist das Verhalten undefiniert.

Beispiel

```
#include <string.h>
#include <stdio.h>
int main(void)
{
  char text1[BUFSIZ];
  char text2[BUFSIZ];
  int n;
  printf("Beispiel strncat - bitte 2 Textzeilen und n eingeben!\n");
  if(scanf("%s %s %d", text1, text2, &n) == 3)
     printf("%s\n", strncat(text1, text2, n));
  return 0;
}
```

Siehe auch strcat

strncmp - Vergleich von zwei Zeichenketten

Definition

```
#include <string.h>

int strncmp(const char *s1, const char *s2, size_t n);
```

`strncmp` vergleicht die Zeichenketten *s1* und *s2* bis zur maximalen Länge *n* lexikalisch, z.B liefert

```
strncmp("Sie","Siemens",3)
```

das Ergebnis 0 (gleich), weil die beiden Argumente in den ersten drei Zeichen übereinstimmen.

Returnwert

< 0	*s1* ist in den ersten *n* Zeichen lexikalisch kleiner als *s2*
0	*s1* und *s2* sind in den ersten *n* Zeichen lexikalisch gleich groß
> 0	*s1* ist in den ersten *n* Zeichen lexikalisch größer als *s2*

Hinweis

Als Argumente werden Zeichenketten erwartet, die mit dem Nullbyte (\0) abgeschlossen sind.

Beispiel In folgendem Rate-Programm wird `strncmp` dazu benutzt, die lexikalische Ordnung zweier Zeichenketten zu bestimmen.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    int i, n, result;
    char s[BUFSIZ], w[BUFSIZ];
    printf("Bitte geben Sie das zu ratende Wort ein:\n");
    scanf("%s", w);
    n = strlen(w);
    printf("\nDas eingegebene Wort hat %d Buchstaben.\n", n);
    i = 0;
    do
    {
       i++;
       printf("Ihr Versuch: \n");
       scanf("%s", s);
       if (strlen(s) > n)
       {
          printf("Ihre Eingabe ist zu lang!\n");
          continue;
       }
       result = strncmp(s, w, n);         /* result wird das Ergebnis
                                             von strncmp zugewiesen */
       if (result > 0)
           printf("%s ist lexikalisch groesser.\n", s);
       else
       {
         if (result < 0)
             printf("%s ist lexikalisch kleiner.\n", s);
       }
    }
    while (result != 0);
    printf("Richtig! Das Wort hiess : %s\n", w);
    printf("Sie haben %d Versuche gebraucht.\n", i);
    return 0;
}
```

Siehe auch strcmp

strncpy - Zeichenkette kopieren

Definition

```
#include <string.h>

char *strncpy(char *s1, const char *s2, size_t n);
```

`strncpy` kopiert maximal *n* Zeichen der Zeichenkette *s2* in die Zeichenkette *s1*.

Enthält die Zeichenkette *s2* weniger als *n* Zeichen, wird nur in der Länge von *s2* (`strlen` + 1) kopiert.

Enthält die Zeichenkette *s2* *n* Zeichen (exklusive Nullbyte) oder mehr, ist die Zeichenkette *s1* nicht automatisch mit dem Nullbyte abgeschlossen.

Enthält die Zeichenkette *s1* mehr als *n* Zeichen und das letzte kopierte Zeichen aus *s2* ist nicht das Nullbyte, bleiben ggf. restliche Daten in *s1* erhalten.

Returnwert Zeiger auf die Ergebniszeichenkette *s1*.
`strncpy` schließt *s1* nicht automatisch mit dem Nullbyte ab.

Hinweise `strncpy` überprüft nicht, ob *s1* groß genug für das Ergebnis ist!

Da `strncpy` die Ergebniszeichenkette nicht automatisch mit dem Nullbyte abschließt, kann es häufig notwendig sein, *s1* explizit mit einem Nullbyte abzuschließen. Das ist z.B. der Fall, wenn nur ein Teilstück aus *s2* kopiert wird und auch *s2* kein Nullbyte enthält.

Bei sich überlappenden Speicherbereichen ist das Verhalten undefiniert.

Beispiel 1 Kopiert die gesamte Zeichenkette *s2* in die Zeichenkette *s1* (wie die Funktion `strcpy`).

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    int n;
    char s1[20];
    char s2[20];
    printf("Bitte s2 (max. 19 Zeichen) eingeben\n");
    scanf("%s", s2);
    printf("s1: %s\n", strncpy(s1, s2, (strlen(s2) + 1)));
    return 0;
}
```

Beispiel 2 Kopiert nur ein Teilstück (10 Zeichen) von *s2* nach *s1*. Die Ergebniszeichenkette wird explizit mit dem Nullbyte abgeschlossen.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
  char *s1 = "                        ";
  char *s2 = "Peter geht schwimmen !";
  strncpy(s1, s2, 10);
  *(s1 + 10) = '\0';
  printf("s1: %s\n", s1);    /* Inhalt von s1: "Peter geht" */
  return 0;
}
```

Beispiel 3 Kopiert nur ein Teilstück (5 Zeichen) von *s2* nach *s1*. Die restlichen Daten in *s1* bleiben erhalten.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
  char *s1 = "Xaver geht arbeiten !";
  char *s2 = "Peter geht schwimmen !";
  strncpy(s1, s2, 5);
  printf("s1: %s\n", s1);    /* Inhalt von s1: "Peter geht arbeiten !" */
  return 0;
}
```

Siehe auch strcpy, strlen

strpbrk - Zeichen in Zeichenkette suchen

Definition #include <string.h>

char *strpbrk(const char *s1, const char *s2);

`strpbrk` sucht das erste Zeichen in der Zeichenkette *s1*, das mit irgendeinem Zeichen aus der Zeichenkette *s2* übereinstimmt. Das abschließende Nullbyte (\0) gilt nicht als Teil der Zeichenkette *s2*.

Returnwert Zeiger auf das erste gefundene Zeichen in *s1*
bei Erfolg.

NULL-Zeiger falls keinerlei Übereinstimmung vorliegt.

Hinweise Als Argumente werden Zeichenketten erwartet, die mit dem Nullbyte (\0) abgeschlossen sind.

Für C++ gelten die beiden folgenden Prototypen für die Funktion `strpbrk`:
const char *strpbrk(const char *s1, const char *s2);
char *strpbrk(char *s1, const char *s2);

Beispiel

```
#include <string.h>
#include <stdio.h>

int main(void)
{
  char text1[40];
  static char text2[] = "0123456789";
  char *result;
  printf("Beispiel fuer strpbrk()\n");
  printf("bitte eine Zeichenkette (max. 40 Zeichen) eingeben!\n");
  scanf("%s",text1);
  result = strpbrk(text1,text2);
  if(result == NULL)
    printf("Die eingegebene Zeichenkette enthaelt keine Ziffer.\n");
  else printf("%s\n", result);
  return 0;
}
```

Siehe auch index, strchr

strptime - Zeichenkette in Datum und Uhrzeit umwandeln

Definition #include <time.h>

char *strptime(const char *buf, const char *format, struct tm *tm);

`strptime` konvertiert unter Berücksichtigung von *format* die Zeichenkette, auf die **buf* zeigt, in Datum- und Uhrzeit-Einzelwerte, die in der Struktur abgelegt werden, auf die **tm* zeigt.

Parameter const char *buf

Datum- und Uhrzeit-Zeichenkette, die konvertiert werden soll.

struct tm *tm

Ergebnisstruktur, in der die konvertierten Datum- und Uhrzeit-Einzelwerte abgelegt werden.
Die Struktur wird von `strptime` nicht mit Nullen initialisiert. Die vom Anwender vorbelegten Werte bleiben erhalten, sofern sie nicht durch Umwandlungsanweisungen oder durch implizite Berechnungen neu belegt werden.
Das Strukturelement `tm_isdst` wird in keinem Fall verändert.
Gegebenenfalls erfolgt implizit ein Datumsabgleich, d.h. bei unvollständigen Datumsangaben werden nicht angegebene Strukturelemente ergänzt und es wird die Plausibilität der Strukturelemente untereinander überprüft. Dies erfolgt aber nur, wenn über %U bzw. %W eine Wochennummer eingegeben wurde. In diesem Fall werden mithilfe der Jahresangabe (`tm_year`) und des Wochentages (`tm_wday`) der Tag im Jahr (`tm_yday`), der Tag des Monats (`tm_mday`) und der Monat des Jahres (`tm_mon`) berechnet und neu zugewiesen. Der Wochentag (`tm_day`) wird hierbei mit dem Wert 0 belegt, sofern er nicht explizit mit %w, %a oder %A angegeben wurde.

const char *format

Die Zeichenkette *format* besteht aus null, einer oder mehreren Umwandlungsanweisungen. Jede Umwandlungsanweisung besteht aus einem der folgenden Elemente:
einem oder mehreren Zwischenraumzeichen (wie in `isspace()` definiert)
einem gewöhnlichen Zeichen (weder % noch Zwischenraumzeichen)
oder einer Konvertierungs-Spezifikation.

Jede Konvertierungs-Spezifikation besteht aus einem %-Zeichen, gefolgt von einem Konvertierungszeichen, das die gewünschte Umwandlung angibt. Zwischen zwei Konvertierungs-Spezifikationen muss ein Zwischenraum-Zeichen oder ein nicht-alphanumerisches Zeichen stehen.

Folgende Konvertierungs-Zeichen werden unterstützt:

%%	wird ersetzt durch %
%a	Wochentag, wobei die Namen aus der Lokalität verwendet werden. Es kann entweder der abgekürzte oder der ausgeschriebene Name angegeben werden
%A	gleiche Bedeutung wie %a
%b	Monat, wobei die Namen aus der Lokalität verwendet werden. Es kann entweder der abgekürzte oder der ausgeschriebene Name angegeben werden
%B	gleiche Bedeutung wie %b
%c	Datums- und Zeitdarstellung entsprechend der Definition in der Lokalität
%C	Jahrhundert (vierstellige Jahreszahl geteilt durch 100 als ganze Zahl) (00-99)
%d	Monatstag (01-31)
%D	Datum als %m/%d/%y
%e	gleiche Bedeutung wie %d
%h	gleiche Bedeutung wie %b
%H	Stunde (00-23), 24-Stunden Darstellung
%I	Stunde (01-12), 12-Stunden Darstellung
%j	Tag des Jahres (001-366)
%m	Nummer des Monats (01-12)
%M	Minute (00-59)
%n	wird ersetzt durch ein Zwischenraum-Zeichen
%p	äquivalente Bezeichnung der Lokalität für AM oder PM
%r	Zeit im Format %I:%M:%S%p
%R	Zeit im Format %H:%M
%S	Sekunden (00-61), erlaubt Schaltsekunden
%t	wird ersetzt durch ein Zwischenraum-Zeichen
%T	Zeit im Format %H:%M:%S
%U	Nummer der Woche im Jahr (00-53). Die erste Woche beginnt mit dem ersten Sonntag des Jahres. Alle Tage vor dem ersten Sonntag des Jahres gehören zur Woche 0.
%w	Wochentag als Zahl (0-6), Sonntag = 0
%W	Nummer der Woche im Jahr (00-53), Montag ist der erste Tag der Woche 1. Alle Tage vor dem ersten Montag des Jahres gehören zur Woche 0.
%x	Datum in der Darstellung der Lokalität
%X	Zeit in der Darstellung der Lokalität

`%y` zweistellige Jahreszahl (00-99).
Jahreszahlen zwischen 00 und 68 werden dabei als Jahre 2000 bis 2068 interpretiert, Jahreszahlen zwischen 69 und 99 als Jahre 1969 bis 1999.

`%Y` vierstellige Jahreszahl in der Form ccyy (z.B. 1966 oder 2001)

Eine Umwandlungsanweisung, die aus Zwischenraum-Zeichen besteht, wird ausgeführt, indem der Input bis zum ersten Zeichen gelesen wird, das kein Zwischenraum-Zeichen ist (dieses Zeichen bleibt ungelesen), oder bis keine Zeichen mehr vorhanden sind.

Eine Umwandlungsanweisung, die aus einem gewöhnlichen Zeichen besteht, wird ausgeführt, indem das nächste Zeichen aus dem Puffer gelesen wird. Wenn das aus dem Puffer gelesene Zeichen nicht mit dem Zeichen der Umwandlungsanweisung übereinstimmt, schlägt diese fehl und das abweichende Zeichen sowie alle weiteren Zeichen bleiben ungelesen.

Eine Folge von Umwandlungsanweisungen, die aus %n, %t, Zwischenraum-Zeichen und Kombinationen davon besteht, wird ausgeführt, indem bis zum ersten Zeichen gelesen wird, das kein Zwischenraum-Zeichen ist (dieses Zeichen bleibt ungelesen), oder bis keine Zeichen mehr vorhanden sind.

Alle anderen Konvertierungs-Spezifikationen werden ausgeführt, indem solange Zeichen eingelesen werden, bis ein zur nächsten Umwandlungsanweisung passendes Zeichen gelesen wird (dieses bleibt im Puffer) oder bis keine Zeichen mehr vorhanden sind. Die gelesenen Zeichen werden dann mit den Werten in der Lokalität verglichen, die der Konvertierungs-Spezifikation entsprechen. Wenn der passende Wert in der Lokalität gefunden wird, werden die entsprechenden Strukturelemente der `tm`-Struktur auf die dieser Information entsprechenden Werte gesetzt.
Groß-/Kleinschreibung wird bei der Suche ignoriert, wenn es sich um den Vergleich von Elementen wie Wochentags- und Monatsnamen handelt.
Wenn kein passender Wert in der Lokalität gefunden wird, schlägt `strptime()` fehl und es werden keine weiteren Zeichen gelesen.

Returnwert Zeiger auf das Zeichen hinter dem letzten gelesenen Zeichen
bei Erfolg.

Nullzeiger sonst.

Hinweis Die spezielle Behandlung von Zwischenraum-Zeichen und viele „gleiche Formate“ sollen den Einsatz von identischen Format-Strings bei `strftime()` und `strptime()` erleichtern.

Siehe auch scanf, strftime, time.

strrchr - Letztes Vorkommen eines Zeichens in einer Zeichenkette

Definition #include <string.h>

char *strrchr(const char *s, int c);

`strrchr` sucht das letzte Vorkommen des Zeichens *c* in der Zeichenkette *s* und liefert bei Erfolg einen Zeiger auf die gesuchte Position in *s*.

Das abschließende Nullbyte (\0) wird als Zeichen mitberücksichtigt.

Returnwert Zeiger auf die Position von *c* in der Zeichenkette *s*
bei Erfolg.

NULL-Zeiger wenn *c* in der Zeichenkette *s* nicht enthalten ist.

Hinweise Die Funktionen `strrchr` und `rindex` sind äquivalent.

Für C++ gelten die beiden folgenden Prototypen für die Funktion `strchr`:
const char *strrchr(const char *s, int c);
char *strrchr(char *s, int c);

Beispiel Finde das letzte 'k':

```
#include <string.h>
#include <stdio.h>

int main(void)
{
   char *s = "was fuer ein Spass im kkuehlen Nass!";
   printf("%s\n", s);
   printf("Wo steckt der Fehler? %s\n", strrchr(s, 'k'));
   return 0;
}
```

Siehe auch index, rindex, strchr

strspn - Zeichenketten vergleichen und Segmentlänge berechnen

Definition #include <string.h>

size_t strspn(const char *s1, const char *s2);

`strspn` berechnet ab Beginn der Zeichenkette *s1* die Länge des Segmentes, das ausschließlich Zeichen aus der Zeichenkette *s2* enthält.

Sobald ein Zeichen in *s1* mit keinem Zeichen in *s2* übereinstimmt, wird die Funktion beendet und die Segmentlänge zurückgeliefert.

Stimmt bereits das erste Zeichen in *s1* mit keinem Zeichen in *s2* überein, ist die Segmentlänge gleich 0.

Returnwert Ganzzahl die die Segmentlänge (Anzahl gleicher Zeichen) ab Beginn der Zeichenkette *s1* angibt.

Hinweis Als Argumente werden Zeichenketten erwartet, die mit dem Nullbyte (\0) abgeschlossen sind.

Beispiel
```
#include <stdio.h>
#include <string.h>
int main(void)
{
 char text1[40];
 char *text2 = "0123456789";
 size_t n;
 printf("Beispiel fuer strspn. Bitte eine Textzeile eingeben:\n");
 scanf("%s", text1);
 n = strspn(text1, text2);
 printf("Laenge des Anfangssegments mit Ziffern (0 - 9): %d\n", n);
 return 0;
}
```

Siehe auch strcspn

strstr - Erstes Vorkommen einer Zeichenkette in einer anderen

Definition #include <string.h>

char *strstr(const char *s1, const char *s2);

`strstr` sucht das erste Vorkommen der Zeichenkette *s2* (ohne das abschließende Nullbyte) in der Zeichenkette *s1*

Returnwert Zeiger auf den Beginn der gefundenen Zeichenkette in *s1*
wenn *s2* in *s1* enthalten ist.

0 wenn *s2* in *s1* nicht enthalten ist.

Zeiger auf den Beginn von *s1*
wenn *s2* die Länge 0 hat.

Hinweise Als Argumente werden Zeichenketten erwartet, die mit dem Nullbyte abgeschlossen sind.

Für C++ gelten die beiden folgenden Prototypen für die Funktion `strstr`:
const char *strstr(const char *s1, const char *s2);
char *strstr(char *s1, const char *s2);

Beispiel
```
#include <string.h>
#include <stdio.h>

int main(void)
{
   char *s1 = "Ort: Muenchen, Name: Peter Mueller";
   char *s2 = "Peter";
   printf("Vollstaendiger Name? %s\n", strstr(s1, s2));  /* Peter Mueller */
   return 0;
}
```

Siehe auch strchr

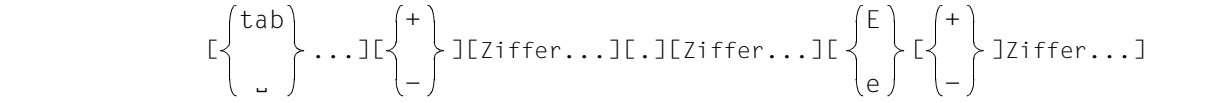
strtod - Zeichenkette in Gleitkommazahl umwandeln

Definition

```
#include <stdlib.h>

double strtod(const char *s, char **zg);
```

`strtod` wandelt eine Zeichenkette, auf die *s* zeigt, in eine Gleitkommazahl vom Typ `double` um. Die umzuwandelnde Zeichenkette kann wie folgt aufgebaut sein:



Für *tab* sind alle Steuerzeichen für „Zwischenraum“ zulässig (siehe Definition bei `isspace`).

`strtod` erkennt auch Zeichenketten, die mit Ziffern beginnen, dann aber mit beliebigen Zeichen enden. In diesem Fall schneidet `strtod` zunächst den Ziffernteil ab und wandelt ihn in einen Gleitkommawert um.

Wenn *zg* nicht als NULL-Zeiger übergeben wird, erhält man von `strtod` über das zweite Argument *zg* vom Typ `char **` einen Zeiger (**zg*) auf das erste nicht umwandelbare Zeichen in der Zeichenkette *s*. Wenn überhaupt keine Umwandlung möglich ist, wird **zg* auf die Anfangsadresse der Zeichenkette *s* gesetzt.

Ist *zg* ein NULL-Zeiger, wird `strtod` wie die Funktion `atof` ausgeführt:

`atof(s)` enspricht `strtod(s, (char **)NULL)` oder auch `strtod(s, NULL)`.

Returnwert

Gleitkommazahl vom Typ `double`	für Zeichenketten, die eine wie oben beschriebene Struktur haben und einen Zahlenwert darstellen, der im zulässigen Gleitkommabereich liegt.
0	für Zeichenketten, die nicht der oben beschriebenen Syntax entsprechen bzw. nicht mit umwandelbaren Zeichen beginnen.
HUGE_VAL	für Zeichenketten, deren Zahlenwert außerhalb des zulässigen Gleitkommabereichs liegt. Zusätzlich wird `errno` auf ERANGE gesetzt (Resultat zu groß).

Hinweis

Das Dezimalpunktzeichen (Punkt oder Komma) in der umzuwandelnden Zeichenkette wird durch die Lokalität (Kategorie LC_NUMERIC) beeinflusst. Voreingestellt ist der Punkt.

Beispiel Folgendes Programm wandelt eine beim Aufruf (Enter Options) übergebene Zeichenkette in die entsprechende Gleitkommazahl um und gibt das erste ggf. nicht umwandelbare Zeichen aus.

```
#include <stdio.h>
#include <stdlib.h>

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

                /* Zahlen werden als Zeichenketten!! übergeben.
                   Eine Umwandlung ist erforderlich,
                   falls der Zahlenwert benötigt wird */
{
  char *p;

  printf("floating : %f\n", strtod(argv[1], &p));
  putchar(*p);
  return 0;
}
```

Siehe auch atof, atoi, atol, strtol, strtoul

strtok - Zeichenkette in Teilzeichenketten zerlegen

Definition #include <string.h>

char *strtok(char *s1, const char *s2);

Mit `strtok` lässt sich eine Gesamtzeichenkette *s1* in Teilzeichenketten - sog. „Tokens“ - zerlegen, z.B. ein Satz in die einzelnen Wörter oder eine Quellprogrammanweisung in die kleinsten syntaktischen Einheiten.
Beginn- und Endekriterium für jedes Token sind Trennzeichen (Separatoren), die Sie in einer zweiten Zeichenkette *s2* angeben. Tokens können durch einen oder mehrere dieser Separatoren begrenzt sein, bzw. durch Beginn und Ende der Gesamtzeichenkette *s1*. Typische Separatoren zwischen den Wörtern eines Satzes sind z.B. Leerzeichen, Doppelpunkt, Komma etc. Pro Aufruf bzw. Token kann eine andere Zeichenfolge *s2* angegeben werden.

Pro Aufruf bearbeitet `strtok` genau eine Teilzeichenkette. Der erste Aufruf liefert einen Zeiger auf den Beginn der ersten gefundenen Teilzeichenkette, die weiteren Aufrufe jeweils einen Zeiger auf den Beginn der nächsten Teilzeichenketten. Jede Teilzeichenkette schließt `strtok` mit dem Nullbyte (\0) ab.

Damit `strtok` die Gesamtzeichenkette *s1* sukzessive abarbeitet, darf nur beim ersten Aufruf die Anfangadresse, d.h. ein Zeiger auf *s1* übergeben werden. Bei allen weiteren Aufrufen ist *s1* als NULL-Zeiger zu übergeben.

Returnwert Zeiger auf den Beginn einer Teilzeichenkette.
bei Erfolg. Beim ersten Aufruf ein Zeiger auf die erste Teilzeichenkette, beim nächsten Aufruf ein Zeiger auf die nachfolgende Teilzeichenkette etc. `strtok` schließt jede Teilzeichenkette in *s1* mit einem Nullbyte (\0) ab, wobei das jeweils erste gefundene Trennzeichen mit \0 überschrieben wird.

NULL-Zeiger falls keine bzw. keine weitere Teilzeichenkette gefunden wurde.

Beispiel
```
#include <string.h>
#include <stdio.h>
int main(void)
{
  static char str[] = "?a???b,,,#c";
  char *t;
  t = strtok(str, "?");        /* t zeigt auf das Token "a" */
  t = strtok(NULL, ",");       /* t zeigt auf das Token "??b" */
  t = strtok(NULL, "#,");      /* t zeigt auf das Token "c" */
  t = strtok(NULL, "?");       /* t ist ein NULL-Zeiger */
  return 0;
}
```

Siehe auch strtok

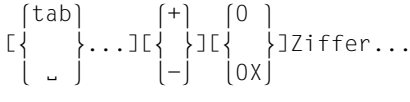
strtol - Zeichenkette in ganze Zahl umwandeln (long int)

Definition

```
#include <stdlib.h>

long int strtol(const char *s, char **zg, int base);
```

`strtol` wandelt eine Zeichenkette, auf die *s* zeigt, in eine ganze Zahl vom Typ `long int` um. Die umzuwandelnde Zeichenkette kann wie folgt aufgebaut sein:



Für *tab* sind alle Steuerzeichen für „Zwischenraum“ zulässig (siehe Definition bei `isspace`).

Für *Ziffer* sind je nach der Basis (siehe *base*) die Ziffern 0 bis 9 und die Buchstaben a (oder A) bis z (oder Z) zulässig.

`strtol` erkennt auch Zeichenketten, die mit umwandelbaren Ziffern (auch Oktal- bzw. Sedezimal-Ziffern) beginnen, dann aber mit beliebigen Zeichen enden. In diesem Fall schneidet `strtol` zunächst den Ziffernteil ab und wandelt ihn um.

Zusätzlich erhält man von `strtol` über das zweite Argument *zg* vom Typ `char **` einen Zeiger auf das erste nicht umwandelbare Zeichen in der Zeichenkette *s*; jedoch nur, wenn *zg* nicht als NULL-Zeiger übergeben wird.

Ein drittes Argument *base* bestimmt die Basis (z.B. Dezimal-, Oktal- oder Sedezimal-Basis) für die Umwandlung.

Parameter

const char *s
: Zeiger auf die umzuwandelnde Zeichenkette.

char **zg
: Wenn *zg* kein NULL-Zeiger ist, wird ein Zeiger (**zg*) auf das erste Zeichen in *s* zurückgeliefert, das die Umwandlung beendet.
Wenn überhaupt keine Umwandlung möglich ist, wird **zg* auf die Anfangsadresse der Zeichenkette *s* gesetzt.

int base
: Ganze Zahl von 0 bis 36, die als Basis für die Berechnung verwendet werden soll.

  Von Basis 11 bis 36 werden die Buchstaben a (oder A) bis z (oder Z) in der umzuwandelnden Zeichenkette als Ziffern angenommen, und zwar mit den entsprechenden Werten 10 (a/A) bis 35 (z/Z).

Falls *base* gleich 0 ist, wird die Basis folgendermaßen aus dem Aufbau der Zeichenkette *s* bestimmt:

führende 0	Basis 8
führendes 0X bzw. 0x	Basis 16
sonst	Basis 10

Falls mit Parameter *base* = 16 gerechnet wird, werden die Zeichen 0X bzw. 0x nach einem evtl. Vorzeichen in der Zeichenkette *s* ignoriert.

Returnwert Ganzzahliger Wert vom Typ `long int`
für Zeichenketten, die eine wie oben beschriebene Struktur haben und einen Zahlenwert darstellen.

0 für Zeichenketten, die nicht der oben beschriebenen Syntax entsprechen. Es wird keine Konvertierung durchgeführt. Wenn der Wert von *base* nicht unterstützt wird, wird `errno` auf EINVAL gesetzt.

LONG_MAX, LONG_MIN
abgängig vom Vorzeichen des Wertes.

ULONG_MAX wenn der richtige Wert außerhalb des Bereichs der darstellbaren Werte liegt. `errno` wird auf ERANGE gesetzt, um den Fehler anzuzeigen.

Hinweis Ist *zg* ein NULL-Zeiger und *base* gleich 10, wird `strtol` wie die Funktion `atol` ausgeführt: `atol(s)` entspricht `strtol(s, NULL, 10)`.

Beispiel

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  char *str1 = "  0x1ff";
  char *str2 = "h0***";
  char *end;
  long l;

  l = strtol(str1, &end, 0);          /* Die Basis 16 wird aus der */
                                      /* Zeichenkette str1 abgeleitet. */
  printf("Erster Wert: %ld\n", l);    /* 511 wird ausgegeben. */

  l = strtol(str2, &end, 20);         /* Basis = 20 */
  printf("Zweiter Wert: %ld\n", l);   /* 340 (17*20) wird ausgegeben. */
  printf("Rest von str2: %s\n", end); /* "***" wird ausgegeben. */
  return 0;
}
```

Siehe auch atol, atoi, strtod, strtoll, strtoul, strtoull, wcstol, wcstoll, wcstoul, wcstoull

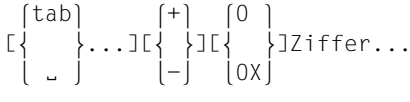
strtoll - Zeichenkette in ganze Zahl umwandeln (long long int)

Definition

```
#include <stdlib.h>

long long int strtoll(const char restrict *s, char ** restrict zg, int base);
```

`strtoll` wandelt eine Zeichenkette, auf die *s* zeigt, in eine ganze Zahl vom Typ `long long int` um. Die umzuwandelnde Zeichenkette kann wie folgt aufgebaut sein:



Für *tab* sind alle Steuerzeichen für „Zwischenraum“ zulässig (siehe Definition bei `isspace`).

Für *Ziffer* sind je nach der Basis (siehe *base*) die Ziffern 0 bis 9 und die Buchstaben a (oder A) bis z (oder Z) zulässig.

`strtoll` erkennt auch Zeichenketten, die mit umwandelbaren Ziffern (auch Oktal- bzw. Sedezimal-Ziffern) beginnen, dann aber mit beliebigen Zeichen enden. In diesem Fall schneidet `strtoll` zunächst den Ziffernteil ab und wandelt ihn um.

Zusätzlich erhält man von `strtoll` über das zweite Argument *zg* vom Typ `char **` einen Zeiger auf das erste nicht umwandelbare Zeichen in der Zeichenkette *s*; jedoch nur, wenn *zg* nicht als NULL-Zeiger übergeben wird.

Ein drittes Argument *base* bestimmt die Basis (z.B. Dezimal-, Oktal- oder Sedezimal-Basis) für die Umwandlung.

Parameter

const char *s
: Zeiger auf die umzuwandelnde Zeichenkette.

char **zg
: Wenn *zg* kein NULL-Zeiger ist, wird ein Zeiger (**zg*) auf das erste Zeichen in *s* zurückgeliefert, das die Umwandlung beendet.
Wenn überhaupt keine Umwandlung möglich ist, wird **zg* auf die Anfangsadresse der Zeichenkette *s* gesetzt.

int base
: Ganze Zahl von 0 bis 36, die als Basis für die Berechnung verwendet werden soll.

  Von Basis 11 bis 36 werden die Buchstaben a (oder A) bis z (oder Z) in der umzuwandelnden Zeichenkette als Ziffern angenommen, und zwar mit den entsprechenden Werten 10 (a/A) bis 35 (z/Z).

Falls *base* gleich 0 ist, wird die Basis folgendermaßen aus dem Aufbau der Zeichenkette *s* bestimmt:

führende 0	Basis 8
führendes 0X bzw. 0x	Basis 16
sonst	Basis 10

Falls mit Parameter *base* = 16 gerechnet wird, werden die Zeichen 0X bzw. 0x nach einem evtl. Vorzeichen in der Zeichenkette *s* ignoriert.

Returnwert Ganzzahliger Wert vom Typ `long long int`
für Zeichenketten, die eine wie oben beschriebene Struktur haben und einen Zahlenwert darstellen.

0 für Zeichenketten, die nicht der oben beschriebenen Syntax entsprechen. Es wird keine Konvertierung durchgeführt. Wenn der Wert von *base* nicht unterstützt wird, wird `errno` auf EINVAL gesetzt.

LLONG_MAX bzw. LLONG_MIN
abhängig vom Vorzeichen.

ULLONG_MAX
bei Überlauf. `errno` wird auf ERANGE gesetzt.

Hinweise Ist *zg* ein NULL-Zeiger und *base* gleich 10, unterscheidet sich `strtoll` von der Funktion `atoll` nur durch die Fehlerbehandlung.

`atoll(s)` entspricht `strtoll(s, (char **)NULL, 10)`.

Siehe auch atol, atoll, atoi, strtol, stroul, stroull, wcstol, wcstoll, wcstoul, wcstoull

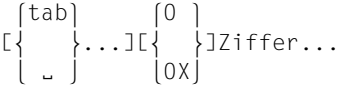
strtoul - Zeichenkette in ganze Zahl umwandeln (unsigned long int)

Definition

```
#include <stdlib.h>

unsigned long int strtoul(const char *s, char **zg, int base);
```

`strtoul` wandelt eine Zeichenkette, auf die *s* zeigt, in eine ganze Zahl vom Typ `unsigned long` int um. Die umzuwandelnde Zeichenkette kann wie folgt aufgebaut sein:



Für *tab* sind alle Steuerzeichen für „Zwischenraum“ zulässig (siehe Definition bei `isspace`).

Für *Ziffer* sind je nach der Basis (siehe *base*) die Ziffern 0 bis 9 und die Buchstaben a (oder A) bis z (oder Z) zulässig.

`strtoul` erkennt auch Zeichenketten, die mit umwandelbaren Ziffern (auch Oktal- bzw. Sedezimal-Ziffern) beginnen, dann aber mit beliebigen Zeichen enden. In diesem Fall schneidet `strtoul` zunächst den Ziffernteil ab und wandelt ihn um.

Zusätzlich erhält man von `strtoul` über das zweite Argument *zg* vom Typ `char` ** einen Zeiger auf das erste nicht umwandelbare Zeichen in der Zeichenkette *s*; jedoch nur, wenn *zg* nicht als NULL-Zeiger übergeben wird.

Ein drittes Argument *base* bestimmt die Basis (z.B. Dezimal-, Oktal- oder Sedezimal-Basis) für die Umwandlung.

Parameter

const char *s
: Zeiger auf die umzuwandelnde Zeichenkette.

char **zg
: Wenn *zg* kein NULL-Zeiger ist, wird ein Zeiger (**zg*) auf das erste Zeichen in *s* zurückgeliefert, das die Umwandlung beendet.
Wenn überhaupt keine Umwandlung möglich ist, wird **zg* auf die Anfangsadresse der Zeichenkette *s* gesetzt.

int base
: Ganze Zahl von 0 bis 36, die als Basis für die Berechnung verwendet werden soll.

  Von Basis 11 bis 36 werden die Buchstaben a (oder A) bis z (oder Z) in der umzuwandelnden Zeichenkette als Ziffern angenommen, und zwar mit den entsprechenden Werten 10 (a/A) bis 35 (z/Z).

Falls *base* gleich 0 ist, wird die Basis folgendermaßen aus dem Aufbau der Zeichenkette *s* bestimmt:

führende 0	Basis 8
führendes 0X bzw. 0x	Basis 16
sonst	Basis 10

Falls mit Parameter *base* = 16 gerechnet wird, werden die Zeichen 0X bzw. 0x in der Zeichenkette *s* ignoriert.

Returnwert Ganzzahliger Wert vom Typ `unsigned long`
für Zeichenketten, die eine wie oben beschriebene Struktur haben und einen Zahlenwert darstellen.

0 für Zeichenketten, die nicht der oben beschriebenen Syntax entsprechen. Es wird keine Konvertierung durchgeführt. Wenn der Wert von *base* nicht unterstützt wird, wird `errno` auf EINVAL gesetzt.

LONG_MAX, LONG_MIN
abgängig vom Vorzeichen des Wertes.

ULONG_MAX wenn der richtige Wert außerhalb des Bereichs der darstellbaren Werte liegt. `errno` wird auf ERANGE gesetzt, um den Fehler anzuzeigen.

Siehe auch atol, atoll, atoi, strtol, strtoll, stroull, wcstol, wcstoll, wcstoul, wcstoull

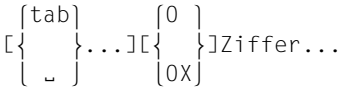
strtoull - Zeichenkette in ganze Zahl umwandeln (unsigned long long)

Definition

```
#include <stdlib.h>

unsigned long long int strtoull(const char restrict *s, char **restrict zg, int base);
```

`strtoull` wandelt eine Zeichenkette, auf die *s* zeigt, in eine ganze Zahl vom Typ `unsigned` long `long int` um. Die umzuwandelnde Zeichenkette kann wie folgt aufgebaut sein:



Für *tab* sind alle Steuerzeichen für „Zwischenraum“ zulässig (siehe Definition bei `isspace`).

Für *Ziffer* sind je nach der Basis (siehe *base*) die Ziffern 0 bis 9 und die Buchstaben a (oder A) bis z (oder Z) zulässig.

`strtoull` erkennt auch Zeichenketten, die mit umwandelbaren Ziffern (auch Oktal- bzw. Sedezimal-Ziffern) beginnen, dann aber mit beliebigen Zeichen enden. In diesem Fall schneidet `strtoull` zunächst den Ziffernteil ab und wandelt ihn um.

Zusätzlich erhält man von `strtoull` über das zweite Argument *zg* vom Typ `char` ** einen Zeiger auf das erste nicht umwandelbare Zeichen in der Zeichenkette *s*; jedoch nur, wenn *zg* nicht als NULL-Zeiger übergeben wird.

Ein drittes Argument *base* bestimmt die Basis (z.B. Dezimal-, Oktal- oder Sedezimal-Basis) für die Umwandlung.

Parameter

const char *s
: Zeiger auf die umzuwandelnde Zeichenkette.

char **zg
: Wenn *zg* kein NULL-Zeiger ist, wird ein Zeiger (**zg*) auf das erste Zeichen in *s* zurückgeliefert, das die Umwandlung beendet.
Wenn überhaupt keine Umwandlung möglich ist, wird **zg* auf die Anfangsadresse der Zeichenkette *s* gesetzt.

int base
: Ganze Zahl von 0 bis 36, die als Basis für die Berechnung verwendet werden soll.

  Von Basis 11 bis 36 werden die Buchstaben a (oder A) bis z (oder Z) in der umzuwandelnden Zeichenkette als Ziffern angenommen, und zwar mit den entsprechenden Werten 10 (a/A) bis 35 (z/Z).

Falls *base* gleich 0 ist, wird die Basis folgendermaßen aus dem Aufbau der Zeichenkette *s* bestimmt:

führende 0	Basis 8
führendes 0X bzw. 0x	Basis 16
sonst	Basis 10

Falls mit Parameter *base* = 16 gerechnet wird, werden die Zeichen 0X bzw. 0x in der Zeichenkette *s* ignoriert.

Returnwert Ganzzahliger Wert vom Typ `unsigned long long int`

für Zeichenketten, die eine wie oben beschriebene Struktur haben und einen Zahlenwert darstellen.

0 für Zeichenketten, die nicht der oben beschriebenen Syntax entsprechen. Es wird keine Konvertierung durchgeführt. Wenn der Wert von *base* nicht unterstützt wird, wird `errno` auf EINVAL gesetzt.

LLONG_MAX bzw. LLONG_MIN
abhängig vom Vorzeichen.

ULLONG_MAX
bei Überlauf. `errno` wird auf ERANGE gesetzt.

Siehe auch atol, atoll, atoi, strtol, strtoll, stroul, wcstol, wcstoll, wcstoul, wcstoull

strupper - Zeichenkette kopieren mit Umwandlung in Großbuchstaben

Definition #include <string.h>

char *strupper(char *s1, const char *s2);

`strupper` kopiert die Zeichenkette *s2* einschließlich des Nullbytes (\0) in die Zeichenkette *s1* und wandelt dabei die Kleinbuchstaben in Großbuchstaben um.
s1 muss sie lang genug sein, um die Zeichenkette *s2* einschließlich des Nullbytes (\0) aufnehmen zu können.

Wird die Zeichenkette *s2* als NULL-Zeiger übergeben, entfällt der Kopiervorgang und es werden in *s1* die Kleinbuchstaben in Großbuchstaben umgewandelt.

Returnwert Zeiger auf die Ergebniszeichenkette *s1*.

Hinweise Als Argumente werden Zeichenketten erwartet, die mit dem Nullbyte (\0) abgeschlossen sind.

`strupper` überprüft nicht, ob *s1* groß genug für das Ergebnis ist. Ist *s1* kürzer als *s2* (einschließlich des Nullbytes), so wird der Speicherplatz hinter *s1* überschrieben!

Bei sich überlappenden Speicherbereichen ist das Verhalten undefiniert.

Beispiel Folgendes Programm kopiert den Inhalt von *s2* nach *s1* und wandelt dabei die Klein- in Großbuchstaben um.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *s1 = "                ";
    char *s2 = "kleinbuchstaben!";
    printf("Inhalt s2: %s\n", s2);

                /* s2 nach s1 kopieren mit Umwandlung in Großbuchstaben*/
    strupper(s1, s2);
    printf("Nach strupper:\nInhalt s1: %s\n", s1);
    return 0;
}
```

Siehe auch strlower, tolower, toupper

strxfrm - Transformieren einer Zeichenkette

Definition

```
#include <string.h>

size_t strxfrm(char *s1, const char *s2, size_t n);
```

`strxfrm` transformiert die Zeichen der Zeichenkette *s2*, so dass die lexikalische Reihenfolge jedes Zeichens entsprechend der LC_COLLATE-Kategorie der aktuellen Lokalität interpretiert wird. Anschließend werden maximal *n* transformierte Zeichen (inklusive des abschließenden Nullbytes) in die Zeichenkette *s1* kopiert.
Hat *n* den Wert 0, kann die Ergebniszeichenkette *s1* ein NULL-Zeiger sein.

Ein Vergleich von zwei mit `strxfrm` transformierten Zeichenketten mit der Funktion `strcmp` liefert dann das gleiche Ergebnis, wie ein Vergleich mit der Funktion `strcoll`, angewandt auf dieselben originalen Zeichenketten.

Returnwert Länge der transformierten Zeichenkette (exklusive des abschließenden Nullbytes).

Hinweise Als Argument *s2* wird eine Zeichenkette erwartet, die mit dem Nullbyte abgeschlossen ist.

Die Zeichenkette *s2* wird durch `strxfrm` nicht verändert. Die Transformation wird in einem Arbeitsbereich durchgeführt.

Falls der Returnwert größer oder gleich *n* ist, ist der Inhalt der Zeichenkette *s1* unbestimmbar, da kein Nullbyte geschrieben wurde.

Ist in der aktuellen Lokalität einem der Zeichen in der Zeichenkette *s2* der sedezimale Wert 0 zugeordnet, schließt dieses Zeichen als Nullbyte die transformierte Zeichenkette ab (vgl. „Benutzerspezifische Lokalitäten“, S.116).

Bei sich überlappenden Speicherbereichen ist das Verhalten undefiniert.

Das Lokalitätskonzept ist ausführlich im Kapitel „Lokalität“ auf Seite 99 beschrieben.

Beispiel

```
#include <stdio.h>
#include <string.h>
#include <locale.h>

int main(void)
{
  char alpha2[11];
  char num2[11];
  int comp1;
  int comp2;
  int comp3;
  size_t i = 11;
  char *alpha1 = "ABCDEFGHIJ";
  char *num1 = "0123456789";

  setlocale(LC_COLLATE, "ANNE");      /* Aktivieren der benutzerspezifischen
                                         Lokalität, in der die Ziffern einen
                                         niedrigeren Sortierwert haben als
                                         die Buchstaben */

  comp1 = strcoll(alpha1, num1);      /* Vergleich der originären Zeichen- */
  if(comp1 > 0)                       /* ketten mit strcoll */
     printf ("alpha1 groesser num1\n");
  else printf("Fehler\n");

  comp2 = strcmp(alpha1, num1);       /* Vergleich der originären Zeichen- */
  if(comp2 < 0)                       /* ketten mit strcmp */
     printf ("alpha1 kleiner num1\n");
  else printf("Fehler\n");

  strxfrm(num2, num1, i);             /* Transfomieren mit strxfrm */
  strxfrm(alpha2, alpha1, i);

  comp3 = strcmp(alpha2, num2);       /* Vergleich der transformierten */
  if(comp3 > 0)                       /* Ergebniszeichenketten mit strcmp */
     printf ("alpha2 groesser num2\n");
  else printf("Fehler\n");

return 0;
}
```

Siehe auch setlocale, strcoll, strcmp

swprintf - Langzeichen formatiert ausgeben

Definition
```
#include <wchar.h>
int swprintf(wchar_t *s, size_t n, const wchar_t *format [, arglist]);
```

Beschreibung siehe `fwprintf`.

swscanf - formatiert lesen

Definition
```
#include <wchar.h>
int swscanf(const wchar_t *s, const wchar_t *format [, arglist]);
```

Beschreibung siehe `fwscanf`.

system - Systemkommando ausführen

Definition

```
#include <stdlib.h>

int system(const char *cmd);
```

`system` führt das BS2000-Systemkommando aus, das in der Zeichenkette *cmd* steht.

Returnwert

0	das Systemkommando wurde erfolgreich ausgeführt (Returnwert des entsprechenden Systemkommandos: 0).
-1	das Systemkommando wurde nicht erfolgreich ausgeführt (Returnwert des Systemkommandos: Fehlercode ≠ 0).
undefiniert	nach dem Systemkommando wurde nicht in das Programm zurückverzweigt.

Hinweise

Das Systemkommando kann maximal 2048 Zeichen lang sein und braucht nicht mit dem System-Schrägstrich (/) angegeben zu werden.

Nach einigen Kommandos (START-PROG, LOAD-PROG, CALL-PROCEDURE, DO, HELP-SDF) wird nicht in das aufrufende Programm zurückverzweigt. Läßt ein Programm solche vorzeitigen Programmbeendigungen zu, sollte es vor dem `system`-Aufruf die Puffer leeren (`fflush`) bzw. die Dateien schließen.

`system` übergibt die Zeichenkette *cmd* unverändert dem BS2000-Kommandoprozessor MCLP als Eingabe (siehe auch Handbuch „Makroaufrufe an den Ablaufteil“). Es erfolgt keine Umsetzung in Großbuchstaben.

Beispiel

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  char cmd[225];
  int result;
  printf("Bitte Systemkommando eingeben\n");
  gets(cmd);
  result = system(cmd);
  printf("Returnwert: %d\n", result);
  return 0;
}
```

tan - Tangens

Definition #include <math.h>

double tan(double x);

`tan` berechnet im zulässigen Gleitkommaintervall die trigonometrische Funktion Tangens. x gibt den Winkel im Bogenmaß an.

Returnwert `tan(x)` für eine zulässige Gleitkommazahl x.

+/-HUGE_VAL bei Überlauf. Zusätzlich wird `errno` auf ERANGE gesetzt (Resultat zu groß).

Beispiel Folgendes Programm gibt den Tangens einer eingelesenen Zahl aus.

```
#include <math.h>
#include <stdio.h>

int main(void)
{
     double x;
     printf("Geben Sie bitte eine Zahl ein :\n");
     scanf("%lf", &x);
     printf("Der Tangens von %g ist %g \n", x, tan(x));
     return 0;
}
```

Siehe auch sin, cos, tanh, atan

tanh - Tangens hyperbolicus

Definition #include <math.h>

double tanh(double x);

`tanh` berechnet im zulässigen Gleitkommaintervall die Funktion Tangens hyperbolicus von x.

Returnwert `tanh(x)` für eine zulässige Gleitkommazahl x.

Beispiel Folgendes Programm gibt den Tangens hyperbolicus einer eingelesenen Zahl aus.

```
#include <math.h>
#include <stdio.h>

int main(void)
{
     double x;
     printf("Geben Sie bitte eine Zahl ein :\n");
     scanf("%lf", &x);
     printf("Der Tangens hyperbolicus von %g ist %g \n", x, tanh(x));
     return 0;
}
```

Siehe auch sin, cos, tan, atan

tell - Aktuelle Position des Lese-/Schreibzeigers ermitteln

Definition

```
#include <stdio.h>

long tell(int dk);
```

`tell` liefert die aktuelle Position des Lese-/Schreibzeigers für die Datei mit Dateikennzahl *dk*.

`tell` lässt sich auf Binärdateien (PAM, INCORE) und Textdateien (SAM, ISAM) anwenden. SAM-Dateien werden mit elementaren Funktionen stets als Textdateien verarbeitet.

Returnwert

Position in der Datei bei Erfolg, und zwar
bei Binärdateien die Anzahl Bytes, die der Lese-/Schreibzeiger vom Dateianfang entfernt ist,
bei Textdateien die absolute Position des Lese-/Schreibzeigers.

-1 im Fehlerfall. Zusätzlich wird in `errno` eine entsprechende Fehlerinformation abgelegt (z.B. `tell` nicht erlaubt, Block/Satzanzahl zu groß).

Hinweise

Die Aufrufe `tell(dk)` und `lseek(dk, 0L, SEEK_CUR)` sind äquivalent.

`tell` ist nicht anwendbar für Systemdateien (SYSDTA, SYSLST, SYSOUT).

Da die Informationen über die Dateiposition in einem 4 Byte langen Feld abgelegt werden, ergeben sich für die Größe von SAM- und ISAM-Dateien folgende Einschränkungen bei der Bearbeitung mit `tell/lseek`:

1. SAM-Datei

Satzlänge	≤ 2048 Byte
Satzanzahl/Block	≤ 256
Blockanzahl	≤ 2048

2. ISAM-Datei

Satzlänge	≤ 32 KByte
Satzanzahl	≤ 32 K

Beispiel

siehe Beispiel bei `lseek`.

Siehe auch

lseek, lseek64, fseek, fseek64, ftell, ftell64

time, time64 - Aktuelle Zeitangabe

Definition

```
#include <time.h>

time_t time(time_t *sek_zg);
time64_t time64(time64_t *sek_zg);
```

`time` und `time64` liefern die aktuelle Zeit (Ortszeit) als Anzahl der seit dem Stichtag (Epoche) vergangenen Sekunden.

Bei `time` hängt der Stichtag von der Verwendung des TIMESHIFT-Bindeschalters ab (siehe Abschnitt „Zeitfunktionen“ auf Seite 41):
– ohne TIMESHIFT-Bindeschalter (Standard): 1.1.1950 00:00:00.
– mit TIMESHIFT-Bindeschalter: 1.1.1970 00:00:00.
Bei `time64` ist der Stichtag immer der 1.1.1970 00:00:00.

Bei Sommer-/Winterzeitumstellungen springt der Wert um 3600 bzw. -3600 Sekunden.

Ab dem 19.01.2018 03:14:08 (ohne TIMESHIFT-Bindeschalter) bzw. ab dem 19.01.2038 03:14:08 (mit TIMESHIFT-Bindeschalter) gibt `time` die Meldung `CCMO014` aus und beendet das Programm.

`time64` liefert unabhängig von der Verwendung des TIMESHIFT-Bindeschalters bis zum 18.3.4317 02:44:48 korrekte Ergebnisse.

Parameter

time_t *sek_zg
time64_t *sek_zg

Zeiger auf das von `time` gelieferte Ergebnis.

Wird als Argument ein NULL-Zeiger übergeben, ist dieser Parameter ohne Bedeutung.

Wird kein NULL-Zeiger übergeben, wird das Ergebnis von `time` bzw. `time64` zusätzlich in den Bereich eingetragen, auf den *sek_zg* zeigt.

Returnwert Anzahl der seit dem Stichtag vergangenen Sekunden.

Siehe auch ctime, ctime64, difftime, difftime64, ftime, ftime64, mktime, mktime64

_ _TIME_ _ - Ausgabe der Übersetzungsuhrzeit (Makro)

Definition _ _TIME_ _

Dieses Makro generiert die Übersetzungsuhrzeit einer Quelldatei als Zeichenkette in der Form:

"hh:mm:ss\0"

Dabei bedeutet

hh Stunden

mm Minuten

ss Sekunden

Hinweise Das Format der Zeitangabe entspricht der Funktion `asctime`.

Das Makro muss in keiner Include-Datei definiert werden. Sein Name wird vom Compiler erkannt und ersetzt.

Beispiel

```
#include <stdio.h>

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

{
printf("Uebersetzung des Programms %s am %s um %s Uhr\n", argv[0], __DATE__,
__TIME__);
return 0;
}
```

Siehe auch asctime, _ _DATE_ _

tmpfile, tmpfile64 - Temporäre Binärdatei öffnen

Definition

```
#include <stdio.h>

FILE *tmpfile(void);
FILE *tmpfile64(void);
```

`tmpfile` und `tmpfile64` erzeugen einen eindeutigen Dateinamen - analog zur Funktion `tmpnam` - und öffnet unter diesem Namen eine binäre SAM-Datei mit Standardattributen. Die Datei wird dabei im Modus `wb+` (Neuschreiben und Lesen) geöffnet.

Diese Datei wird automatisch gelöscht, wenn das Programm normal beendet oder wenn die Datei geschlossen wird.

Es besteht kein funktioneller Unterschied zwischen `tmpfile` und `tmpfile64`, außer dass `tmpfile64` einen Dateizeiger auf eine temporäre Datei zurückliefert, die > 2 GB sein kann.

Für die Bearbeitung von Dateien > 2 GB verfahren Sie wie folgt:

– Falls das Define `_FILE_OFFSET_BITS 64` (siehe Seite 71) gesetzt ist, rufen Sie `tmpfile` auf. Implizit wird dann `tmpfile64` mit den passenden Parametern verwendet.

– Andernfalls müssen Sie `tmpfile64` aufrufen.

Returnwert

Zeiger auf die zugewiesene FILE-Struktur bei Erfolg.

NULL-Zeiger wenn die Datei nicht geöffnet werden konnte.

Hinweis

Bei abnormalem Programmabbruch mit `abort` bzw. `_exit(-1)` werden die temporären Dateien nicht gelöscht.

Siehe auch

tmpnam, mktemp, abort

tmpnam - Eindeutigen temporären Dateinamen erzeugen

Definition

```
#include <stdio.h>

char *tmpnam(char *s);
```

`tmpnam` erzeugt einen eindeutigen Dateinamen aus der TSN-Nummer der aktuellen Task, einem internen Kennzeichen, der Uhrzeit, dem Datum und einer maximal vierstelligen Nummer. Bei jedem Aufruf von `tmpnam` ändert sich die maximal vierstellige Nummer und nach Ablauf einer Sekunde die Uhrzeit. Dadurch ist sichergestellt, dass der Name immer unterschiedlich zu den Namen bereits existierender Dateien ist.
`tmpnam` kann maximal TMP_MAX mal aufgerufen werden.

Der Dateiname kann anschließend zum Neuanlegen einer beliebigen Datei verwendet werden.

Returnwert

Zeiger auf den erzeugten Namen:
Wenn *s* ein NULL-Zeiger ist, schreibt `tmpnam` das Ergebnis in einen C-internen Speicherbereich, der bei jedem Aufruf überschrieben wird.
Wenn *s* kein NULL-Zeiger ist, schreibt `tmpnam` das Ergebnis in die Ergebniszeichenkette *s*. Für *s* ist ein Speicherplatz für die Aufnahme von mindestens `L_tmpnam` Zeichen bereitzustellen. `L_tmpnam` ist in <stdio.h> definiert.

0 falls `tmpnam` öfter als TMP_MAX mal aufgerufen wurde.

Hinweise

`tmpnam` erzeugt maximal TMP_MAX Namen. TMP_MAX ist in der Include-Datei <stdio.h> definiert.

Dateien, die mit `tmpnam`-erzeugten Namen geöffnet wurden, werden nicht automatisch bei Programm- bzw. Taskende gelöscht. Die Dateien müssen explizit (z.B. mit `remove`) gelöscht werden.

Beispiel

```
#include <stdio.h>

int main(void)
{
 FILE *fp1;
 FILE *fp2;
 char nam1[L_tmpnam];
 char nam2[L_tmpnam];

 tmpnam(nam1);
 printf("Name1: %s\n", nam1); /* Name1: S.C.UNQ.1RCP.00.132112.27091991.0000 */
 fp1 = fopen(nam1, "w+r");

 tmpnam(nam2);
 printf("Name2: %s\n", nam2); /* Name2: S.C.UNQ.1RCP.00.132112.27091991.0001 */
 fp2 = fopen(nam2, "w+r");

 fclose(fp1);
 fclose(fp2);

 remove(nam1);
 remove(nam2);
}
```

Siehe auch tmpfile, tmpfile64, mktemp, remove

toascii - integer-Wert in gültigen EBCDIC-Wert umwandeln

Definition

```
#include <ctype.h>

int toascii(int i);
```

`toascii` setzt die ersten 3 Bytes einer `integer`-Variablen *i* durch Bit-UND-Verknüpfung (i & 0XFF) auf 0 und liefert den Wert des niedrigstwertigen Bytes zurück.

`toascii` ist ein Synonym für `toebcdic`. Auf EBCDIC-Rechnern liefert `toascii` einen gültigen Wert aus dem EBCDIC-Zeichensatz. Ist Portabilität zu ASCII-Rechnern erforderlich, sollte `toascii` verwendet werden.

Returnwert Wert des niedrigstwertigen Bytes der Variablen *i*.

Hinweise `toascii` wandelt keine Werte aus anderen Zeichensätzen (z.B. ASCII auf EBCDIC-Rechnern) um.

Siehe auch toebcdic

toebcdic - integer-Wert in gültigen EBCDIC-Wert umwandeln

Definition
```
#include <ctype.h>
int toebcdic(int i);
```

`toebcdic` liefert einen gültigen Wert aus dem EBCDIC-Zeichensatz.

`toebcdic` setzt die ersten 3 Bytes einer `integer`-Variablen *i* durch Bit-UND-Verknüpfung (i & 0XFF) auf 0 und liefert den Wert des niedrigstwertigen Bytes zurück.

Returnwert Das niedrigstwertige Byte der Variablen *i*.

Hinweise `toebcdic` ist sowohl als Makro als auch als Funktion realisiert (siehe Abschnitt „Funktionen und Makros“ auf Seite 19).

`toebcdic` wandelt keine Werte aus anderen Zeichensätzen (z.B. ASCII) um.

`toebcdic` ist ein Synonym für `toascii`. Ist Portabilität zu ASCII-Rechnern erforderlich, sollten Sie statt `toebcdic` `toascii` verwenden.

Siehe auch toascii

tolower - Großbuchstaben in Kleinbuchstaben umwandeln

Definition
```
#include <ctype.h>
int tolower(int c);
```

`tolower` wandelt den Großbuchstaben *c* (angegeben als EBCDIC-Wert) in den entsprechenden Kleinbuchstaben um.

Returnwert Kleinbuchstabe zu *c*
wenn *c* ein Großbuchstabe ist.

c unverändert wenn *c* kein Großbuchstabe ist.

Hinweis `tolower` ist sowohl als Makro als auch als Funktion realisiert (siehe Abschnitt „Funktionen und Makros“ auf Seite 19).

Beispiel Folgendes Programm liest eine Zeichenkette ein, und wandelt die Zeichen zunächst in Kleinbuchstaben und dann in Großbuchstaben um. Zeichen, die weder Groß- bzw. Kleinbuchstaben sind (Ziffern, Sonderzeichen etc.), bleiben unverändert.

```
#include <ctype.h>
#include <stdio.h>

int main(void)
{
  int i;
  char s[81];

  printf("Bitte geben Sie eine Zeichenkette (max. 80 Zeichen) ein\n");
  scanf("%s", s);

  printf("Und jetzt alles in Kleinbuchstaben \n");
  for (i=0; s[i] != '\0'; ++i)
       if (isupper(s[i]))
           printf("%c", tolower(s[i]));
       else printf("%c", s[i]);

  printf("\n Und in Grossbuchstaben \n");
  for (i=0; s[i] != '\0'; ++i)
       if (islower(s[i]))
           printf("%c", toupper(s[i]));
       else printf("%c", s[i]);


  printf("\n");

  return 0;
}
```

Siehe auch strlower, strupper, toupper, toascii, toebcdic, towlower

toupper - Kleinbuchstaben in Großbuchstaben umwandeln

Definition
```
#include <ctype.h>
int toupper(int c);
```

`toupper` wandelt den Kleinbuchstaben c in den entsprechenden Großbuchstaben um.

Returnwert
Großbuchstabe zu c wenn c ein Kleinbuchstabe ist.

c unverändert wenn c kein Kleinbuchstabe ist.

Hinweis `toupper` ist sowohl als Makro als auch als Funktion realisiert (siehe Abschnitt „Funktionen und Makros“ auf Seite 19).

Beispiel siehe Beispiel bei `tolower`

Siehe auch strupper, strlower, tolower, toascii, toebcdic, towupper

towctrans - Langzeichen abbilden

Definition #include <wctype.h>

wint_t towctrans(wint_t wc, wctrans_t desc);

`towctrans` transformiert das Langzeichen *wc* gemäß der Angabe *desc*. Der aktuelle Wert der Kategorie LC_CTYPE muss derselbe sein, der für den Aufruf von `towctrans` gültig war, der den Wert *desc* zurückgab.

Die beiden folgenden Aufrufe von `towctrans` wirken genauso, wie die dahinter in Kommentarzeichen angegebenen Aufrufe zur Umwandlung in Klein- bzw. Großbuchstaben:

```
towctrans(wc, wctrans("tolower"))          /* towlower(wc) */
towctrans(wc, wctrans("toupper"))          /* towupper(wc) */
```

Returnwert transformiertes Langzeichen
bei Erfolg.

Hinweis In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Siehe auch tolower, toupper, towlower, towupper, wctrans

towlower - Langzeichen in Kleinbuchstaben umwandeln

Definition #include <wctype.h>

wint_t towlower(wint_t wc);

`towlower` wandelt das Langzeichen *wc,* falls es ein Großbuchstabe ist, in den entsprechenden Kleinbuchstaben um.

Returnwert Kleinbuchstabe zu *wc*
wenn *wc* ein Großbuchstabe ist.

wc unverändert, falls *wc* kein Großbuchstabe ist.

Hinweis In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Siehe auch setlocale, tolower, towupper

towupper - Langzeichen in Großbuchstaben umwandeln

Definition #include <wctype.h>

wint_t towupper(wint_t wc);

`towupper` wandelt das Langzeichen *wc,* falls es ein Kleinbuchstabe ist, in den entsprechenden Großbuchstaben um.

Returnwert Großbuchstabe zu *wc*
wenn *wc* ein Kleinbuchstabe ist.

wc unverändert, falls *wc* kein Kleinbuchstabe ist.

Hinweis In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Siehe auch setlocale, toupper, towlower

ungetc - Zeichen in den Puffer zurückstellen

Definition

```
#include <stdio.h>

int ungetc(int c, FILE *dz);
```

`ungetc` schreibt das Zeichen *c* in den Puffer zurück, der der Datei mit Dateizeiger *dz* zugeordnet ist. Die nächste Leseoperation, die zeichenweise von dieser Datei einliest (`getc`), liefert dann nochmals *c*.
Ist *c* gleich EOF, macht `ungetc` nichts und liefert EOF zurück.

Returnwert

Das zurückgestellte Zeichen *c*	bei Erfolg.
EOF	wenn `ungetc` das Zeichen nicht zurückschreiben kann (Fehler oder *c* ist EOF).

Hinweise

Es muss immer wenigstens ein Zeichen vor dem ersten `ungetc`-Aufruf aus der Datei gelesen worden sein.

EOF kann nicht zurückgestellt werden.

Nach einem erfolgreichen `ungetc`-Aufruf ist der Lese-/Schreibzeiger um ein Zeichen rückwärts verschoben.

Der Aufruf einer der folgenden Funktionen hebt die Effekte des `ungetc`-Aufrufs (z.B. Rückwärtspositionierung) auf: `fseek`/`fseek64`, `fsetpos`/`fsetpos64`, `lseek`/`lseek64`, `rewind`, `fflush`.

Wenn an Stelle des zuvor eingelesenen Zeichens ein anderes Zeichen in den Puffer zurückgestellt wurde, ist das Verhalten je nach KR- oder ANSI-Funktionalität unterschiedlich.
Bei KR-Funktionalität (nur bei C/C++ Versionen kleiner V3.0 vorhanden) werden beim Schreiben des Pufferinhalts in die externe Datei Originaldaten verändert.
Bei ANSI-Funktionalität werden beim Schreiben des Pufferinhalts in die externe Datei Originaldaten nicht verändert, d.h. es werden stets die Originaldaten vor dem `ungetc`-Aufruf in die externe Datei geschrieben.

Siehe auch

getc, ungetwc

ungetwc - Langzeichen in Eingabestrom zurückstellen

Definition
```
#include <stdio.h>
#include <wchar.h>

wint_t ungetwc(wint_t c, FILE *dz);
```

`ungetwc` stellt das Langzeichen *c* in den Puffer zurück, der der Datei mit Dateizeiger *dz* zugeordnet ist. Die nächste Leseoperation, die zeichenweise von dieser Datei einliest (`getwc`), liefert dann nochmals *c*.

Mindestens ein Langzeichen kann zurückgestellt werden. Das gilt auch dann, wenn `ungetwc` direkt einem Aufruf zum formatierten Einlesen von Langzeichen folgt (`fwscanf` oder `wscanf`). Wird `ungetwc` für dieselbe Datei ohne zwischenzeitliche Lesezugriffe oder Positionierungsaufrufe zu oft hintereinander aufgerufen, kann die Funktion fehlschlagen.

Hat *c* den Wert des Makros WEOF, schlägt die Funktion fehl und liefert WEOF zurück.

Returnwert

zurückgestelltes Langzeichen	bei Erfolg.
WEOF	wenn die Funktion fehlschlägt.

Hinweise

In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

WEOF kann nicht zurückgestellt werden.

Nach einem erfolgreichen `ungetwc`-Aufruf ist der Lese-/Schreibzeiger um ein Zeichen rückwärts verschoben.

Der Aufruf einer der folgenden Funktionen hebt die Effekte des `ungetwc`-Aufrufs (z.B. Rückwärtspositionierung) auf: `fseek`/`fseek64`, `fsetpos`/`fsetpos64`, `lseek`/`lseek64`, `rewind`, `fflush`.

Wenn an Stelle des zuvor eingelesenen Zeichens ein anderes Zeichen in den Puffer zurückgestellt wurde, werden beim Schreiben des Pufferinhalts in die externe Datei Originaldaten nicht verändert, d.h. es werden stets die Originaldaten vor dem `ungetc`-Aufruf in die externe Datei geschrieben.

Siehe auch getc, getwc, ungetc, ungetwc

unlink - Datei löschen

Definition

```
#include <stdio.h>

int unlink(const char *d_name);
```

`unlink` wird aus Kompatibilitätsgründen weiter unterstützt und arbeitet wie die ANSI-Funktion `remove` (Beschreibung siehe dort).

Siehe auch remove

va_arg - Variable Argumentenliste abarbeiten

Definition

```
#include <stdarg.h>

<typ> va_arg(va_list arg_zg, <typ>);
```

Das Makro `va_arg` dient, zusammen mit den Makros `va_start` und `va_end`, zur Bearbeitung einer Liste von Argumenten, deren Anzahl und Typ bei jedem Funktionsaufruf variieren kann. Eine variable Argumentenliste wird in der Formalparameterliste der Funktionsdefinition mit ", ..." gekennzeichnet.

Das Makro `va_arg` liefert den Datentyp und Wert des jeweils nächsten Arguments einer variablen Argumentenliste, beginnend mit dem ersten Argument. Technisch gesehen, expandiert das Makro zu einem Ausdruck vom Datentyp und Wert des Arguments.

Vor dem ersten Aufruf von `va_arg` muss die variable Argumentenliste, auf die *arg_zg* zeigt, mit `va_start` initialisiert worden sein. Jeder Aufruf von `va_arg` verändert *arg_zg* so, dass der Wert des jeweils nächsten Arguments zur Verfügung steht.

Parameter

va_list arg_zg
: Zeiger auf die Argumentenliste, die vor dem ersten Aufruf von `va_arg` mit `va_start` initialisiert wurde.

<typ>
: Typname, der zum Typ des aktuellen Argumentes passt. Es sind alle C-Datentypen zulässig, für die gilt: Ein Zeiger auf ein Objekt vom Typ *typ* ist durch ein einfaches Anhängen von * an *typ* definiert. Unzulässig sind z.B. Array- und Funktionstypen.

Returnwert

Wert des Arguments
: Der erste Aufruf nach dem Aufruf von `va_start` liefert den Wert des ersten Arguments. Dieses Argument liegt hinter dem letzten „benannten“ Argument *parmN* in der Formalparameterliste (vgl. `va_start`).
Darauf folgende Aufrufe liefern sukzessive die restlichen Argumentenwerte.

undefiniert
: Falls es kein nächstes Argument gibt oder <typ> nicht zum aktuellen Argument passt.

Hinweise

Die Kompatibilität von Argumenttypen wird vom C-Laufzeitsystem dahingehend unterstützt, dass ähnliche Typen in der selben Weise in der Parameterliste abgelegt werden, und zwar:
Alle `unsigned`-Typen (inklusive `char`) werden wie `unsigned int` dargestellt (rechtsbündig in einem Wort).
Alle anderen ganzzahligen Typen werden wie `int` dargestellt (rechtsbündig in einem Wort).

`float` wird wie `double` dargestellt (rechtsbündig in einem Doppelwort).
Vor der Rückkehr aus einer Funktion, deren Argumentenliste mit `va_arg` abgearbeitet wurde, muss das Makro `va_end` aufgerufen werden.

Beispiel Die Funktion *f1* füllt ein Array mit einer Liste von Argumenten, die vom Typ Zeiger auf Zeichenkette sind. Dabei sollen nicht mehr als MAXARGS Argumente verarbeitet werden. Die Anzahl der Zeigerargumente wird als erstes Argument für *f1* definiert. Das aufgefüllte Array wird anschließend einer Funktion *f2* übergeben.

```
#include <stdarg.h>
#include <stdio.h>
#define MAXARGS 20

extern int f2(int i, char *a[]);

void f1(int n_ptrs, ...)
{
   va_list ap;
   char *array[MAXARGS];
   int ptr_no = 0;

   if (n_ptrs > MAXARGS)
      n_ptrs = MAXARGS;
   va_start(ap, n_ptrs);
   while (ptr_no < n_ptrs)
      array[ptr_no++] = va_arg(ap, char *);
   va_end(ap);
   f2(n_ptrs, array);
   return 0;
}
```

Siehe auch va_start, va_end

va_end - Variable Argumentenliste abschließen

Definition

```
#include <stdarg.h>

void va_end(va_list arg_zg);
```

Das Makro `va_end` dient, zusammen mit den Makros `va_start` und `va_arg`, zur Bearbeitung einer Liste von Argumenten, deren Anzahl und Typ bei jedem Funktionsaufruf variieren kann. Eine variable Argumentenliste wird in der Formalparameterliste der Funktionsdefinition mit ", ..." gekennzeichnet.

`va_end` führt Abschlussarbeiten an der variablen Argumentenliste *arg_zg* durch. Das Makro muss vor der Rückkehr aus einer Funktion aufgerufen werden, deren Argumentenliste mit `va_start` und `va_arg` abgearbeitet wurde.

`va_end` kann die Argumentenliste *arg_zg* verändern, so dass sie nicht mehr verwenbar ist. Für eine weiter Verwendung ist daher die Argumentenliste mit `va_start` neu zu initialisieren.

Beispiel siehe bei `va_arg`

Siehe auch va_arg, va_start

va_start - Variable Argumentenliste initialisieren

Definition

```
#include <stdarg.h>

void va_start(va_list arg_zg, parmN);
```

Das Makro `va_start` dient, zusammen mit den Makros `va_arg` und `va_end`, zur Bearbeitung einer Liste von Argumenten, deren Anzahl und Typ bei jedem Funktionsaufruf variieren kann. Eine variable Argumentenliste wird in der Formalparameterliste der Funktionsdefinition mit ", ..." gekennzeichnet.

`va_start` muss vor dem ersten Zugriff auf ein „namenloses“ Argument aufgerufen werden. Das Makro initialisiert die variable Argumentenliste *arg_zg* für nachfolgende `va_arg`- und `va_end`-Aufrufe.

Parameter

va_list arg_zg
: Argumentenliste.

parmN
: Name des letzten „benannten“ Parameters in der Formalparameterliste der Funktionsdefinition. Das ist der Parameter, auf den ", ..." folgt. Funktionen, die variable Argumentenlisten verarbeiten, müssen mindestens einen benannten Parameter definieren.

  parmN darf nicht vom Typ Register, Funktion oder Array sein.

Hinweise

Das Verhalten ist undefiniert, wenn *parmN* einen unzulässigen Datentyp hat oder wenn der Datentyp nicht zum aktuellen Argument passt.

Die Kompatibilität von Argumenttypen wird vom C-Laufzeitsystem dahingehend unterstützt, dass ähnliche Typen in der selben Weise in der Parameterliste abgelegt werden, und zwar:
Alle `unsigned`-Typen (inklusive `char`) werden wie `unsigned int` dargestellt (rechtsbündig in einem Wort).
Alle anderen ganzzahligen Typen werden wie `int` dargestellt (rechtsbündig in einem Wort).
`float` wird wie `double` dargestellt (rechtsbündig in einem Doppelwort).

Beispiel

siehe bei `va_arg`

Siehe auch

va_arg, va_end

vfprintf - Formatierte Ausgabe in eine Datei

Definition

```
#include <stdio.h>

int vfprintf(FILE *dz, const char *format, va_list arg);
```

`vfprintf` gleicht der Funktion `fprintf`. Im Unterschied zu `fprintf` erlaubt `vfprintf` die Ausgabe von Argumenten, deren Anzahl und Datentyp zum Übersetzungszeitpunkt nicht bekannt sind.
`vfprintf` wird innerhalb von Funktionen benutzt, an die der Aufrufer jeweils eine andere Formatzeichenkette sowie andere auszugebende Argumente übergeben kann. Die Formalparameterliste der Funktionsdefinition sieht dafür eine Formatzeichenkette *format* und eine variable Argumentenliste ", ..." vor.
format ist eine Formatzeichenkette wie bei `printf` mit ANSI-Funktionalität beschrieben (siehe dort).

`vfprintf` arbeitet eine Argumentenliste *arg* mit internen `va_arg`-Aufrufen sukzessive ab und schreibt die Argumente gemäß der Formatzeichenkette *format* in die Datei mit Dateizeiger *dz*. Die variable Argumentenliste *arg* muss vor dem Aufruf von `vfprintf` mit dem Makro `va_start` initialisiert worden sein.

Returnwert

Anzahl der ausgegebenen Zeichen	bei Erfolg.
Integer< 0	bei Fehler.

Hinweise `vfprint` beginnt in der variablen Argumentenliste immer mit dem ersten Argument. Die Ausgabe ab einem beliebigen Argument lässt sich mit entsprechend vielen `va_arg`-Aufrufen vor Aufruf der Funktion `vfprintf` erreichen. Jeder `va_arg`-Aufruf positioniert die Argumentenliste um ein Argument weiter.

`vfprintf` ruft nicht das Makro `va_end` auf. Da `vfprintf` das Makro `va_arg` benutzt, ist der Wert von *arg* nach der Rückkehr unbestimmt.

Beispiel In folgendem Programmausschnitt gibt die Funktion `vfprintf` pro Aufruf der Fehlerroutine *error* unterschiedliche Arten von Informationen aus.

```
#include <stdarg.h>
#include <stdio.h>

void error(char *f, ...);
int main(void)
{
  .
  .
  char *weight = "WARNING";
  int num = 20;
  error("Fehlerklasse: %s, Anzahl: %d\n", weight, num);
  .
  .
  error("Kein Fehler\n");
  .
  .
}

void error(char *format, ...)
{
  va_list arg;
  va_start(arg, format);
  vfprintf(stderr, format, arg);
  va_end (arg);
}
```

Siehe auch vprintf, vsprintf, vsnprintf

vfwprintf - Langzeichen formatiert ausgeben

Definition
```
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>

int vfwprintf(FILE *dz, const wchar_t *format, va_list arg);
```

Beschreibung siehe `fwprintf`.

vprintf - Formatierte Ausgabe auf Standardausgabe

Definition #include <stdio.h>

int vprintf(const char *format, va_list arg);

`vprintf` gleicht der Funktion `printf`. Im Unterschied zu `printf` erlaubt `vprintf` die Ausgabe von Argumenten, deren Anzahl und Datentyp zum Übersetzungszeitpunkt nicht bekannt sind.
`vprintf` wird innerhalb von Funktionen benutzt, an die der Aufrufer jeweils eine andere Formatzeichenkette sowie andere auszugebende Argumente übergeben kann. Die Formalparameterliste der Funktionsdefinition sieht dafür eine Formatzeichenkette *format* und eine variable Argumentenliste ", ..." vor.
format ist eine Formatzeichenkette wie bei `printf` mit ANSI-Funktionalität beschrieben (siehe dort).

`vprintf` arbeitet eine Argumentenliste *arg* mit internen `va_arg`-Aufrufen sukzessive ab und schreibt die Argumente gemäß der Formatzeichenkette *format* auf die Standardausgabe stdout. Die variable Argumentenliste *arg* muss vor dem Aufruf von `vprintf` mit dem Makro `va_start` initialisiert worden sein.

Returnwert Anzahl der ausgegebenen Zeichen
bei Erfolg.

Integer< 0) bei Fehler.

Hinweise `vprintf` beginnt in der variablen Argumentenliste immer mit dem ersten Argument. Die Ausgabe ab einem beliebigen Argument lässt sich mit entsprechend vielen `va_arg`-Aufrufen vor Aufruf der Funktion `vprintf` erreichen. Jeder `va_arg`-Aufruf positioniert die Argumentenliste um ein Argument weiter.

`vprintf` ruft nicht das Makro `va_end` auf. Da vprintf das Makro `va_arg` benutzt, ist der Wert von *arg* nach der Rückkehr unbestimmt.

Beispiel siehe bei `vfprintf`

Siehe auch vfprintf, vsprintf, vsnprintf

vsnprintf - Formatierte Ausgabe in eine Zeichenkette

Definition

```
#include <stdarg.h>
#include <stdio.h>

int vnsprintf(char *s, size_t n, const char *format, va_list arg);
```

`vsnprintf` gleicht der Funktion `snprintf`. Im Unterschied zu `snprintf` erlaubt `vsnprintf` die Ausgabe von Argumenten, deren Anzahl und Datentyp zum Übersetzungszeitpunkt nicht bekannt sind.
`vsnprintf` wird innerhalb von Funktionen benutzt, an die der Aufrufer jeweils eine andere Formatzeichenkette sowie andere auszugebende Argumente übergeben kann. Die Formalparameterliste der Funktionsdefinition sieht dafür eine Formatzeichenkette *format* und eine variable Argumentenliste ", ..." vor.

`vsnprintf` arbeitet eine Argumentenliste *arg* mit internen va_arg-Aufrufen sukzessive ab und schreibt die Argumente gemäß der Formatzeichenkette *format* in die Zeichenkette *s*. Die variable Argumentenliste *arg* muss vor dem Aufruf von `vsnprintf` mit dem Makro `va_start` initialisiert worden sein.

`vsnprintf` bricht die Ausgabe beim Erreichen der mit dem Parameter *n* spezifizierten Länge ab, wodurch ein Pufferüberlauf verhindert werden kann.

Parameter

char *s
: Zeiger auf die Ergebniszeichenkette. `vsnprintf` schließt die Zeichenkette mit dem Nullbyte (\0) ab. Die maximale Länge der Ausgabe beträgt daher *n*-1.

size_t n
: Länge des für die Ergebniszeichenkette reservierten Bereichs. *n* darf nicht größer sein als `INT_MAX`. Bei *n* = 0 erfolgt keine Ausgabe.

const char *format
: Formatzeichenkette wie bei `printf` mit KR- oder ANSI-Funktionalität (Beschreibung siehe dort).

  Es gibt nur bzgl. der Steuerzeichen für Zwischenraum (\n, \t, etc.) folgenden Unterschied: `vsnprintf` trägt in die Ergebniszeichenkette den EBCDIC-Wert des Steuerzeichens ein. Erst bei der Ausgabe in Textdateien werden die Steuerzeichen je nach Art der Textdatei in ihre entsprechende Wirkung umgesetzt (siehe Abschnitt „Zwischenraum“ auf Seite 68).

va_list arg
: Zeiger auf die variable Argumentenliste, die mit `va_start` initialisiert wurde.

Returnwert

< 0	*n* > INT_MAX oder Ausgabefehler.
= 0 .. n-1	Die Ausgabe konnte vollständig aufbereitet werden. Der Returnwert gibt die Länge der Ausgabe ohne das abschließende NULL-Zeichen an.
> n	Die Ausgabe konnte nicht vollständig aufbereitet werden. Der Returnwert gibt die Länge ohne das abschließende NULL-Zeichen an, die eine vollständige Ausgabe benötigen würde.

Hinweise

vsnprintf beginnt in der variablen Argumentenliste immer mit dem ersten Argument. Die Ausgabe ab einem beliebigen Argument lässt sich mit entsprechend vielen va_arg-Aufrufen vor Aufruf der Funktion vsnprintf erreichen. Jeder va_arg-Aufruf positioniert die Argumentenliste um ein Argument weiter.

vsnprintf ruft nicht das Makro va_end auf. Da vsnprintf das Makro va_arg benutzt, ist der Wert von *arg* nach der Rückkehr unbestimmt.

Bei sich überlappenden Speicherbereichen ist das Verhalten undefiniert.

Siehe auch vfprintf, vprintf , vsprintf

vsprintf - Formatierte Ausgabe in eine Zeichenkette

Definition

```
#include <stdarg.h>
#include <stdio.h>

int vsprintf(char *s, const char *format, va_list arg);
```

`vsprintf` gleicht der Funktion `sprintf`. Im Unterschied zu `sprintf` erlaubt `vsprintf` die Ausgabe von Argumenten, deren Anzahl und Datentyp zum Übersetzungszeitpunkt nicht bekannt sind.
`vsprintf` wird innerhalb von Funktionen benutzt, an die der Aufrufer jeweils eine andere Formatzeichenkette sowie andere auszugebende Argumente übergeben kann. Die Formalparameterliste der Funktionsdefinition sieht dafür eine Formatzeichenkette *format* und eine variable Argumentenliste ", ..." vor.

`vsprintf` arbeitet eine Argumentenliste *arg* mit internen va_arg-Aufrufen sukzessive ab und schreibt die Argumente gemäß der Formatzeichenkette *format* in die Zeichenkette *s*. Die variable Argumentenliste *arg* muss vor dem Aufruf von `vsprintf` mit dem Makro `va_start` initialisiert worden sein.

Parameter

char *s
: Zeiger auf die Ergebniszeichenkette. `vsprintf` schließt die Zeichenkette mit dem Nullbyte (\0) ab.

const char *format
: Formatzeichenkette wie bei `printf` mit ANSI-Funktionalität (Beschreibung siehe dort).

  Es gibt nur bzgl. der Steuerzeichen für Zwischenraum (\n, \t, etc.) folgenden Unterschied: `vsprintf` trägt in die Ergebniszeichenkette den Wert des Steuerzeichens ein. Erst bei der Ausgabe in Textdateien werden die Steuerzeichen je nach Art der Textdatei in ihre entsprechende Wirkung umgesetzt (siehe Abschnitt „Zwischenraum“ auf Seite 68).

va_list arg
: Zeiger auf die variable Argumentenliste, die mit `va_start` initialisiert wurde.

Returnwert

Anzahl der in *s* gespeicherten Zeichen. Das durch `vsprintf` generierte abschließende Nullbyte (\0) wird dabei nicht mitgezählt.

Hinweise `vsprintf` beginnt in der variablen Argumentenliste immer mit dem ersten Argument. Die Ausgabe ab einem beliebigen Argument lässt sich mit entsprechend vielen `va_arg`-Aufrufen vor Aufruf der Funktion vsprintf erreichen. Jeder `va_arg`-Aufruf positioniert die Argumentenliste um ein Argument weiter.

`vsprintf` ruft nicht das Makro `va_end` auf. Da `vsprintf` das Makro `va_arg` benutzt, ist der Wert von *arg* nach der Rückkehr unbestimmt.

Bei sich überlappenden Speicherbereichen ist das Verhalten undefiniert.

Beispiel siehe bei `vfprintf`

Siehe auch vfprintf, vprintf , vsnprintf

vswprintf - Langzeichen formatiert ausgeben

Definition
```
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>

int vswprintf(wchar_t *s, size_t n, const wchar_t *format, va_list arg);
```

Beschreibung siehe `fwprintf`.

vwprintf - Langzeichen formatiert ausgeben

Definition
```
#include <stdarg.h>
#include <wchar.h>

int vwprintf(const wchar_t *format, va_list arg);
```

Beschreibung siehe `fwprintf`.

wcrtomb - Langzeichen in Multibyte-Zeichen konvertieren

Definition

```
#include <wchar.h>

size_t wcrtomb(char *s, wchar_t wc, mbstate_t *ps);
```

Wenn *s* ein NULL-Zeiger ist, entspricht `wcrtomb` dem Aufruf
`wcrtomb(buf, L'\0', ps)`
wobei *buf* einen internen Puffer bezeichnet.

Wenn *s* kein NULL-Zeiger ist, bestimmt `wcrtomb` die Anzahl der Bytes, die unter Berücksichtigung eventueller Umschalt-Sequenzen zur Darstellung des *wc* entsprechenden Multibyte-Zeichens benötigt werden. Die Ergebnisbytes werden in das Feld geschrieben, auf dessen erstes Element *s* zeigt. Es werden maximal {`MB_CUR_MAX`} Bytes geschrieben. Ist *wc* ein Nullzeichen, wird ein Nullbyte geschrieben, dem eine Umschalt-Sequenz vorausgehen kann, die den „initial shift“-Zustand wiederherstellt.

Der Ergebniszustand entspricht dem „initial conversion“ Zustand.

Returnwert `(size_t)-1` wenn *wc* kein gültiges Langzeichen darstellt. In `errno` wird der Wert des Makros `EILSEQ` geschrieben. Der Konversions-Zustand ist undefiniert.

Anzahl der in das Feld **s* geschriebenen Bytes
sonst.

Hinweis In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Siehe auch mblen, mbtowc, wcstombs, wctomb

wcscat - zwei Langzeichenketten zusammenfügen

Definition

```
#include <wchar.h>

wchar_t *wcscat(wchar_t *ws1, const wchar_t *ws2);
```

`wcscat` hängt eine Kopie der Langzeichenkette *ws2* an das Ende der Langzeichenkette *ws1* an und liefert einen Zeiger auf *ws1* zurück.

Das Null-Langzeichen (`\0`) am Ende der Langzeichenkette *ws1* wird vom ersten Zeichen der Langzeichenkette *ws2* überschrieben.
`wcscat` schließt die Langzeichenkette mit dem Null-Langzeichen (`\0`) ab.

Returnwert Zeiger auf die Ergebnis-Langzeichenkette *ws1*.

Hinweise In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Als Argumente werden Langzeichenketten erwartet, die mit dem Null-Langzeichen (`\0`) abgeschlossen sind.
`wcscat` überprüft nicht, ob *ws1* groß genug für das Ergebnis ist.

Bei sich überlappenden Speicherbereichen ist das Verhalten undefiniert.

Siehe auch strcat, wcsncat

wcschr - Langzeichenkette nach Langzeichen durchsuchen

Definition #include <wchar.h>

wchar_t *wcschr(const wchar_t *ws, wchar_t wc);

`wcschr` sucht das erste Vorkommen des Zeichens *wc* in der Langzeichenkette *ws* und liefert bei Erfolg einen Zeiger auf die gesuchte Position in *ws*. Der Wert von *wc* muss einem Zeichen des Typs `wchar_t` entsprechen und muss ein Langzeichen sein, das einem gültigen Zeichen in der aktuellen Lokalität entspricht.

Das abschließende Null-Langzeichen (`\0`) wird als Zeichen mitberücksichtigt.

Returnwert

Zeiger	auf die Position von *wc* in der Langzeichenkette *ws*.
NULL-Zeiger	wenn *wc* in der Langzeichenkette *ws* nicht enthalten ist.

Hinweise In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Für C++ gelten die beiden folgenden Prototypen für die Funktion `wcschr`:
const wchar_t* wcschr(const wchar_t *ws, wchar_t wc);
wchar_t* wcschr(wchar_t *ws, wchar_t wc);

Siehe auch strchr, wcsrchr

wcscmp - zwei Langzeichenketten vergleichen

Definition #include <wchar.h>

int wcscmp(const wchar_t *ws1, const wchar_t *ws2);

`wcscmp` vergleicht zwei Langzeichenketten *ws1* und *ws2* lexikalisch.

Returnwert

< 0	*ws1* ist lexikalisch kleiner als *ws2.*
= 0	*ws1* und *ws2* sind lexikalisch gleich groß.
> 0	*ws1* ist lexikalisch größer als *ws2.*

Hinweise In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Als Argumente werden Langzeichenketten erwartet, die mit dem Null-Langzeichen (`\0`) abgeschlossen sind.

Siehe auch strncmp, wcsncmp

wcscoll - zwei Langzeichenketten gemäß LC_COLLATE vergleichen

Definition

```
#include <wchar.h>

int wcscoll(const wchar_t *ws1, const wchar_t *ws2);
```

`wcscoll` vergleicht zwei Langzeichenketten *ws1* und *ws2* lexikalisch unter Berücksichtigung der in LC_COLLATE für die Lokalität festgelegten Sortierreihenfolge.

Returnwert

< 0	*ws1* ist bezüglich der festgelegten Sortierreihenfolge kleiner als *ws2*.
= 0	*ws1* und *ws2* sind bezüglich der festgelegten Sortierreihenfolge gleich groß.
> 0	*ws1* ist bezüglich der festgelegten Sortierreihenfolge größer als *ws2*.

Wenn eine der beiden Langzeichenketten lässt sich nicht in eine Multibyte-Zeichenkette umwandeln lässt, schlägt `wcscoll` fehl und `errno` wird auf EINVAL gesetzt.

Hinweise

In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Da es im Standard keinen festlegten Wert für den Fehlerfall gibt, wird empfohlen, `errno` auf den Wert 0 zu setzen, dann `wcscoll` aufzurufen und nach dem Aufruf `errno` zu überprüfen. Falls `errno` ungleich 0 ist, kann angenommen werden, dass ein Fehler aufgetreten ist. Zum Sortieren großer Listen sollten die Funktionen `wcsxfrm` und `wcscmp` verwendet werden.

Siehe auch strcoll, wcsncmp, wcsxfrm

wcscpy - Langzeichenkette kopieren

Definition

```
#include <wchar.h>

wchar_t *wcscpy(wchar_t *ws1, const wchar_t *ws2);
```

`wcscpy` kopiert die Langzeichenkette *ws2* einschließlich des Null-Langzeichens (`\0`) in den Speicherbereich, auf den *ws1* zeigt. *ws1* muss groß genug sein, um die Langzeichenkette *ws2* einschließlich des Null-Langzeichens (`\0`) aufnehmen zu können.

Returnwert Zeiger auf die Ergebnis-Langzeichenkette *ws1*.

Hinweise In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Als Argumente werden Langzeichenketten erwartet, die mit dem Null-Langzeichen (`\0`) abgeschlossen sind.
`wcscpy` überprüft nicht, ob *ws1* groß genug für das Ergebnis ist.
Bei sich überlappenden Speicherbereichen ist das Verhalten undefiniert.

Siehe auch strcpy, wcsncpy

wcscspn - Länge einer komplementären Langzeichenteilkette ermitteln

Definition

```
#include <wchar.h>

size_t wcscspn(const wchar_t *ws1, const wchar_t *ws2);
```

`wcscspn` berechnet ab Beginn der Langzeichenkette *ws1* die Länge des Segmentes, das kein einziges Zeichen aus der Langzeichenkette *ws2* enthält. Das abschließende Null-Langzeichen (`\0`) gilt nicht als Teil der Langzeichenkette *ws2*.
Sobald ein Zeichen in *ws1* mit einem Zeichen in *ws2* übereinstimmt, wird die Funktion beendet und die Segmentlänge zurückgeliefert.
Wenn bereits das erste Zeichen in *ws1* mit einem Zeichen in *ws2* übereinstimmt, ist die Segmentlänge gleich 0.

Returnwert Ganzzahliger Wert
der die Segmentlänge (Anzahl ungleicher Zeichen) ab Beginn der Langzeichenkette *ws1* angibt.

Hinweis In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Siehe auch strcspn, wcsspn

wcsftime - Datum und Uhrzeit in Langzeichenkette umwandeln

Definition

```
#include <wchar.h>

size_t wcsftime(wchar_t *wcs, size_t maxsize, const wchar_t *format,
                const struct tm *timptr);
```

`wcsftime` schreibt Langzeichen-Codes gemäß dem in *format* angegebenen String in das Feld, auf das *wss* zeigt.

Die Funktion verhält sich so, als ob eine von `strftime` erzeugte Zeichenkette als Argument an `mbtowcs` übergeben worden wäre und `mbtowcs` das Ergebnis wiederum als Langzeichenkette mit maximal *maxsize* Langzeichen-Codes an `wcsftime` übergibt.

Falls zwischen sich überlappenden Objekten kopiert wird, ist das Ergebnis undefiniert.

Returnwert

Ganzzahl>0	Anzahl der in das Feld geschriebenen Langzeichen-Codes (ohne abschließende Null), wenn die Anzahl der Langzeichen-Codes inklusive der abschließenden Null kleiner oder gleich *maxsize* ist.
0	sonst. In diesem Falle ist der Feldinhalt unbestimmt.

Siehe auch strftime, mbtowcs

wcslen - Länge einer Langzeichenkette ermitteln

Definition

```
#include <wchar.h>

size_t wcslen(const wchar_t *ws);
```

`wcslen` bestimmt die Anzahl der Langzeichen in der Langzeichenkette *ws*, ohne das abschließende Null-Langzeichen (\0).

Returnwert Länge der Langzeichenkette *ws*.
Das abschließende Null-Langzeichen (`\0`) wird nicht mitgezählt.

Hinweise In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Als Argument wird eine Langzeichenkette erwartet, die mit dem Null-Langzeichen (`\0`) abgeschlossen ist.

Siehe auch strlen

wcsncat - zwei Langzeichenteilketten zusammenfügen

Definition

```
#include <wchar.h>

wchar_t *wcsncat(wchar_t *ws1, const wchar_t *ws2, size_t n);
```

`wcsncat` hängt maximal *n* Zeichen der Langzeichenkette *ws2* an das Ende der Langzeichenkette *ws1* an und liefert einen Zeiger auf *ws1* zurück.

Das Null-Langzeichen (`\0`) am Ende der Langzeichenkette *ws1* wird vom ersten Zeichen der Langzeichenkette *ws2* überschrieben.

Wenn die Langzeichenkette *ws2* weniger als *n* Zeichen enthält, werden nur die Zeichen aus *ws2* an *ws1* angehängt. Wenn die Langzeichenkette *ws2* mehr als *n* Zeichen enthält, werden nur die führenden *n* Zeichen von *ws2* an *ws1* angehängt.

`wcsncat` schließt die Langzeichenkette mit dem Null-Langzeichen (`\0`) ab.

Returnwert Zeiger auf die Ergebnis-Langzeichenkette *ws1*.

Hinweise In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Als Argumente werden Langzeichenketten erwartet, die mit dem Null-Langzeichen (`\0`) abgeschlossen sind.

`wcsncat` überprüft nicht, ob *ws1* groß genug für das Ergebnis ist.
Bei sich überlappenden Speicherbereichen ist das Verhalten undefiniert.

Siehe auch strncat, wcscat

wcsncmp - zwei Langzeichenteilketten vergleichen

Definition

```
#include <wchar.h>

int wcsncmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);
```

`wcsncmp` vergleicht die Langzeichenketten *ws1* und *ws2* bis zur maximalen Länge *n* lexikalisch; z.B liefert

Zeichen, die dem Null-Langzeichen folgen, werden nicht in den Vergleich einbezogen.

Returnwert

< 0	*ws1* ist in den ersten *n* Zeichen lexikalisch kleiner als *ws2*.
= 0	*ws1* und *ws2* sind in den ersten *n* Zeichen lexikalisch gleich groß.
> 0	*ws1* ist in den ersten *n* Zeichen lexikalisch größer als *ws2*.

Hinweise

In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Als Argumente werden Langzeichenketten erwartet, die mit dem Null-Langzeichen (`\0`) abgeschlossen sind.

Siehe auch strncmp, wcscmp

wcsncpy - Langzeichenteilkette kopieren

Definition

```
#include <wchar.h>

wchar_t *wcsncpy(wchar_t *ws1, const wchar_t *ws2, size_t n);
```

`wcsncpy` kopiert maximal *n* Zeichen der Langzeichenkette *ws2* in den Speicherbereich, auf den *ws1* zeigt. Zeichen, die dem Null-Langzeichen folgen, werden nicht kopiert.

Wenn die Langzeichenkette *ws2* weniger als *n* Zeichen enthält, wird nur in der Länge von *ws2* (wcslen + 1) kopiert, *ws1* wird dann bis zur Länge *n* mit Null-Langzeichen aufgefüllt.

Wenn die Langzeichenkette *ws2 n* Zeichen (ohne das Null-Langzeichen) oder mehr enthält, ist die Langzeichenkette *ws1* nicht automatisch mit dem Null-Langzeichen abgeschlossen.

Wenn die Langzeichenkette *ws1* mehr als *n* Zeichen enthält und das letzte kopierte Zeichen aus *ws2* ist nicht das Null-Langzeichen, bleiben ggf. restliche Daten in *ws1* erhalten.

`wcsncpy` schließt *ws1* nicht automatisch mit dem Null-Langzeichen ab.

Returnwert Zeiger auf die Ergebnis-Langzeichenkette *ws1*.

Hinweise In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

`wcsncpy` überprüft nicht, ob der Speicherbereich *ws1* groß genug für das Ergebnis ist!

Da `wcsncpy` die Ergebnis-Langzeichenkette nicht automatisch mit dem Null-Langzeichen abschließt, kann es häufig notwendig sein, *ws1* explizit mit einem Null-Langzeichen abzuschließen. Das ist z.B. der Fall, wenn nur ein Teilstück aus *ws2* kopiert wird und auch *ws2* kein Null-Langzeichen enthält.

Bei sich überlappenden Speicherbereichen ist das Verhalten undefiniert.

Siehe auch strncpy, wcscpy

wcspbrk - erstes Vorkommen eines Langzeichens in Langzeichenkette ermitteln

Definition

```
#include <wchar.h>

wchar_t *wcspbrk(const wchar_t *ws1, const wchar_t *ws2);
```

`wcspbrk` sucht das erste Zeichen in der Langzeichenkette *ws1*, das mit irgendeinem Zeichen aus der Langzeichenkette *ws2* übereinstimmt. Das abschließende Null-Langzeichen (`\0`) gilt nicht als Teil der Langzeichenkette *ws2*.

Returnwert

Zeiger auf das erste gefundene Zeichen in *ws1* bei Erfolg.

NULL-Zeiger falls keinerlei Übereinstimmung vorliegt.

Hinweise

In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Als Argumente werden Langzeichenketten erwartet, die mit dem Null-Langzeichen (`\0`) abgeschlossen sind.

Für C++ gelten die beiden folgenden Prototypen für die Funktion `wcspbrk`:

```
const wchar_t* wcspbrk(const wchar_t *ws1, const wchar_t *ws2);
      wchar_t* wcspbrk(       wchar_t *ws1, const wchar_t *ws2);
```

Siehe auch strpbrk, wcschr, wcsrchr

wcsrchr - letztes Vorkommen eines Langzeichens in Langzeichenkette ermitteln

Definition

```
#include <wchar.h>

wchar_t *wcsrchr(const wchar_t *ws, wchar_t wc);
```

`wcsrchr` sucht das letzte Vorkommen des Zeichens *wc* in der Langzeichenkette *ws* und liefert bei Erfolg einen Zeiger auf die gesuchte Position in *ws*.

Das abschließende Null-Langzeichen (`\0`) wird als Zeichen mitberücksichtigt.

Returnwert

Zeiger auf die Position von *wc* in der Langzeichenkette *ws* bei Erfolg.

NULL-Zeiger wenn *wc* in der Langzeichenkette *ws* nicht enthalten ist.

Hinweise

In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Für C++ gelten die beiden folgenden Prototypen für die Funktion `wcsrchr`:

```
const wchar_t* wcsrchr(const wchar_t *ws, wchar_t wc);
      wchar_t* wcsrchr(       wchar_t *ws, wchar_t wc);
```

Siehe auch

strrchr, wcsch

wcsrtombs - Langzeichenkette in Multibyte-Zeichenkette umwandeln

Definition #include <wchar.h>

size_t wcsrtombs(char *dst, const wchar_t **src, size_t len, mbstate_t *ps);

`wcsrtombs` konvertiert eine Folge von Langzeichen aus dem Feld, auf das *src* indirekt zeigt, in Multibyte-Zeichen. `mbsrtowcs` beginnt die Umwandlung mit dem Konvertierungszustand, der in **ps* beschrieben wird. Die konvertierten Zeichen werden in das Feld geschrieben, auf das *dst* zeigt, sofern *dst* kein NULL-Zeiger ist. Jedes einzelne Zeichen wird so konvertiert, als sei die Funktion `wcrtomb` aufgerufen worden.

Die Umwandlung ist beendet, wenn ein abschließendes Nullzeichen auftritt. Das Nullzeichen wird ebenfalls umgewandelt und in das Feld geschrieben.

Die Umwandlung wird vorher abgebrochen, wenn

– eine Bytefolge auftritt, zu der kein gültiges Multibyte-Zeichen korrespondiert oder

– *dst* kein NULL-Zeiger ist und das nächste Multibyte-Zeichen die Gesamtlänge ***len*** der in das Feld zu schreibenden Bytes übersteigen würde.

Wenn *dst* kein NULL-Zeiger ist, wird dem Zeigerobjekt, auf das *src* zeigt, einer der beiden folgenden Werte zugewiesen:

– ein NULL-Zeiger, falls die Umwandlung mit dem Erreichen eines Nullzeichens beendet wurde

– die Adresse direkt hinter dem letzten umgewandelten Langzeichen.

Wenn *dst* kein NULL-Zeiger ist und die Umwandlung mit dem Erreichen eines Nullzeichens beendet wurde, entspricht der Ergebniszustand dem „initial conversion“ Zustand.

Returnwert `(size_t)-1` wenn ein Konvertierungsfehler auftritt, das heißt eine Folge von Bytes, zu der kein gültiges Multibyte-Zeichen korrespondiert. In `errno` wird der Wert des Makros `EILSEQ` geschrieben. Der Konversions-Zustand ist undefiniert.

Anzahl der Bytes in der konvertierten Multibyte-Zeichenkette (ohne abschließendes Nullzeichen) sonst.

Siehe auch mblen, mbtowc, wcstombs, wctomb

wcsspn - Länge einer Langzeichenteilkette ermitteln

Definition

```
#include <wchar.h>

size_t wcsspn(const wchar_t *ws1, const wchar_t *ws2);
```

`wcsspn` berechnet ab Beginn der Langzeichenkette *ws1* die Länge des Segmentes, das ausschließlich Zeichen aus der Langzeichenkette *ws2* enthält.

Sobald ein Zeichen in *ws1* mit keinem Zeichen in *ws2* übereinstimmt, wird die Funktion beendet und die Segmentlänge zurückgeliefert.

Wenn bereits das erste Zeichen in *ws1* mit keinem Zeichen in *ws2* übereinstimmt, ist die Segmentlänge gleich 0.

Returnwert Ganzzahliger Wert
der die Segmentlänge (Anzahl passender Zeichen) ab Beginn der Langzeichenkette *ws1* angibt.

Hinweise In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Als Argumente werden Langzeichenketten erwartet, die mit dem Null-Langzeichen (`\0`) abgeschlossen sind.

Siehe auch strspn, wcscspn

wcsstr - erstes Vorkommen einer Langzeichenkette suchen

Definition

```
#include <wchar.h>

wchar_t *wcsstr( const wchar_t *ws1, const wchar_t *ws2);
```

`wcsstr` sucht das erste Vorkommen der Langzeichenkette *ws2* (ohne das abschließende Nullzeichen) in der Langzeichenkette *ws1*.

Returnwert

Zeiger auf den Beginn der gefundenen Zeichenkette	wenn *ws2* in *ws1* gefunden wird.
NULL-Zeiger	wenn *ws2* in *ws1* nicht gefunden wird.
ws1	wenn *ws2* ein NULL-Zeiger ist.

Hinweis

Für C++ gelten die beiden folgenden Prototypen für die Funktion `wcsstr`:

```
const wchar_t* wcsstr(const wchar_t *ws1, const wchar_t *ws2);
      wchar_t* wcsstr(       wchar_t *ws1, const wchar_t *ws2);
```

Siehe auch strstr, wmemcmp, wmemcpy, wmemchr

wcstod - Langzeichenkette in Gleitkommazahl (double) umwandeln

Definition

```
#include <wchar.h>

double wcstod(const wchar_t *nptr, wchar_t **endptr);
```

`wcstod` wandelt den ersten Teil der Zeichenkette aus Langzeichenwerten, auf die *nptr* zeigt, in eine Darstellung mit doppelter Genauigkeit um. Zuerst wird die Eingabe-Zeichenkette aus Langzeichenwerten in drei Teile zerlegt:

– eine möglicherweise leere Folge von Zwischenraumzeichen als Langzeichenwerte (entsprechend der Angabe durch `iswspace`) am Anfang,

– eine Folge, die als Gleitkommakonstante interpretiert wird,

– und schließlich eine Zeichenkette aus Langzeichenwerten mit einem oder mehr nicht erkannten Langzeichenwerten, einschließlich abschließendem Nullbyte der Eingabe-Zeichenkette aus Langzeichenwerten.

Dann wird versucht, die mittlere Folge in eine Gleitkommazahl umzuwandeln. Anschließend wird das Ergebnis zurückgegeben.

Es wird erwartet, dass diese mittlere Folge folgendes Format hat:
Das Vorzeichen + oder - (optional), eine nichtleere Folge von Ziffern, die optional ein Dezimalzeichen enthalten kann, und schließlich ein optionaler Exponententeil. Ein Exponententeil besteht aus dem Zeichen `e` bzw. `E`, gefolgt von einem Vorzeichen (optional) und einer oder mehreren dezimalen Ziffern. Diese mittlere Folge ist als die längste Teilfolge der Eingabe-Zeichenkette aus Langzeichenwerten definiert. Sie beginnt mit dem ersten Langzeichenwert, der kein Zwischenraumzeichen ist und das erwartete Format aufweist. Diese Folge enthält keine Langzeichenwerte, wenn die Eingabe-Zeichenkette aus Langzeichenwerten leer ist oder nur aus Langzeichenwerten besteht, die Zwischenraumzeichen sind, bzw. wenn der erste Langzeichenwert, der kein Zwischenraumzeichen ist, etwas anderes ist als ein Vorzeichen, eine Ziffer oder ein Dezimalzeichen.

Wenn diese mittlere Folge das erwartete Format aufweist, wird die Folge der Langzeichenwerte, die mit der ersten Ziffer oder dem Dezimalzeichen beginnt (je nachdem, was zuerst steht), als Gleitkommakonstante entsprechend der Definition in der Sprache C interpretiert. Der Unterschied besteht darin, dass das Dezimalzeichen statt des Punktes verwendet wird und, wenn weder ein Exponententeil noch ein Dezimalzeichen erscheint, nach der letzten Ziffer in der Zeichenkette aus Langzeichenwerten ein Dezimalzeichen angenommen wird. Wenn die Folge mit einem Minuszeichen beginnt, ist das Ergebnis der Umwandlung negativ. Ein Zeiger auf die letzte Zeichenkette aus Langzeichenwerten wird in dem Objekt abgelegt, auf das *endptr* zeigt, wenn *endptr* kein NULL-Zeiger ist.

Das Dezimalzeichen ist in der Lokalität des Programms definiert (Kategorie LS_NUMERIC).

Wenn die mittlere Folge leer ist oder nicht das erwartete Format aufweist, wird keine Umwandlung durchgeführt. Der Wert von *nptr* wird in dem Objekt abgelegt, auf das *endptr* zeigt, wenn *endptr* kein NULL-Zeiger ist.

Returnwert konvertierter Wert bei Erfolg.

0	wenn keine Umwandlung durchgeführt werden konnte.
HUGE_VAL	wenn der richtige Wert außerhalb des Bereichs der darstellbaren Werte liegt (entsprechend dem Vorzeichen des Wertes). `errno` wird auf ERANGE gesetzt, um den Fehler anzuzeigen.

Hinweise In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Da 0 sowohl bei einem Fehler zurückgegeben wird als auch bei Erfolg einen gültigen Returnwert darstellt, muss eine Anwendung, die auf Fehler prüfen will, die folgenden Aktionen ausführen: `errno` wird auf 0 gesetzt, `wcstod` aufgerufen und der Wert von `errno` überprüft. Falls dieser Wert ungleich Null ist, wird angenommen, dass ein Fehler aufgetreten ist.

Siehe auch iswspace, localeconv, scanf, setlocale, strtod, wcstol

wcstok - Langzeichenkette in Langzeichenteilkette zerlegen

Definition

```
#include <wchar.h>

wchar_t *wcstok(wchar_t *ws1, const wchar_t *ws2, wchar_t **ptr);
```

Mit `wcstok` lässt sich eine Gesamtlangzeichenkette *ws1* in Langzeichenteilketten - sog. "Tokens" - zerlegen, z.B. ein Satz in die einzelnen Wörter oder eine Quellprogrammanweisung in die kleinsten syntaktischen Einheiten. Der Zeiger auf *ws1* darf nur beim ersten `wcstok`-Aufruf übergeben werden. In *ptr* speichert `wcstok` die Informationen, die zur weiteren Zerlegung der Langzeichenkette notwendig sind.
Ab dem zweiten Aufruf ist für *ws1* ein NULL-Zeiger anzugeben und in *ptr* der Wert, der beim vorhergehenden Aufruf mit der gleichen Langzeichenkette gespeichert wurde.

Beginn- und Endekriterium für jedes Token sind Trennzeichen (Separatoren), die in einer zweiten Langzeichenkette *ws2* anzugeben sind. Token können durch einen oder mehrere dieser Separatoren begrenzt sein, bzw. durch Beginn und Ende der Gesamtlangzeichenkette *ws1*. Typische Separatoren zwischen den Wörtern eines Satzes sind z.B. Leerzeichen, Doppelpunkt, Komma etc.

Pro Aufruf bearbeitet `wcstok` genau eine Langzeichenteilkette. Der erste Aufruf liefert einen Zeiger auf den Beginn der ersten gefundenen Langzeichenteilkette, die weiteren Aufrufe jeweils einen Zeiger auf den Beginn der nächsten Langzeichenteilketten. Jede Langzeichenteilkette schließt `wcstok` mit dem Null-Langzeichen (`\0`) ab.

Bei jedem Aufruf kann eine andere Trennzeichenfolge *ws2* angegeben werden.

Returnwert

Zeiger auf den Beginn einer Langzeichenteilkette.
Beim ersten Aufruf ein Zeiger auf die erste Langzeichenteilkette, beim nächsten Aufruf ein Zeiger auf die nachfolgende Langzeichenteilkette etc. `wcstok` schließt jede Langzeichenteilkette in *ws1* mit einem Null-Langzeichen (`\0`) ab, wobei das jeweils erste gefundene Trennzeichen mit dem Null-Langzeichen (`\0`) überschrieben wird.

NULL-Zeiger falls keine bzw. keine weitere Langzeichenteilkette gefunden wurde.

Hinweis

In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Siehe auch

strtok

wcstol - Langzeichenkette in ganze Zahl (long int) umwandeln

Definition

```
#include <wchar.h>

long int wcstol(const wchar_t *nptr, wchar_t **endptr, int base);
```

`wcstol` wandelt den ersten Teil der Zeichenkette aus Langzeichenwerten, auf die *nptr* zeigt, in die Darstellung `long int` um. Zuerst wird die Eingabe-Zeichenkette aus Langzeichenwerten in drei Teile zerlegt:

– eine möglicherweise leere Folge von Zwischenraumzeichen als Langzeichenwerte (entsprechend der Angabe durch `iswspace`) am Anfang,

– eine Folge, die als ganze Zahl mit einer Dezimalzeichen-Darstellung interpretiert wird, die durch den Wert von *base* bestimmt wird,

– und schließlich eine Zeichenkette aus Langzeichenwerten mit einem oder mehr nicht erkannten Langzeichenwerten, einschließlich abschließendem Nullbyte der Eingabe-Zeichenkette aus Langzeichenwerten.

Dann wird versucht, die mittlere Folge in eine ganze Zahl umzuwandeln. Anschließend wird das Ergebnis zurückgegeben.

Wenn der Wert von *base* gleich Null ist, wird als Format der mittleren Folge eine dezimale Konstante, oktale Konstante oder hexadezimale Konstante erwartet. Dieser kann + bzw. - vorangestellt sein. Eine dezimale Konstante beginnt mit einer Ziffer ungleich Null und besteht aus einer Folge dezimaler Ziffern. Eine oktale Konstante besteht aus dem Präfix 0 und optional einer Folge nur dezimaler Ziffern. Eine hexadezimale Konstante besteht aus dem Präfix 0x bzw. 0X und einer Folge dezimaler Ziffern und der Buchstaben a (bzw. A) bis f (bzw. F) mit den Werten 10 bis 15.

Wenn der Wert von *base* zwischen 2 und 36 liegt, wird als Format der mittleren Folge eine Sequenz von Buchstaben und Ziffern erwartet, die eine ganze Zahl darstellt mit der Basis, die durch *base* bestimmt wird (allerdings keine ganze Zahl mit Suffix). Optional kann das Vorzeichen + bzw. - vorangestellt sein. Den Buchstaben von a (bzw. A) bis einschließlich z (bzw. Z) sind die Werte 10 bis 35 zugeordnet. Es sind nur Buchstaben zulässig, deren Wert kleiner ist als der Wert von *base*. Ist der Wert von *base* gleich 16, können die Darstellungen 0x bzw. 0X für Langzeichenwerte, gegebenenfalls mit Vorzeichen, der Zeichen- und Buchstabenfolge voranstehen.

Diese mittlere Folge ist als die längste beginnende Teilfolge der Eingabe-Zeichenkette aus Langzeichenwerten definiert. Sie beginnt mit dem ersten Langzeichenwert, der kein Zwischenraumzeichen ist und das erwartete Format aufweist. Diese Folge enthält keine Langzeichenwerte, wenn die Eingabe-Zeichenkette aus Langzeichenwerten leer ist oder nur aus Langzeichenwerten besteht, die Zwischenraumzeichen sind, bzw. wenn der erste Langzeichenwert, der kein Zwischenraumzeichen ist, etwas anderes als ein Vorzeichen oder ein zulässiger Buchstabe bzw. eine zulässige Ziffer ist.

Wenn diese mittlere Folge das erwartete Format aufweist und der Wert von *base* gleich Null ist, wird die Folge der Langzeichenwerte, die mit der ersten Ziffer beginnt, als Integer-Konstante interpretiert. Wenn diese mittlere Folge das erwartete Format aufweist und der Wert von *base* zwischen 2 und 36 liegt, wird sie als Grundlage für die Umwandlung verwendet. Jedem Buchstaben wird sein Wert (siehe oben) zugeordnet. Wenn die Folge mit einem Minuszeichen beginnt, ist das Ergebnis der Umwandlung negativ. Wenn *endptr* kein NULL-Zeiger ist, wird ein Zeiger auf die abschließende Zeichenkette aus Langzeichenwerten in dem Objekt abgelegt, auf das *endptr* zeigt.

Wenn diese mittlere Folge leer ist oder nicht das erwartete Format aufweist, wird keine Umwandlung durchgeführt. Der Wert von *nptr* wird in dem Objekt abgelegt, auf das *endptr* zeigt, wenn *endptr* kein NULL-Zeiger ist.

Returnwert

konvertierter Wert
: bei Erfolg.

0
: wenn keine Umwandlung durchgeführt werden konnte. `errno` wird auf EINVAL gesetzt, wenn der Wert von *base* nicht unterstützt wird.

LONG_MAX, LONG_MIN
: abgängig vom Vorzeichen des Wertes.

ULONG_MAX
: wenn der richtige Wert außerhalb des Bereichs der darstellbaren Werte liegt. `errno` wird auf ERANGE gesetzt, um den Fehler anzuzeigen.

Hinweise

In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Da 0 sowohl bei einem Fehler zurückgegeben wird als auch einen bei Erfolg gültigen Returnwert darstellt, muss eine Anwendung, die auf Fehler prüfen will, die folgenden Aktionen ausführen: `errno` wird auf 0 gesetzt, `wcstol` aufgerufen und der Wert von `errno` überprüft. Falls dieser Wert ungleich Null ist, wird angenommen, dass ein Fehler aufgetreten ist.

Siehe auch

iswalpha, iswspace, scanf, strtol, strtoll, strtoul,strtoull, wcstod, wcstoull

wcstoll - Langzeichenkette in ganze Zahl (long long) umwandeln

Definition

```
#include <wchar.h>

long long int wcstoll(const wchar_t *restrict nptr, wchar_t **restrict endptr, int base);
```

`wcstoll` wandelt den ersten Teil der Zeichenkette aus Langzeichenwerten, auf die *nptr* zeigt, in die Darstellung `long long int` um. Zuerst wird die Eingabe-Zeichenkette aus Langzeichenwerten in drei Teile zerlegt:

– eine möglicherweise leere Folge von Zwischenraumzeichen als Langzeichenwerte (entsprechend der Angabe durch `iswspace`) am Anfang,

– eine Folge, die als ganze Zahl mit einer Dezimalzeichen-Darstellung interpretiert wird, die durch den Wert von *base* bestimmt wird,

– und schließlich eine Zeichenkette aus Langzeichenwerten mit einem oder mehr nicht erkannten Langzeichenwerten, einschließlich abschließendem Nullbyte der Eingabe-Zeichenkette aus Langzeichenwerten.

Dann wird versucht, die mittlere Folge in eine ganze Zahl umzuwandeln. Anschließend wird das Ergebnis zurückgegeben.

Wenn der Wert von *base* gleich Null ist, wird als Format der mittleren Folge eine dezimale Konstante, oktale Konstante oder hexadezimale Konstante erwartet. Dieser kann + bzw. - vorangestellt sein. Eine dezimale Konstante beginnt mit einer Ziffer ungleich Null und besteht aus einer Folge dezimaler Ziffern. Eine oktale Konstante besteht aus dem Präfix 0 und optional einer Folge nur dezimaler Ziffern. Eine hexadezimale Konstante besteht aus dem Präfix 0x bzw. 0X und einer Folge dezimaler Ziffern und der Buchstaben a (bzw. A) bis f (bzw. F) mit den Werten 10 bis 15.

Wenn der Wert von *base* zwischen 2 und 36 liegt, wird als Format der mittleren Folge eine Sequenz von Buchstaben und Ziffern erwartet, die eine ganze Zahl darstellt mit der Basis, die durch *base* bestimmt wird (allerdings keine ganze Zahl mit Suffix). Optional kann das Vorzeichen + bzw. - vorangestellt sein. Den Buchstaben von a (bzw. A) bis einschließlich z (bzw. Z) sind die Werte 10 bis 35 zugeordnet. Es sind nur Buchstaben zulässig, deren Wert kleiner ist als der Wert von *base*. Ist der Wert von *base* gleich 16, können die Darstellungen 0x bzw. 0X für Langzeichenwerte, gegebenenfalls mit Vorzeichen, der Zeichen- und Buchstabenfolge voranstehen.

Diese mittlere Folge ist als die längste beginnende Teilfolge der Eingabe-Zeichenkette aus Langzeichenwerten definiert. Sie beginnt mit dem ersten Langzeichenwert, der kein Zwischenraumzeichen ist und das erwartete Format aufweist. Diese Folge enthält keine Langzeichenwerte, wenn die Eingabe-Zeichenkette aus Langzeichenwerten leer ist oder nur aus Langzeichenwerten besteht, die Zwischenraumzeichen sind, bzw. wenn der erste Langzeichenwert, der kein Zwischenraumzeichen ist, etwas anderes als ein Vorzeichen oder ein zulässiger Buchstabe bzw. eine zulässige Ziffer ist.

Wenn diese mittlere Folge das erwartete Format aufweist und der Wert von *base* gleich Null ist, wird die Folge der Langzeichenwerte, die mit der ersten Ziffer beginnt, als Integer-Konstante interpretiert. Wenn diese mittlere Folge das erwartete Format aufweist und der Wert von *base* zwischen 2 und 36 liegt, wird sie als Grundlage für die Umwandlung verwendet. Jedem Buchstaben wird sein Wert (siehe oben) zugeordnet. Wenn die Folge mit einem Minuszeichen beginnt, ist das Ergebnis der Umwandlung negativ. Wenn *endptr* kein NULL-Zeiger ist, wird ein Zeiger auf die abschließende Zeichenkette aus Langzeichenwerten in dem Objekt abgelegt, auf das *endptr* zeigt.

Wenn diese mittlere Folge leer ist oder nicht das erwartete Format aufweist, wird keine Umwandlung durchgeführt. Der Wert von *nptr* wird in dem Objekt abgelegt, auf das *endptr* zeigt, wenn *endptr* kein NULL-Zeiger ist.

Returnwert konvertierter Wert
bei Erfolg.

0 wenn keine Umwandlung durchgeführt werden konnte.
`errno` wird auf EINVAL gesetzt, wenn der Wert von *base* nicht unterstützt wird.

LLONG_MAX, LLONG_MIN
abgängig vom Vorzeichen des Wertes.

ULLONG_MAX
wenn der richtige Wert außerhalb des Bereichs der darstellbaren Werte liegt. `errno` wird auf ERANGE gesetzt, um den Fehler anzuzeigen.

Hinweise In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Da 0 sowohl bei einem Fehler zurückgegeben wird als auch bei Erfolg einen gültigen Returnwert darstellt, muss eine Anwendung, die auf Fehler prüfen will, die folgenden Aktionen ausführen: `errno` wird auf 0 gesetzt, `wcstoll` aufgerufen und der Wert von `errno` überprüft. Falls dieser Wert ungleich Null ist, wird angenommen, dass ein Fehler aufgetreten ist.

Der C-Compiler, der den Datentyp `long long` unterstützt, erzeugt nur Objekte im LLM-Format. Aus diesem Grunde werden auch die `long long`-Bibliotheksfunktionen nur als LLM's zur Verfügung gestellt und sind nicht in den Großmodulen enthalten. Sie müssen wie Datenmodule entweder fest eingebunden oder aus der Bibliothek nachgeladen werden.

Siehe auch iswalpha, iswspace, scanf, strtol, strtoll, strtoul, strtoull, wcstod, wcstol, wcstoul

wcstombs - Langzeichen in Multibyte-Zeichenkette umwandeln

Definition #include <stdlib.h>

size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);

`wcstombs` wandelt eine Folge von Langzeichen (`wchar_t`-Werten) in *pwcs* in die entsprechenden Multibyte-Zeichen um und speichert diese in die Zeichenkette *s*.
n gibt die maximale Anzahl Bytes an, die in *s* abgespeichert werden sollen.

Die Zuweisung wird beendet, wenn

- das Langzeichen 0 in *pwcs* auftritt,
- bereits *n* Bytes zugewiesen wurden oder
- ein Langzeichen nicht in einem Byte dargestellt werden kann.

Returnwert (`size_t`)-1 wenn ein Langzeichen nicht in ein Multibyte-Zeichen umgewandelt werden kann.

Anzahl der zugewiesenen Bytes
sonst.

Hinweise Wenn ein Langzeichen in *pwcs* nicht in ein Multibyte-Zeichen umgewandelt werden kann, werden die bereits vorher umgewandelten Langzeichen in *s* abgespeichert.

Bei sich überlappenden Speicherbereichen ist das Verhalten undefiniert.

In dieser Implementierung sind Zeichen, die aus mehreren Bytes bestehen, nicht realisiert. Multibyte-Zeichen und Langzeichen haben immer die Länge 1 Byte. Mit `wcstombs` wird jedes Langzeichen in *pwcs* in ein1 Byte langes Multibyte-Zeichen umgewandelt und in der Zeichenkette *s* gespeichert.

Siehe auch mblen, mbtowc, mbstowcs, wctomb

wcstoul - Langzeichenkette in ganze Zahl (unsigned long) umwandeln

Definition

```
#include <wchar.h>

unsigned long int wcstoul(const wchar_t *nptr, wchar_t **endptr, int base);
```

`wcstoul` wandelt den ersten Teil der Zeichenkette aus Langzeichenwerten, auf die *nptr* zeigt, in die Darstellung `unsigned long int` um. Zuerst wird die Eingabe-Zeichenkette aus Langzeichenwerten in drei Teile zerlegt:

– eine möglicherweise leere Folge von Zwischenraumzeichen als Langzeichenwerte (entsprechend der Angabe durch `iswspace`) am Anfang,

– ein Folge, die als ganze Zahl mit einer Dezimalzeichen-Darstellung interpretiert wird, die durch den Wert von *base* bestimmt wird,

– und schließlich eine Zeichenkette aus Langzeichenwerten mit einem oder mehr nicht erkannten Langzeichenwerten, einschließlich abschließendem Nullbyte der Eingabe-Zeichenkette aus Langzeichenwerten.

Dann wird versucht, die mittlere Folge in eine ganze Zahl vom Typ `unsigned long int` umzuwandeln. Anschließend wird das Ergebnis zurückgegeben.

Wenn der Wert von *base* gleich Null ist, wird als Format der mittleren Folge eine dezimale Konstante, oktale Konstante oder hexadezimale Konstante erwartet. Dieser kann + bzw. - vorangestellt sein. Eine dezimale Konstante beginnt mit einer Ziffer ungleich Null und besteht aus einer Folge dezimaler Ziffern. Eine oktale Konstante besteht aus dem Präfix 0 und optional einer Folge nur dezimaler Ziffern. Eine hexadezimale Konstante besteht aus dem Präfix 0x bzw. 0X und einer Folge dezimaler Ziffern und der Buchstaben a (bzw. A) bis f (bzw. F) mit den Werten 10 bis 15.

Wenn der Wert von *base* zwischen 2 und 36 liegt, wird als Format der mittleren Folge eine Sequenz von Buchstaben und Ziffern erwartet, die eine ganze Zahl darstellt mit der Basis, die durch *base* bestimmt wird (allerdings keine ganze Zahl mit Suffix). Optional kann das Vorzeichen + bzw. - vorangestellt sein. Den Buchstaben von a (bzw. A) bis einschließlich z (bzw. Z) sind die Werte 10 bis 35 zugeordnet. Es sind nur Buchstaben zulässig, deren Wert kleiner ist als der Wert von *base*. Ist der Wert von *base* gleich 16, können die Darstellungen 0x bzw. 0X für Langzeichenwerte, gegebenenfalls mit Vorzeichen, der Zeichen- und Buchstabenfolge voranstehen.

Diese mittlere Folge ist als die längste beginnende Teilfolge der Eingabe-Zeichenkette aus Langzeichenwerten definiert. Sie beginnt mit dem ersten Langzeichenwert, der kein Zwischenraumzeichen ist und das erwartete Format aufweist. Diese Folge enthält keine Langzeichenwerte, wenn die Eingabe-Zeichenkette aus Langzeichenwerten leer ist oder nur aus Langzeichenwerten besteht, die Zwischenraumzeichen sind, bzw. wenn der erste Langzeichenwert, der kein Zwischenraumzeichen ist, etwas anderes als ein Vorzeichen oder ein zulässiger Buchstabe bzw. eine zulässige Ziffer ist.

Wenn diese mittlere Folge das erwartete Format aufweist und der Wert von *base* gleich Null ist, wird die Folge der Langzeichenwerte, die mit der ersten Ziffer beginnt, als Integer-Konstante interpretiert. Wenn diese mittlere Folge das erwartete Format aufweist und der Wert von *base* zwischen 2 und 36 liegt, wird sie als Grundlage für die Umwandlung verwendet. Jedem Buchstaben wird sein Wert (siehe oben) zugeordnet. Wenn die Folge mit einem Minuszeichen beginnt, ist das Ergebnis der Umwandlung negativ. Wenn *endptr* kein NULL-Zeiger ist, wird ein Zeiger auf die abschließende Zeichenkette aus Langzeichenwerten in dem Objekt abgelegt, auf das *endptr* zeigt.

Wenn diese mittlere Folge leer ist oder nicht das erwartete Format aufweist, wird keine Umwandlung durchgeführt. Der Wert von *nptr* wird in dem Objekt abgelegt, auf das *endptr* zeigt, wenn *endptr* kein NULL-Zeiger ist.

Returnwert

konvertierter Wert
: bei Erfolg.

0
: wenn keine Umwandlung durchgeführt werden konnte. `errno` wird auf EINVAL gesetzt, wenn der Wert von *base* nicht unterstützt wird.

`LONG_MAX, LONG_MIN`
: abgängig vom Vorzeichen des Wertes.

`ULONG_MAX`
: wenn der richtige Wert außerhalb des Bereichs der darstellbaren Werte liegt. `errno` wird auf ERANGE gesetzt, um den Fehler anzuzeigen.

Hinweise

In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Da 0 sowohl bei einem Fehler zurückgegeben wird als auch bei Erfolg einen gültigen Returnwert darstellt, muss eine Anwendung, die auf Fehler prüfen will, die folgenden Aktionen ausführen: `errno` wird auf 0 gesetzt, `wcstoul` aufgerufen und der Wert von `errno` überprüft. Falls dieser Wert ungleich Null ist, wird angenommen, dass ein Fehler aufgetreten ist.

Siehe auch

iswalpha, iswspace, scanf, strtol, strtoll, strtoul, stroull, wcstod, wcstol, wcstoll

wcstoull - Langzeichenkette in ganze Zahl (unsigned long long) umwandeln

Definition

```
#include <wchar.h>

unsigned long long int wcstoull(const wchar_t *restrict nptr, wchar_t **restrict endptr,
                                int base);
```

`wcstoull` wandelt den ersten Teil der Zeichenkette aus Langzeichenwerten, auf die *nptr* zeigt, in die Darstellung `unsigned long int` um. Zuerst wird die Eingabe-Zeichenkette aus Langzeichenwerten in drei Teile zerlegt:

– eine möglicherweise leere Folge von Zwischenraumzeichen als Langzeichenwerte (entsprechend der Angabe durch `iswspace`) am Anfang,

– ein Folge, die als ganze Zahl mit einer Dezimalzeichen-Darstellung interpretiert wird, die durch den Wert von *base* bestimmt wird,

– und schließlich eine Zeichenkette aus Langzeichenwerten mit einem oder mehr nicht erkannten Langzeichenwerten, einschließlich abschließendem Nullbyte der Eingabe-Zeichenkette aus Langzeichenwerten.

Dann wird versucht, die mittlere Folge in eine ganze Zahl vom Typ `unsigned long int` umzuwandeln. Anschließend wird das Ergebnis zurückgegeben.

Wenn der Wert von *base* gleich Null ist, wird als Format der mittleren Folge eine dezimale Konstante, oktale Konstante oder hexadezimale Konstante erwartet. Dieser kann + bzw. - vorangestellt sein. Eine dezimale Konstante beginnt mit einer Ziffer ungleich Null und besteht aus einer Folge dezimaler Ziffern. Eine oktale Konstante besteht aus dem Präfix 0 und optional einer Folge nur dezimaler Ziffern. Eine hexadezimale Konstante besteht aus dem Präfix 0x bzw. 0X und einer Folge dezimaler Ziffern und der Buchstaben a (bzw. A) bis f (bzw. F) mit den Werten 10 bis 15.

Wenn der Wert von *base* zwischen 2 und 36 liegt, wird als Format der mittleren Folge eine Sequenz von Buchstaben und Ziffern erwartet, die eine ganze Zahl darstellt mit der Basis, die durch *base* bestimmt wird (allerdings keine ganze Zahl mit Suffix). Optional kann das Vorzeichen + bzw. - vorangestellt sein. Den Buchstaben von a (bzw. A) bis einschließlich z (bzw. Z) sind die Werte 10 bis 35 zugeordnet. Es sind nur Buchstaben zulässig, deren Wert kleiner ist als der Wert von *base*. Ist der Wert von *base* gleich 16, können die Darstellungen 0x bzw. 0X für Langzeichenwerte, gegebenenfalls mit Vorzeichen, der Zeichen- und Buchstabenfolge voranstehen.

Diese mittlere Folge ist als die längste beginnende Teilfolge der Eingabe-Zeichenkette aus Langzeichenwerten definiert. Sie beginnt mit dem ersten Langzeichenwert, der kein Zwischenraumzeichen ist und das erwartete Format aufweist. Diese Folge enthält keine Langzeichenwerte, wenn die Eingabe-Zeichenkette aus Langzeichenwerten leer ist oder nur

aus Langzeichenwerten besteht, die Zwischenraumzeichen sind, bzw. wenn der erste Langzeichenwert, der kein Zwischenraumzeichen ist, etwas anderes als ein Vorzeichen oder ein zulässiger Buchstabe bzw. eine zulässige Ziffer ist.

Wenn diese mittlere Folge das erwartete Format aufweist und der Wert von *base* gleich Null ist, wird die Folge der Langzeichenwerte, die mit der ersten Ziffer beginnt, als Integer-Konstante interpretiert. Wenn diese mittlere Folge das erwartete Format aufweist und der Wert von *base* zwischen 2 und 36 liegt, wird sie als Grundlage für die Umwandlung verwendet. Jedem Buchstaben wird sein Wert (siehe oben) zugeordnet. Wenn die Folge mit einem Minuszeichen beginnt, ist das Ergebnis der Umwandlung negativ. Wenn *endptr* kein NULL-Zeiger ist, wird ein Zeiger auf die abschließende Zeichenkette aus Langzeichenwerten in dem Objekt abgelegt, auf das *endptr* zeigt.

Wenn diese mittlere Folge leer ist oder nicht das erwartete Format aufweist, wird keine Umwandlung durchgeführt. Der Wert von *nptr* wird in dem Objekt abgelegt, auf das *endptr* zeigt, wenn *endptr* kein NULL-Zeiger ist.

Returnwert

konvertierter Wert
: bei Erfolg.

0
: wenn keine Umwandlung durchgeführt werden konnte. `errno` wird auf EINVAL gesetzt, wenn der Wert von *base* nicht unterstützt wird.

`LLONG_MAX, LLONG_MIN`
: abgängig vom Vorzeichen des Wertes.

`ULLONG_MAX`
: wenn der richtige Wert außerhalb des Bereichs der darstellbaren Werte liegt. `errno` wird auf ERANGE gesetzt, um den Fehler anzuzeigen.

Hinweise

In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Da 0 sowohl bei einem Fehler zurückgegeben wird als auch bei Erfolg einen gültigen Returnwert darstellt, muss eine Anwendung, die auf Fehler prüfen will, die folgenden Aktionen ausführen: `errno` wird auf 0 gesetzt, `wcstoull` aufgerufen und der Wert von `errno` überprüft. Falls dieser Wert ungleich Null ist, wird angenommen, dass ein Fehler aufgetreten ist.

Der C-Compiler, der den Datentyp `long long` unterstützt, erzeugt nur Objekte im LLM-Format. Aus diesem Grunde werden auch die `long long`-Bibliotheksfunktionen nur als LLM's zur Verfügung gestellt und sind nicht in den Großmodulen enthalten. Sie müssen wie Datenmodule entweder fest eingebunden oder aus der Bibliothek nachgeladen werden.

Siehe auch

iswalpha, iswspace, scanf, strtoul, wcstod, wcstol

wcsxfrm - Langzeichenkette transformieren

Definition

```
#include <wchar.h>

size_t wcsxfrm(wchar_t *ws1, const wchar_t *ws2, size_t n);
```

`wcsxfrm` transformiert die Langzeichenkette, auf die *ws2* zeigt, und schreibt das Ergebnis der Transformation in das Feld, auf das *ws1* zeigt. Die Transformation wird so durchgeführt, dass die Funktion `wcscmp` für zwei transformierte Langzeichenketten denselben Returnwert (größer, gleich oder kleiner Null) liefert, wie die Funktion `wcscoll` für die beiden ursprünglichen, nicht transformierten Langzeichenketten.
Es werden maximal *n* Langzeichen-Codes in das Feld geschrieben (inklusive des abschließenden Null-Zeichens).
Wenn *n* den Wert 0 hat, darf *wc1* ein NULL-Zeiger sein.
Falls zwischen sich überlappenden Objekten kopiert wird, ist das Ergebnis undefiniert.

Returnwert

Ganzzahliger Wert < n
: der die Anzahl der in das Feld geschriebenen Langzeichen-Codes angibt (ohne abschließende Null).

Ganzzahliger Wert ≥ n
: In diesem Falle ist der Inhalt des Feldes *ws1* unbestimmt.

(size_t) - 1
: bei Fehler. `errno` wird gesetzt, um den Fehler anzuzeigen:

EINVAL	Die Langzeichenkette, auf die *ws2* zeigt, enthält Langzeichen-Codes, die außerhalb des Wertebereichs der gewählten Sortierfolge liegen.
ENOMEM	Es steht nicht genügend Speicherplatz für die internen Verwaltungsdaten zur Verfügung.

Hinweise

In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Es wird so transformiert, dass zwei transformierte Langzeichenketten von `wcscmp` gemäß der in LC_COLLATE festgelegten Sortierfolge geordnet werden.

Die Tatsache, dass *ws1* ein NULL-Zeiger sein darf, wenn *n* den Wert 0 hat, ist nützlich, wenn die Größe des Feldes vor der Transformation bestimmt werden soll.

Da es im Standard keinen festlegten Wert für den Fehlerfall gibt, wird empfohlen, `errno` auf den Wert 0 zu setzen, dann `wcscoll` aufzurufen und nach dem Aufruf `errno` zu überprüfen. Falls `errno` ungleich 0 ist, kann angenommen werden, dass ein Fehler aufgetreten ist.

Siehe auch strxfrm, wcscmp, wcscoll

wctob - Langzeichen in (ein-byte) Multibyte-Zeichen konvertieren

Definition

```
#include <stdio.h>
#include <wchar.h>

int wctob(wint_t c);
```

`wctob` überprüft, ob das Zeichen c zu einem Element des erweiterten Zeichensatzes korrespondiert, dessen Multibyte-Darstellung im „initial shift“-Zustand aus einem Byte besteht.

Returnwert

`EOF` falls zu c kein korrespondierendes Multibyte-Zeichen der Länge eins im „initial shift“-Zustand existiert.

Multibyte-Zeichen der Länge 1 Byte
das zu c korrespondiert, sonst.

Siehe auch mblen, mbtowc, wcstombs, wctomb

wctomb - Langzeichen in Multibyte-Zeichen umwandeln

Definition

```
#include <stdlib.h>

int wctomb(char *s, wchar_t wc);
```

`wctomb` wandelt das Langzeichen *wc* in das entsprechende Multibyte-Zeichen um und speichert dieses in die Zeichenkette *s*.

Keine Zuweisung erfolgt, wenn *s* ein NULL-Zeiger ist oder wenn das Langzeichen nicht in einem Byte dargestellt werden kann.

Returnwert

0	falls *s* ein NULL-Zeiger ist.
-1	falls das Langzeichen nicht in ein Multibyte-Zeichen umgewandelt werden kann.
1	sonst.

Hinweis In dieser Implementierung sind Zeichen, die aus mehreren Bytes bestehen, nicht realisiert. Multibyte-Zeichen und Langzeichen haben immer die Länge 1 Byte.

Siehe auch mblen, mbtowc, wcstombs, wcrtomb

wctrans - Abbildung zwischen Langzeichen definieren

Definition

```
#include <wctype.h>

wctrans_t wctrans(const char *property);
```

`wctrans` konstruiert aus *property* einen Wert des Typs `wctrans_t`, der eine Abbildung zwischen Langzeichen beschreibt.

Die beiden Zeichenketten "`tolower`" und "`toupper`" sind in allen Lokalitäten als Werte des Arguments *property* zugelassen.

Wenn *property* eine Abbildung identifiziert, die gemäß der LC_CTYPE-Kategorie der aktuellen Lokalität gültig ist, gibt `wctrans` einen Wert ungleich 0 zurück, der als gültiges zweites Argument der Funktion `towctrans` verwendet werden kann.

Returnwert

Wert ≠ 0	wenn *property* eine gültige Abbildung identifiziert.
0	sonst.

Hinweis In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Siehe auch towctrans

wctype - Langzeichenklasse definieren

Definition

```
#include <wctype.h>

wctype_t wctype(const char *charclass);
```

`wctype` ist für gültige Namen von Zeichenklassen definiert, wie sie in der aktuellen Umgebung festgelegt sind. *charclass* ist eine Zeichenkette, die eine generische Zeichenklasse angibt, für die zeichensatzspezifische Typinformationen benötigt werden. Die folgenden Namen von Zeichenklassen sind in jeder Umgebung definiert: `"alnum"`, `"alpha"`, `"blank"`, `"cntrl"`, `"digit"`, `"graph"`, `"lower"`, `"print"`, `"punct"`, `"space"`, `"upper"` und `"xdigit"`.

Es können weitere Namen von Zeichenklassen angegeben werden, wenn sie in der Definitionsdatei der Umgebung definiert sind (Kategorie LC_CTYPE).

Die Funktion gibt einen Wert vom Typ `wctype_t` zurück. Dieser Wert kann als zweites Argument für einen darauffolgenden Aufruf von `iswctype` verwendet werden. `wctype` bestimmt entsprechend den Regeln des durch die Zeichentyp-Informationen der Umgebung (Kategorie LC_CTYPE) definierten Zeichensatzes `wctype_t`-Werte. Die von `wctype` zurückgegebenen Werte sind solange gültig, bis ein Aufruf von `setlocale` die Kategorie LC_CTYPE modifiziert.

Returnwert

0	wenn der Name der Zeichenklasse in der aktuellen Lokalität nicht gültig ist (Kategorie LC_CTYPE).
$\neq 0$	es wird ein Objekt vom Typ wctype_t zurückgegeben, das in Aufrufen von iswctype verwendet werden kann.

Hinweis In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Siehe auch iswctype

wmemchr - Langzeichenkette nach Langzeichen durchsuchen

Definition #include <wchar.h>

wchar_t *wmemchr(const wchar_t *ws, wchar_t *wc, size_t n);

`wmemchr`sucht das erste Vorkommen des Langzeichens *wc* in den ersten *n* Bytes der Langzeichenkette *ws* und liefert bei Erfolg einen Zeiger auf die gesuchte Position in *ws*.

Returnwert Zeiger auf die Position von *wc* in *ws*
bei Erfolg,

NULL-Zeiger sonst.

Hinweise In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Für C++ gelten die beiden folgenden Prototypen für die Funktion `wmemchr`:
const wchar_t* wmemchr(const wchar_t *ws, wchar_t *wc, size_t n);
wchar_t* wmemchr(wchar_t *ws, wchar_t *wc, size_t n);

Siehe auch memchr, wcsstr, wmemcmp, wmemcpy

wmemcmp - zwei Langzeichenketten vergleichen

Definition

```
#include <wchar.h>

int wmemchr(const wchar_t *ws1, const wchar_t *ws2, size_t n);
```

`wmemcmp` vergleicht die ersten *n* Bytes der beiden Langzeichenketten *ws1* und *ws2* lexikalisch.

Returnwert

< 0	*ws1* ist lexikalisch kleiner als *ws2.*
= 0	*ws1* und *ws2* sind lexikalisch gleich groß.
> 0	*ws1* ist lexikalisch größer als *ws2.*

Hinweis In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Siehe auch memcmp, wcsstr, wmemchr, wmemcpy

wmemcpy - Langzeichenkette kopieren

Definition

```
#include <wchar.h>

wchar_t *wmemcpy(wchar_t *ws1, const wchar_t *ws2, size_t n);
```

`wmemcpy` kopiert die ersten *n* Bytes der Langzeichenkette *ws2* in die ersten *n* Bytes der Langzeichenkette *ws1*.

Returnwert Zeiger auf die Langzeichenkette *ws1*.

Hinweis In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Siehe auch memcmp, wmemmove, wmemset

wmemmove - Langzeichenkette in überlappenden Bereich kopieren

Definition
```
#include <wchar.h>
wchar_t *wmemmove(wchar_t *ws1, const wchar_t *ws2, size_t n);
```

`wmemmove` kopiert die ersten *n* Bytes der Langzeichenkette *ws2* in die ersten *n* Bytes der Langzeichenkette *ws1*. Das Kopieren findet statt, als ob die *n* Langzeichen zuerst in ein temporäres Feld kopiert würden, das weder *ws1* noch *ws2* überlappt, und anschließend von diesem Feld in *ws1*.

Returnwert Zeiger auf die Langzeichenkette *ws1*.

Hinweis In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Siehe auch memmove, wmemcpy, wmemset

wmemset - ersten *n* Langzeichen in Langzeichenkette setzen

Definition
```
#include <wchar.h>
wchar_t *wmemset(wchar_t *ws, wchar_t *c, size_t n);
```

`wmemset` setzt die ersten *n* Langzeichen in der Langzeichenkette *ws* auf den Wert *c*.

Returnwert Zeiger auf *ws*.

Hinweis In dieser Version des C-Laufzeitsystems werden nur 1-Byte-Zeichen als Langzeichen unterstützt.

Siehe auch memset, wmemcpy, wmemmove

wprintf - Langzeichen formatiert ausgeben

```
#include <wchar.h>
int wprintf(const wchar_t *format [, arglist]);
```

Beschreibung siehe `fwprintf`.

write - In eine Datei schreiben (elementar)

Definition

```
#include <stdio.h>

int write(int dk, const char *puf, int anz);
```

`write` ist die elementare Schreiboperation.

`write` schreibt bis zu *anz* zusammenhängende Bytes aus dem Bereich, auf den *puf* zeigt, in die Datei mit Dateikennzahl *dk*.

SAM-Dateien werden mit elementaren Funktionen stets als Textdateien verarbeitet.

Parameter

int dk
: Dateikennzahl der Ausgabedatei.

  Eine Dateikennzahl (positive ganze Zahl) ist das Ergebnis eines erfolgreichen Aufrufs von `open`/`open64` oder `creat`/`creat64`.
  Die Dateikennzahlen für `stdin` (0), `stdout` (1) und `stderr` (2) sind nach Programmstart automatisch zugeordnet.

const char *puf
: Zeiger auf den Bereich, in dem die Daten stehen, die in die Ausgabedatei geschrieben werden sollen.

int anz
: Anzahl der Bytes, die in die Datei geschrieben werden sollen. Es ist nicht sichergestellt, dass `write` tatsächlich *anz* Bytes schreibt!

Returnwert

Anzahl der tatsächlich geschriebenen Bytes, bei Erfolg.

-1 `write` hat nichts geschrieben, weil einer der folgenden Fehler vorliegt:
- physikalischer Ein-/Ausgabefehler
- *dk* ist keine gültige Dateikennzahl
- die Datei ist nicht vorhanden
- es besteht kein Zugriffsrecht bzw. keine Schreiberlaubnis für die Datei
- der Bereich, in dem die Daten stehen, ist nicht korrekt angegeben

Hinweise Sie sollten nach jedem `write`-Aufruf die Anzahl der tatsächlich geschriebenen Bytes überprüfen.
Ist das Ergebnis kleiner als die Angabe in *anz*, liegt im Allgemeinen ein Fehler vor.
Ist das Ergebnis größer als die Angabe in *anz*, wurden Tabulatorzeichen (\t) in eine Textdatei geschrieben. Tabulatorzeichen werden dabei in die entsprechenden Leerzeichen umgesetzt und bei der Ergebnisanzahl berücksichtigt.

Um sicherzugehen, dass Ihre Angabe in *anz* die Größe des Puffers nicht überschreitet, sollten Sie die Funktion `sizeof` verwenden.

Die Daten werden nicht sofort in die externe Datei geschrieben, sondern in einem C-internen Puffer zwischengespeichert (siehe Abschnitt „Pufferung“ auf Seite 65).

Bei der Ausgabe in Textdateien werden die Steuerzeichen für Zwischenraum (\n, \t, etc.) je nach Art der Textdatei in ihre entsprechende Wirkung umgesetzt (siehe Abschnitt „Zwischenraum“ auf Seite 68).

Beispiel Folgendes Programm kopiert die Standardeingabe (Dateikennzahl 0) auf die Standardausgabe (Dateikennzahl 1). Wenn Sie den Umlenkmechanismus ausnützen, können Sie damit von einer beliebigen Quelle auf ein beliebiges Ziel kopieren. BUFSIZ (8192 Bytes) ist in der Include-Datei <stdio.h> definiert.

```
#include <stdio.h>

int main(void)
{
  char buf[BUFSIZ];
  int n;

  while((n = read(0, buf, sizeof(buf))) > 0)
        write(1, buf, n);
  return 0;
}
```

Siehe auch read, open, open64, creat, creat64

wscanf - formatiert lesen

#include <wchar.h>

int wscanf(const wchar_t *format [, arglist]);

Beschreibung siehe `fwscanf`.

y0, y1, yn - Besselfunktionen der zweiten Art

Definition

```
#include <math.h>

double y0(double x);

double y1(double x);

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

Die Funktionen `y0`, `y1` und `yn` berechnen die Besselfunktionen der zweiten Art für reelle Argumente x und die ganzzahligen Ordnungen 0, 1 bzw. n.

Returnwert Besselfunktion für das reelle Argument x>0.

-HUGE_VAL bei Argumenten ≤ 0. Zusätzlich wird `errno` auf EDOM gesetzt (domain error, d.h. unzulässiges Argument).

Siehe auch j0, j1, jn

8 Anhang

Übersicht über die Funktionen im BS2000 und ANSI-Standard

Auf den folgenden Seiten sind alle Funktionen aufgelistet, die das C-Laufzeitsystem zur Verfügung stellt. Bei jeder Funktion wird angegeben, ob sie im ANSI-Standard definiert ist oder eine Erweiterung darstellt.

Es bedeuten:

X ANSI-Standard

A AMENDMENT 1 zum Standard ISO/IEC 9899:1990

\- Erweiterung, in einem ANSI-definierten Include-Header deklariert

o Erweiterung, in einem BS2000-spezifischen Include-Header deklariert (keine Abfrage des Defines `_STRICT_STDC`, siehe Seite 42).

Funktion	ANSI
_a2e, _e2a	-
_a2e_dup, _e2a_dup	-
_a2e_dup_n, _e2a_dup_n	-
_a2e_max, _e2a_max	-
_a2e_n, _e2a_n	-
abort	X
abs	X
acos	X
alarm	-
asctime	X
asin	X
assert	X
atan	X
atan2	X
atexit	X
atof	X
atoi	X
atol	X
atoll	-
bs2cmd	-
bs2exit	-
bs2fstat	-
bsearch	X
btowc	A
cabs	-
calloc	X
cdisco	o
ceil	X
cenaco	o
clearerr	X
clock	X
close	-
cos	X

Funktion	ANSI
cosh	X
cputime	-
creat	-
creat64	-
cstxit	o
_cstxit	o
ctime	X
ctime64	-
_ _DATE_ _	X
difftime	X
difftime64	-
div	X
double2ieee	-
ecvt	-
_edt	-
erf, erfc	-
exit	X
_exit	-
exp	X
fabs	X
fclose	X
fcvt	-
fdelrec	-
fdopen	-
feof	X
ferror	X
fflush	X
fgetc	X
fgetpos	X
fgetpos64	X
fgets	X
fgetwc	A
fgetws	A

Funktion	ANSI
_ _FILE_ _	X
float2ieee	-
flocate	-
floor	X
fmod	X
fopen	X
fprintf	X
fputc	X
fputwc	A
fputws	A
fputs	X
fread	X
free	X
freopen	X
freopen64	-
frexp	X
fscanf	X
fseek	X
fseek64	-
fsetpos	X
fsetpos64	-
ftell	X
ftell64	-
ftime	o
ftime64	-
fwide	A
fwprintf	A
fwrite	X
fwscanf	A
gamma	-
garbcoll	-
gcvt	-
getc	X

Funktion	ANSI
getchar	X
getenv	X
getlogin	-
getpgmname	-
gets	X
gettsn	-
getw	-
getwc	A
getwchar	A
gmtime	X
gmtime64	-
hypot	-
ieee2double	-
ieee2float	-
index	-
isalnum	X
isalpha	X
isascii	-
iscntrl	X
isdigit	X
isebcdic	-
isgraph	X
islower	X
isprint	X
ispunct	X
isspace	X
isupper	X
iswalnum	A
iswalpha	A
iswcntrl	A
iswctype	A
iswdigit	A
iswgraph	A

Funktion	ANSI
iswlower	A
iswprint	A
iswpunct	A
iswspace	A
iswupper	A
iswxdigit	A
isxdigit	X
j0, j1, jn	-
kill	-
labs	X
ldexp	X
ldiv	X
_ _LINE_ _	X
llabs	-
lldiv	-
llrint	-
llrintf	-
llrintl	-
llround	-
llroundf	-
llroundl	-
localeconv	X
localtime	X
localtime64	-
log	X
log10	X
longjmp	X
lrint	-
lrintf	-
lrintl	-
lround	-
lroundf	-
lroundl	-

Funktion	ANSI
lseek	-
lseek64	-
malloc	X
mblen	X
mbrlen	A
mbrtowc	A
mbsinit	A
mbsrtowcs	A
mbstowcs	A
mbtowc	X
memalloc	-
memchr	X
memcmp	X
memcpy	X
memfree	-
memmove	X
memset	X
mktemp	-
mktime	X
mktime64	-
modf	X
offsetof	X
open	-
open64	-
perror	X
pow	X
printf	X
putc	X
putchar	X
puts	X
putw	-
putwc	A
putwchar	A

Funktion	ANSI
qsort	X
raise	X
rand	X
read	-
realloc	X
remove	X
rename	X
rewind	X
rindex	-
rint	-
rintf	-
rintl	-
round	-
roundf	-
roundl	-
scanf	X
setbuf	X
setjmp	X
setlocale	X
setvbuf	X
signal	X
sin	X
sinh	X
sleep	-
snprintf	-
sprintf	X
sqrt	X
srand	X
sscanf	X
_ _STDC_ _	X
_ _STDC_ _VERSION	A
strcat	X
strchr	X

Funktion	ANSI
strcmp	X
strcoll	X
strcpy	X
strcspn	X
strerror	X
strfill	-
strftime	X
strlen	X
strlower	-
strncat	X
strncmp	X
strncpy	X
strpbrk	X
strrchr	X
strspn	X
strstr	X
strtod	X
strtok	X
strtol	X
strtoll	-
strtoul	X
strtoull	-
strupper	-
strxfrm	X
swprintf	A
swscanf	A
system	X
tan	X
tanh	X
tell	-
time	X
time64	-
_ _TIME_ _	X

Funktion	ANSI
tmpfile	X
tmpfile64	-
tmpnam	X
toascii	-
toebcdic	-
tolower	X
toupper	X
towctrans	A
towlower	A
towupper	A
ungetc	X
ungetwc	A
unlink	-
va_arg	X
va_end	X
va_start	X
vfprintf	X
vfwprint	A
vprintf	X
vsnprintf	-
vsprintf	X
vswprintf	A
vwprintf	A
wcrtomb	A
wcscat	A
wcschr	A
wcscmp	A

Funktion	ANSI
wcscoll	A
wcscpy	A
wcscspn	A
wcsftime	A
wcslen	A
wcsncat	A
wcsncmp	A
wcsncpy	A
wcspbrk	A
wcsrchr	A
wcsrtombs	A
wcsspn	A
wcsstr	A
wcstombs	X
wctob	A
wcstod	A
wcstok	A
wcstol	A
wcstoll	A
wcstoul	A
wcstoull	A
wcsxfrm	A
wctomb	A
wctrans	A
wctype	A
wmemcmp	A
wmemcpy	A

KR- oder ANSI-Funktionalität für C/C++ kleiner V3.0

Die C-Bibliotheksfunktionen wurden erstmals mit C V1.0 zur Verfügung gestellt. Zu diesem Zeitpunkt gab es keinen ANSI-definierten C-Bibliotheksumfang. Die Implementierung orientierte sich an der „vorläufigen“ Definition durch Kernighan/Ritchie bzw. an den marktüblichen UNIX/SINIX-Implementierungen.
Die Anpassung der C-Bibliotheksfunktionen an den ANSI-Standard (C V2.0) führte bei der Ausführung einiger Ein-/Ausgabefunktionen zu Abweichungen gegenüber der Vorgängerversion. Um einerseits den ANSI-Standard voll zu erfüllen, andererseits das gewohnte Ablaufverhalten von „Alt“-Programmen zu gewährleisten, werden die von den Abweichungen betroffenen Ein-/Ausgabefunktionen bis C und C++ V2.2C in zwei Varianten angeboten: Mit der neuen „ANSI“-Funktionalität und mit der zu C V1.0 kompatiblen „KR“-Funktionalität.

Die gewünschte Funktionalität wird zum Übersetzungszeitpunkt mit folgender Compileroption ausgewählt:

```
SOURCE-PROPERTIES = PAR(LIBRARY-SEMANTICS = STD / V1-COMPATIBLE)
```

Die Auswahl der KR-Funktionalität (V1-COMPATIBLE) ist nur in den Übersetzungsmodi KR und ANSI möglich. In den Übersetzungsmodi STRICT-ANSI und CPLUSPLUS wird die Angabe V1-COMPATIBLE ignoriert und automatisch STD angenommen.

Die Unterschiede zwischen KR- und ANSI-Funktionalität sind auf den folgenden Seiten aufgeführt.

Die KR- bzw. ANSI-Funktionalität gilt für die Aufrufe aller Bibliotheksfunktionen einer Übersetzungseinheit.

Achtung

Wird in mehreren, getrennt übersetzten Quellprogrammen dieselbe Datei verarbeitet, müssen diese Quellprogramme mit dem gleichen LIBRARY-SEMANTICS-Parameter übersetzt werden!

KR-Funktionalität

1. Standardattribute von Textdateien

   Wird eine nicht vorhandene Textdatei neu angelegt, wird standardmäßig eine SAM-Datei mit variabler Satzlänge erstellt.

2. Position des Schreib-/Lesezeigers im Anfügemodus

   Wenn der Schreib-/Lesezeiger in einer Datei, die im Anfügemodus geöffnet wurde, explizit vom Dateiende wegpositioniert wurde (`rewind`, `fsetpos`, `fseek`, `lseek`), wird er nur beim Schreiben mit der Elementarfunktion `write` ignoriert und automatisch ans Ende der Datei positioniert.

   Wenn eine Datei im Anfügemodus und zum Lesen geöffnet wurde, ist sie nach dem Öffnen auf Dateiende positioniert. Der alte Inhalt bereits vorhandener Dateien bleibt erhalten.

3. ISAM-Dateien (Pufferleerung)

   Wenn die Daten einer ISAM-Datei im Puffer nicht mit einem Neue-Zeile-Zeichen enden, bewirkt das Schreiben in die externe Datei einen Satzwechsel. Nachfolgende Daten werden in einen neuen Satz geschrieben.

4. `ungetc`

   Beim Schreiben des Pufferinhalts in die externe Datei werden die Originaldaten verändert, wenn an Stelle des zuvor eingelesenen Zeichens ein anderes Zeichen in den Puffer zurückgestellt wurde.

5. Auswertung des Tabulatorzeichens (\t)

   Bei der Ausgabe in Textdateien vom FCBTYP SAM und ISAM wird das Tabulatorzeichen standardmäßig in die entsprechende Anzahl Leerzeichen umgesetzt.

6. `fprintf`, `printf`, `sprintf`, `fscanf`, `scanf`, `sscanf`

   Die ANSI-Erweiterungen der Formatierungs- und Umwandlungszeichen stehen nicht zur Verfügung. Es gilt die Syntax und Semantik der Version C V1.0.

ANSI-Funktionalität

1. Standardattribute von Textdateien

   Wird eine nicht vorhandene Textdatei neu angelegt, wird standardmäßig eine ISAM-Datei mit variabler Satzlänge erstellt.

2. Position des Schreib-/Lesezeigers im Anfügemodus

   Wenn der Schreib-/Lesezeiger in einer Datei, die im Anfügemodus geöffnet wurde, explizit vom Dateiende wegpositioniert wurde (`rewind`, `fsetpos`, `fseek`, `lseek`), wird er bei allen Schreibfunktionen ignoriert und automatisch ans Ende der Datei positioniert.

   Wenn eine Datei im Anfügemodus und zum Lesen geöffnet wurde, ist sie nach dem Öffnen auf Dateianfang positioniert. Der alte Inhalt bereits vorhandener Dateien bleibt erhalten.

3. ISAM-Dateien (Pufferleerung)

   Wenn die Daten einer ISAM-Datei im Puffer nicht mit einem Neue-Zeile-Zeichen enden, bewirkt das Schreiben in die externe Datei keinen Satzwechsel. Nachfolgende Daten verlängern den Satz in der Datei. Beim Lesen einer ISAM-Datei werden daher nur Neue-Zeile-Zeichen eingelesen, die vom Programm explizit geschrieben wurden.

   Wenn das Lesen aus einer beliebigen Textdatei eine Datenübertragung von der externen Datei in den C-internen Puffer notwendig macht, werden die noch in Puffern zwischengespeicherten Daten aller ISAM-Dateien automatisch in die Dateien hinausgeschrieben.

4. `ungetc`

   Beim Schreiben des Pufferinhalts in die externe Datei werden die Originaldaten nicht verändert, wenn an Stelle des zuvor eingelesenen Zeichens ein anderes Zeichen in den Puffer zurückgestellt wurde. Es werden stets die Originaldaten vor dem `ungetc`-Aufruf in die externe Datei geschrieben.

5. Auswertung des Tabulatorzeichens (\t)

   Bei der Ausgabe in Textdateien vom FCBTYP SAM und ISAM wird das Tabulatorzeichen standardmäßig nicht in die entsprechende Anzahl Leerzeichen umgesetzt, sondern als Textzeichen (EBCDIC-Wert) in die Datei geschrieben.

Literatur

Die Handbücher finden Sie im Internet unter *http://manuals.ts.fujitsu.com*. Handbücher, die mit einer Bestellnummer angezeigt werden, können Sie in auch gedruckter Form bestellen.

[1] **BS2000 OSD/BC**
Softbooks Deutsch

[2] **CRTE (BS2000)**
Common RunTime Environment
Benutzerhandbuch

[3] **C-Bibliotheksfunktionen** (BS2000)
für POSIX-Anwendungen
Referenzhandbuch

[4] **C** (BS2000)
C-Compiler
Benutzerhandbuch

[5] **C/C++** (BS2000)
C/C++-Compiler
Benutzerhandbuch

[6] **BS2000 OSD/BC**
Makroaufrufe an den Ablaufteil
Benutzerhandbuch

[7] **JV** (BS2000)
Jobvariablen
Beschreibung

Sonstige Literatur

X/Open CAE Specification
System Interfaces and Headers, Issue 4

ISBN: 1-872630-47-2
X/Open Document Number: C202

X/Open CAE Specification
System Interface Definitions, Issue 4

ISBN: 1-872630-46-4
X/Open Document Number: C204

X/Open CAE Specification
Commands and Utilities, Issue 4

ISBN: 1-872630-48-0
X/Open Document Number: C203

International Standard ISO/IEC 9899 : 1990,
Programming languages - C

International Standard ISO/IEC 9899 : 1990,
Programming languages - C / Amendment 1

Stichwörter