SOFTWARE

WEGA

DIENSTPROGRAMME
BAND B

EAW *electronic*

P8000

Version 1.1 (2008-03-02)

EAW-electronic P8000

W E G A - S o f t w a r e

Dienstprogramme

(Band-B)

Diese Dokumentation wurde von einem Kollektiv des Kombinates

VEB ELEKTRO-APPARATE-WERKE
BERLIN-TREPTOW "FRIEDRICH EBERT"

erarbeitet.

Herausgeber:

Kombinat
VEB ELEKTRO-APPARATE-WERKE
BERLIN-TREPTOW "FRIEDRICH EBERT"
Hoffmannstrasse 15-26
BERLIN
1193

WAE/03-0203-02

Ausgabe: 12/88

Die vorliegende Dokumentation unterliegt nicht dem Aenderungsdienst.

Spezielle Hinweise zum aktuellen Stand der Softwarepakete befinden sich in README-Dateien auf den entsprechenden Vertriebsdisketten.

Dieser Band-B enthaelt folgende Unterlagen:

Teil 1: PGMG
WEGA-Programmierung

Teil 2: CAS
U8000-Assembler

Teil 3: SCREEN/CURSES
Bibliothek zur Bildschirmarbeit

Teil 4: UUCP
Implementierung

Teil 5: C-Besonderheiten

Gegenueber der vorherigen Ausgabe wurde der Teil 4 (UUCP) ueberarbeitet und der Teil 5 (C-Besonderheiten) neu aufgenommen.

Gesamtinhaltsverzeichnis Seite

Teil 1: PGMG

Teil 2: CAS

Teil 3: SCREEN/CURSES

Teil 4: UUCP

Hinweise des Lesers zu diesem Dokumentationsband

Wir sind staendig bemueht, unsere Unterlagen auf einem qualitativ hochwertigen Stand zu halten. Sollten Sie deshalb Hinweise zur Verbesserung dieses Dokumentationsbandes bzw. zur Beseitigung von Fehlern haben, so bitten wir Sie, diesen Fragebogen auszufuellen und an uns zurueckzusenden.

Titel des Dokumentationsbandes: WEGA-Dienstprogramme (Band-B)

Ihr Name / Tel.-Nr.:

Name und Anschrift des Betriebes:

Genuegt diese Dokumentation Ihren Anspruechen? ja / nein
Falls nein, warum nicht?

Was wuerde diese Dokumentation verbessern?

Sonstige Hinweise:

Fehler innerhalb dieser Dokumentation:

Unsere Anschrift: Kombinat VEB ELEKTRO-APPARATE-WERKE
BERLIN-TREPTOW "FRIEDRICH EBERT"
Abteilung Basissoftware
Hoffmannstrasse 15-26
BERLIN
1193

P G M G

WEGA-Programmierung

Vorwort

Dieser Artikel ist eine Einfuehrung in die Programmierung im System WEGA. Die Betonung liegt darauf, wie Programme geschrieben werden muessen, die Schnittstellen zum Betriebssystem besitzen, entweder direkt oder durch die Standard-E/A-Bibliothek. Die diskutierten Themen schliessen ein:

- die Behandlung von Kommandoargumenten
- die Standardein- und Standardausgaben
- die Standard-E/A-Bibliothek; Dateisystemzugriff
- E/A auf unterster Ebene
- Prozesse
- Signale

In diesem Artikel ist Material zusammengestellt, was ueber verschiedene Abschnitte des WEGA-Programmierhandbuchs verteilt ist. Auf Vollstaendigkeit wird dabei kein Wert gelegt, sondern es werden nur allgemein nuetzliche Dinge behandelt. Dabei wird angenommen, dass der Leser in C programmieren kann.

Inhaltsverzeichnis Seite

1. Grundlagen

1.1. Programmargumente

Wenn ein C-Programm als Kommando laeuft, werden die Argumente der Kommandozeile der Funktion main als Argumentzaehler argc und als Feld argv von Zeigern auf Zeichenketten, welche die Argumente enthalten, zur Verfuegung gestellt. Als Konvention gilt, dass argv[0] die Kommandobezeichnung selbst ist, so dass argc grundsaetzlich groesser als 0 ist.

Das folgende Programm verdeutlicht den Mechanismus. Es gibt einfach seine Argumente auf dem Terminal wieder aus. (Das ist im Prinzip das echo-Kommando.)

```
main(argc, argv)    /* echo arguments */
int argc;
char *argv[];
{
    int i;
    for (i = 1; < argc; i++)
    printf("%s%c", argv[i], (i<argc-1) ? ' ' : '\n');
}
```

argv ist ein Zeiger auf ein Feld, dessen einzelne Elemente Zeiger auf Felder von Zeichen darstellen, jedes Feld ist mit 0 abgeschlossen, so dass es als Zeichenkette behandelt werden kann. Das Programm beginnt mit der Ausgabe von argv[1] und gibt zyklisch ein Argument nach dem anderen aus.

Der Argumentzaehler und die Argumente sind Parameter fuer main. Wenn man diese und andere Routinen weitergeben moechte, muss man sie auf externe Variable kopieren.

1.2. Standardein- und Standardausgabe

Der einfachste Eingabemechanismus besteht darin, die Standardeingabe zu lesen, die im allgemeinen das Nutzerterminal ist. Die Funktion getchar liefert bei jedem Aufruf das naechste Zeichen. Durch Benutzung des Umlenkungsgenerators < kann das Terminal durch eine Datei ersetzt werden; wenn prog die Funktion getchar verwendet, bewirkt die Kommandozeile

```
prog < file
```

dass prog anstelle vom Terminal die Datei file liest. Das Programm prog selbst braucht nicht zu wissen, woher die Eingabe kommt. Das trifft auch dann zu, wenn die Eingabe von einem anderen Programm ueber den Pipe-Mechanismus kommt:

```
otherprog | prog
```

liefert die Standardeingabe fuer prog aus der Standardausgabe von otherprog.

Die Funktion getchar liefert den Wert EOF bei Fileende (oder einem Fehler) unabhaengig davon, was gelesen wird. Der Wert von EOF ist normalerweise -1, aber man sollte von der Kenntnis dieser Tatsache keinen Gebrauch machen. Wie bald deutlich wird, ist dieser Wert automatisch definiert, wenn man ein Programm uerbersetzt, d.h. man muss sich darum nicht kuemmern.

In analoger Weise gibt putchar(c) das Zeichen c auf die "Standardausgabe" aus, die ebenso standardmaessig das Terminal ist. Die Ausgabe kann mittels > auf eine Datei umgeleitet werden: wenn prog die Funktion putchar benutzt, so schreibt

```
prog > outfile
```

die Standardausgabe anstelle auf das Terminal in die Datei outfile. Die Datei outfile wird erzeugt, falls sie nicht existiert; wenn sie bereits existiert, wird der bisherige Inhalt ueberschrieben. Ausserdem kann ein Pipe benutzt werden:

```
prog | otherprog
```

Dabei wird die Standardausgabe von prog als Standareingabe fuer otherprog verwendet.

Die Funktion printf formatiert Ausgaben auf verschiedene Weise und benutzt den gleichen Mechanismus wie auch putchar, so dass Aufrufe von printf und putchar gemischt werden koennen. Dabei erfolgt die Ausgabe in der Reihenfolge der Funktionsaufrufe.

Aehnlich hierzu sorgt scanf fuer eine Formatierung der Eingabe; sie liest die Standardeingabe und spaltet sie in Zeichenketten, Zahlen usw. je nach Wunsch auf. Die Funktion scanf benutzt den gleichen Mechanismus wie getchar, so dass die Funktionsaufrufe auch gemischt auftreten koennen.

Viele Programme lesen nur einen Eingabestrom und liefern nur einen Ausgabestrom; fuer solche Programme kann die E/A mittels getchar, putchar, scanf und printf voellig ausreichend sein. Diese trifft insbesondere zu, wenn der WEGA-Pipemechanismus benutzt wird, um die Ausgabe eines Programmes mit der Eingabe des naechsten zu verbinden. Zum Beispiel blendet das folgende Programm alle ASCII-Steuerzeichen (ausser Newline und Tabulatorzeichen) aus seiner Eingabe aus.

```
    #include <stdio.h>
    main() /* ccstrip: strip non-graphic characters */
    {
        int c;
        while ((c = getchar ()) != EOF)
          if ((c>=' ' && c<0177) || c=='\t' || c=='\n')
              putchar (c);
        exit(0);
     }
```

Die Zeile

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

sollte am Anfang jeder Quelltextdatei auftreten. Dadurch liest der C-Compiler eine Datei (/usr/include/stdio.h) von Standardroutinen und Symbolen einschliesslich der Definition von EOF.

Ist es notwendig, mehrere Dateien zu behandeln, kann man mittels cat die Dateien miteinander verketten:

```
        cat file1 file2 ... | ccstrip > output
```

Dadurch kann vermieden werden, etwas ueber den Zugriff auf Dateien von einem Programm aus zu lernen.

2. Die Standard-E/A-Bibliothek

2.1. Einleitung

Die "Standard-E/A-Bibliothek" ist eine Sammlung von Routinen, die effiziente und portable E/A-Dienste fuer die meisten C-Programme liefern. Die Standard-E/A-Bibliothek ist in jedem System verfuegbar, das C unterstuetzt, so dass Programme, die ihre Kommunikation mit dem System auf die Moeglichkeiten der Standard-E/A-Bibliothek beschraenken leicht von einem System zu einem anderen uebertragbar sind.

In diesem Abschnitt werden die Grundlagen der Standard-E/A-Bibliothek diskutiert. Die Standard-E/A-Bibliothek wurde mit folgender Zielstellung entworfen:

Sie muss so effizient wie moeglich sein, sowohl hinsichtlich Speicherplatzbedarf als auch hinsichtlich Laufzeitverhalten, so dass sie ohne Bedenken auch in sehr kritischen Anwendungsfaellen benutzt werden kann.

Sie muss einfach anzuwenden und ausserdem frei sein von magischen Zahlen und geheimnisvollen ufen, deren Anwendung die Verstaendlichkeit und Uebertragbarkeit vieler Programme herabsetzen, die aeltere Pakete verwenden.

Das bereitgestellte Interface sollte auf allen Maschinen anwendbar sein, unabhaengig davon, ob die Programm, welche die Bibliothek implementieren, direkt auf andere Maschinen oder andere Systeme uebertragbar sind.

2.2. Dateizugriff

Alle bisherigen Programme erledigten ihre E/A ueber die Standardeingabe bzw. Standardausgabe, die wir als vordefiniert annahmen. Der naechste Schritt besteht darin, ein Programm zu schreiben, das auf eine Datei zugreift, welche noch nicht bereits mit dem Programm verbunden ist. Ein simples Beispiel ist wc, das die Zeilen, Worte und Zeichen einer Menge von Dateien zaehlt. Zum Beispiel druckt das Kommando

```
wc x.c y.c
```

die Anzahl der Zeilen, Worte und Zeichen von x.c, y.c sowie die Gesamtsummen aus.

Die Frage ist, was muss getan werden, um die bezeichneten Dateien zu lesen - d.h., wie muessen die Dateisystemnamen mit den E/A-Anweisungen, die letztlich die Daten lesen, verbunden werden.

Die Regeln sind einfach, Bevor eine Datei gelesen oder

geschrieben werden kann, muss sie durch die Standardfunktion fopen eroeffnet werden. Die Funktion fopen nimmt einen externen Namen (z.B. x.c or y.c), stellt eine Verbindung zum Betriebssystem her und gibt einen internen Namen zurueck, der bei nachfolgenden Lese- oder Schreibaktionen fuer die Datei verwendet werden muss.

Dieser interne Name ist eingentlich ein Zeiger, genannt Filepointer, auf eine Struktur, die Informationen ueber die Datei enthaelt, wie z.B. die Adresse eines Puffers, die aktuelle Zeichenposition im Puffer, ob die Datei gelesen oder geschrieben wird usw. Ein Nutzer muss keine Details wissen, da eine Strukturdefinition FILE Bestandteil der in stdio.h eingehaltenen Standard-E/A-Definition ist. Die einzige fuer einen Filepointer notwendige Vereinbarung ist

```
FILE    *fp, *fopen();
```

Das bedeutet, dass fp ein Zeiger auf eine Struktur FILE ist, und fopen einen Zeiger auf eine FILE-Struktur liefert. (FILE ist ein Typname, wie int, und keine Strukturtypbezeichnung.)

Der Aufruf von fopen in einem Programm lautet

```
fp = fopen(name, mode);
```

Das erste Argument von fopen ist der Dateiname als Zeichenkette. Das zweite Argument ist der Modus (als Zeichenkette), der angibt, ob die Datei gelesen ("r"), geschrieben ("w"), oder fortgeschrieben ("a") werden soll.

Wenn eine nicht existierende Datei fuer Schreiben oder Fortschreiben eroeffnet wird, so wird sie - falls moeglich - erzeugt. Das Oeffnen einer bereits existierenden Datei fuer Schreiben bewirkt das Loeschen des urspruenglichen Inhalts. Ein Leseversuch fuer eine nichtexistierende Datei fuehrt zu einem Fehler, ebenso beispielsweise der Versuch, eine Datei zu lesen, ohne dass die Erlaubnis vorliegt. Bei einem Fehler liefert fopen einen Zeigerwert NULL (der in stdio.h als 0 definiert ist).

Als naechstes geht es darum, wie eine eroeffnete Datei gelesen bzw. geschrieben werden kann. Es gibt verschiedene Moeglichkeiten, von denen getc und putc die einfachsten sind. getc liefert das naechste Zeichen einer Datei. Der Zugriff auf die Datei erfolgt ueber den zugehoerigen Filepointer. Somit speichert

```
c = getc(fp)
```

das naechste Zeichen einer Datei, auf die mit fp Bezug genommen wird, nach c. Es wird EOF zurueckgegeben, wenn das Dateiende erreicht wurde. putc ist das Gegenstueck zu getc:

```
        putc(c, fp)
```

gibt das Zeichen c auf die Datei fp aus und liefert c. getc und putc liefern im Fehlerfall EOF.

Bei Beginn der Ausfuehrung eines Programmes sind drei Dateien automatisch eroeffnet und dafuer Filepointer bereitgestellt. Diese Dateien sind die Standardeingabe, die Standardausgabe und die Standardfehlerausgabe; die entsprechenden Filepointer heissen stdin, stdout und stderr. Normalerweise sind sie dem Terminal zugeordnet, koennen aber auf Dateien oder Pipes umgeleitet werden, wie im Abschn. 2.2. beschrieben. stdin, stdout und stderr sind in der E/A-Bibliothek als Datei fuer die Standardeingabe, -ausgabe und -fehlerausgabe vordefiniert. Sie koennen ueberall dort verwendet werden, wo ein Objekt vom Typ FILE * verwendet werden kann. Sie sind jedoch Konstanten, und keine Variablen, so dass man keinesfalls versuchen sollte, ihnen Werte zuzuweisen.

Wir koennen nun wc schreiben. Der Grundentwurf ist fuer viele Programme geeignet: Existieren Argumente der Kommandozeile, so werden sie der Reihe nach verarbeitet. Existieren keine Argumente, so wird die Standardeingabe verarbeitet. Auf diese Weise kann das Programm als selbstaendiges Programm oder als Bestandteil eines groesseren Prozesses verarbeitet werden.

```
#include <stdio.h>
main(argc, argv) /* wc: count lines, words, chars */
int argc;
char *argv[];
{
   int c, i, inword;
   FILE *fp, *fopen();
   int linect, wordct, charct;
   int tlinect = 0, twordct = 0, tcharct = 0;
   i = 1;
   fp = stdin;
   do {
        if (argc>1 && (fp=fopen(argv[i], "r")) == NULL) {
          fprintf(stderr, "wc: can't open %s\n", argv[i]);
          continue;
        }
        linect = wordct = charct = inword = 0;
        while ((c = getc(fp)) != EOF) {
          charct++;
          if (c == '\n') linect++;
          if (c==' ' || c=='\t' || c=='\n') inword = 0;
          else if (inword == 0) {
                  inword = 1;
                  wordct++;
                }
        }
        printf("%7ld %7ld %7ld", linect, wordct, charct);
        printf(argc > 1 ? " %s\n" : "\n", argv[i];
```

```
        fclose(fp);
        tlinect += linect;
        twordct += wordct;
        tcharct += charct;
      } while (++i <argc);
 if (argc > 2)
 printf("%7ld %7ld %7ld total\n",tlinect,twordct,tcharct);
 exit (0);
}
```

Die Funktion fprintf ist zu printf identisch, abgesehen davon, dass das erste Argument ein Filepointer ist, der angibt, in welche Datei geschrieben werden soll.

Die Funktion fclose ist das Gegenstueck zu fopen; sie loest die urspruenglich von fopen aufgestellte Verbindung zwischen dem Filepointer und dem externen Namen, indem sie den Filepointer fuer eine andere Datei freigibt. Da die Anzahl der von einem Programm gleichzeitig eroeffneten Dateien begrenzt ist, sollten nicht mehr benoetigte Dateien freigegeben werden. Der andere Grund fuer den Aufruf von fclose fuer eine Ausgabedatei besteht darin, dass dadurch der Puffer geleert wird. (fclose wird fuer jede eroeffnete Datei automatisch aufgerufen, wenn ein Programm normal endet.)

2.3. Fehlerbehandlung

stderr ist einem Programm in der gleichen Weise wie stdin und stdout zugewiesen. Auf stderr ausgegebene Daten erscheinen auf dem Terminal selbst dann, wenn die Standardausgabe umgelenkt wurde. Das Programm wc gibt Fehlernachrichten auf stderr aus anstatt auf stdout. Kann auf eine der Dateien aus irgendeinem Grund nicht zugegriffen werden, so erscheint diese Nachricht auf dem Terminal und verschwindet nicht in einer Pipeline oder in einer Ausgabedatei.

Das Programm zeigt Fehler auf eine andere Weise an, indem es die Funktion exit benutzt, um die Programmausfuehrung zu beenden. Das Argument von exit wird dem aufrufenden Programm grundsaetzlich zur Verfuegung gestellt (vgl. Abschn. 6), so dass von einem anderen Programm, das dieses als Unterprozess benutzte, getestet werden kann, ob es erfolgreich oder fehlerhaft ausgefuehrt wurde. Ein Rueckgabewert 0 gilt per Konvention als erfolgreiche Verarbeitung, waehrend ein Wert ungleich 0 eine Fehlerbedingung anzeigt.

Die Funktion exit selbst ruft fclose fuer jede eroeffnete Ausgabedatei auf, um eventuell noch gefuellte Puffer zu leeren. Anschliessend wird die Routine exit aufgerufen, die eine sofortige Beendigung ohne Leeren von Puffern bewirkt. Die Funktion exit kann ggf. direkt aufgerufen werden.

2.4. Verschiedene E/A-Funktionen

Die Standard-E/A-Bibliothek enthaelt neben den bereits erwaehnten verschiedene andere E/A-Funktionen.

Normale Ausgaben mit putc werden (bis auf stderr) gepuffert; um sie sofort auszugeben, kann fflush(fp) benutzt werden.

fsanf ist identisch zu scanf abgesehen davon, dass als erstes Argument ein Filepointer stehen muss (wie bei fprintf), der angibt, von welcher Datei die Eingabe erfolgen soll. Bei Dateiende wird EOF geliefert.

Die Funktionen sscanf und sprintf sind identisch zu fscanf und fprintf abgesehen davon, dass als erstes Argument eine Zeichenkette anstelle eines Filepointers stehen muss. Die Uebertragung erfolgt bei sscanf aus der und bei sprintf in die Zeichenkette.

fgets(buf, size, fp) kopiert die naechste Zeile von fp bis einschliesslich einem Newlinezeichen in buf; es werden maximal size-1 Zeichen kopiert, bei Dateiende wird NULL zurueckgegeben. fputs(buf,fp) gibt die in buf befindlichen Zeichen in die Datei fp aus.

Die Funktion ungetc(c, fp) bewirkt das "Zurueckschreiben" des Zeichens c in die Eingabedatei fp, ein darauffolgender Aufruf von getc, fscanf usw. liefert c. Pro Datei kann nur ein Zeichen zurueckgeschrieben werden.

2.5. Allgemeine Anwendung

Jedes Programm, das die Routinen der Standard-E/A-Bibliothek nutzen will, muss die Zeile

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

enthalten, die bestimmte Makros und Variablen definiert. Die Routinen befinden sich in der normalen C-Bibliothek, so dass kein spezielles Bibliotheksargument fuer den Ladevorgang notwendig ist. Alle Bereiche in den include-Files, die nur fuer den internen Gebrauch bestimmt sind, beginnen mit einem Unterstreichungszeichen (_), um die Moeglichkeit der Kollision mit einem vom Nutzer definierten Bezeichner zu reduzieren. Die nach aussen hin sichtbaren Bezeichner sind:

stdin Der Name der Standardeingabedatei

stdout Der Name der Standardausgabedatei

stderr Der Name der Standardfehlernachrichtendatei

EOF Der Wert, der von den Leseroutinen bei Dateiende oder einem Fehler geliefert wird: -1

NULL Eine Notation fuer den Null-Pointer; dieser Wert wird von den Funktionen, die einen Zeigerwert liefern, im Fehlerfall zurueckgegeben.

FILE Bezeichnet den Strukturtyp _iob und kann zur Deklaration von Zeigern auf Datenstroeme benutzt werden.

BUFSIZE Ist die Anzahl der Bytes fuer einen vom Nutzer bereitgestellten E/A-Puffer, siehe setbuf.

getc, getchar, putc, putchar
feof, ferror, fileno
Sind als Makros definiert. Ihre Wirkung wird spaeter beschrieben. Sie sind hier erwaehnt, um klarzumachen, dass sie nicht neu deklariert werden koennen und dass sie wirklich keine Funktionen sind. Man kann somit beispielsweise keine Unterbrechungspunkte bzgl. ihrer Namen setzen.

Die Routinen dieses Paketes bieten die Moeglichkeit der automatischen Pufferzuordnung und der automatischen Pufferleerung bei Ausgabe. Die Bezeichner stdin, stdout und stderr sind als Konstanten zu betrachten, d.h. ihnen kann kein Wert zugewiesen werden.

2.6. Funktionsaufrufe

FILE *fopen(filename, type) char *filename, *type;
Eroeffnet die Datei und ordnet bei Bedarf einen Puffer zu. Die Zeichenkette filename gibt den Namen der Datei an. Type ist ebenfalls eine Zeichenkette (kein einzelnes Zeichen); sie kann die Werte "r", "w" oder "a" annehmen, d.h. read, write oder append. Die Funktion liefert als Ergebnis einen Filepointer. Ist dieser Wert NULL, konnte die Datei nicht eroeffnet werden.

FILE *freopen(filename, type, ioptr) char *filename, *type;
FILE *ioptr;
Die durch ioptr bezeichnete Datei wird geschlossen und falls notwendig, wie bei fopen wieder eroeffnet. Geht das Eroeffnen schief, so liefert die Funktion den Wert NULL, ansonsten ioptr, der jetzt auf die neue Datei zeigt. Oft handelt es sich bei der wiedereroeffneten Datei um stdin oder stdout.

int getc(ioptr) FILE *ioptr;
liefert das naechste Zeichen der Datei, auf das

sich ioptr bezieht. Der Zeiger ioptr wurde vorher z.B. von fopen bereitgestellt, oder es ist stdin. Der Integerwert EOF wird bei Dateiende oder im Fehlerfall geliefert. Das Zeichen 0 ist ein legales Zeichen.

int fgetc(ioptr) FILE *ioptr;
Die Wirkung ist analog zu getc, jedoch handelt es sich um eine echte Funktion und nicht um einen Makro. Somit kann ein Zeiger auf fgetc definiert werden, die Funktion kann als Funktionsargument uebergeben werden usw.

putc(c, ioptr) char c; FILE *ioptr;
Schreibt das Zeichen c in die durch ioptr angegebene Datei. Den Wert von ioptr hat man z.B. von fopen erhalten oder es handelt sich um stdout bzw. stderr. putc liefert als Ergebnis den Wert des Zeichens c oder im Fehlerfall EOF.

fputc(c, ioptr) char c; FILE *ioptr;
Analog zu putc, nur dass es sich um eine Funktion handelt.

fclose(ioptr) FILE *ioptr;
Die zu ioptr korrespondierende Datei wird geschlossen, nachdem alle Puffer geleert wurden. Ein durch das E/A- System zugeordneter Puffer wird freigegeben. Die Funktion fclose wird bei normaler Beendigung des Programmes automatisch ausgefuehrt.

fflush(ioptr) FILE *ioptr;
Alle gepufferten Informationen der durch ioptr bezeichneten Ausgabedatei werden ausgegeben. Ausgabedateien werden normalerweise genau dann gepuffert, wenn sie nicht auf das Terminal gelenkt werden. stderr ist jedoch immer ungepuffert, wenn nicht setbuf benutzt wird oder stderr mittels setbuf wieder eroeffnet wurde.

exit(errcode);
Beendet den Prozess und gibt sein Argument als Status an den Elternprozess. Dies ist eine spezielle Version der Routine, die fflush fuer jede Ausgabedatei ruft. Sollen die Ausgabepuffer nicht geleert werden, muss _exit benutzt werden.

feof(ioptr) FILE *ioptr;
Liefert einen Wert ungleich Null, wenn fuer diese Eingabedatei die Dateiendebedingung vorlag.

ferror(ioptr) FILE *ioptr;
Liefert einen Wert ungleich Null, wenn waehrend des Lesens oder Schreibens der Datei ein Fehler auftrat. Die Fehleranzeige bleibt bis zum Schliessen der Datei bestehen.

getchar();
Ist identisch zu getc(stdin).

putchar(c) char c;
Ist identisch zu putc(c, stdout).

char *fgets(s, n, ioptr) char *s; int n; FILE *ioptr;
Liest bis zu n-1 Zeichen von der durch ioptr bezeichneten Datei in das durch s adressierte Zeichenfeld. Das Lesen wird bei Erkennen des Newline-Zeichens ('\n') beendet. Das Zeichen '\n' wird in den Puffer abgelegt und das Zeichen '\0' angefuegt. fgets liefert das erste Argument oder NULL, falls ein Fehler oder ein Dateiende aufgetreten ist.

fputs(s, ioptr) char *s; FILE *ioptr;
Schreibt die durch '\0' abgeschlossene Zeichenkette s in die durch ioptr bezeichnete Datei. Es wird kein Newline-Zeichen angefuegt. fputs liefert keinen Wert.

ungetc(c, ioptr) char c; FILE *ioptr;
Das Zeichen c wird in die durch ioptr bezeichnete Eingabedatei zurueckgespeichert. (Der naechste getc-Aufruf liefert dieses Zeichen.) Es kann nur ein Zeichen zurueckgespeichert werden.

printf(format, a1, ...) char *format;
fprintf(ioptr, format, a1, ...) FILE *ioptr; char *format;
sprintf(s, format, a1, ...) char *s, *format;
printf schreibt auf die Standardausgabe. fprintf schreibt in die angegebene Ausgabedatei. sprintf gibt Zeichen in das Zeichenfeld s aus. Die Formatspezifikationen dieser Funktion sind im Abschnitt printf(3) des WEGA-Programmierhandbuches, Sektion 3, beschrieben.

scanf(format, a1, ...) char *format;
fscanf(ioptr, format, a1, ...) FILE *ioptr; char *format;
sscanf(s, format, a1, ...) char *s, *format;
scanf liest von der Standardeingabe. fscanf liest von den angegebenen Dateien. sscanf liest von Zeichenfeld s. scanf liest Zeichen, interpretiert sie entsprechend der Formatspezifikationen und speichert die Ergebnisse in die angegebenen Argumente. Jede dieser Funktionen erwartet als Argumente eine Zeichenkette format (die Formatspezifikationen erhaelt) sowie abhaengig von der Anzahl der Formatspezifikationen eine Reihe von Zeigern die angeben, wohin die konvertierte Eingabe gespeichert werden soll.

scanf liefert als Funktionswert die Anzahl der erfolgreich durchgefuehrten Konvertierungen und Eingabezuordnungen. Dies kann verwendet werden, um

festzustellen, wieviele Eingabewerte gefunden worden sind. Bei Dateiende wird EOF zurueckgegeben. Man beachte, dass EOF ungleich 0 ist. Der Wert 0 bedeutet, dass das naechste Eingabezeichen nicht auf die entsprechende Formatspezifikation passte, oder anders ausgedrueckt, keine erfolgreiche Konvertierung durchgefuehrt werden konnte.

fread(ptr, sizeof(*ptr), nitems, ioptr) char *ptr;
int nitems; FILE *ioptr;
Liest nitems Dateneinheiten des angegebenen Typs, beginnend ab ptr in der Datei ioptr.

fwrite(ptr, sizeof(*ptr), nitems, ioptr) char *ptr;
int nitems; FILE *ioptr;
analog zu fread, nur genau umgekehrt.

rewind(ioptr) FILE *ioptr;
Setzt auf den Anfang der durch ioptr bezeichneten Datei zurueck. rewind sollte nur fuer Eingabedateien benutzt werden, da eine Ausgabedatei nach einer solchen Operation immer noch fuer Schreiben geoeffnet waere.

system(string) char *string;
Das Kommando string wird durch Shell ausgefuehrt, als ob es sich um eine am Terminal eingegebene Kommandozeile handeln wuerde.

getw(ioptr) FILE *ioptr;
Liefert das naechste Wort von der durch ioptr bezeichneten Eingabedatei. Bei Dateiende oder im Fehlerfall wird EOF zurueckgegeben. Da EOF jedoch ein gueltiger Integerwert ist, sollte feof und ferror benutzt werden.

putw(w, ioptr) FILE *ioptr;
Schreibt das Wort w in die durch ioptr bezeichnete Ausgabedatei.

setbuf(ioptr, buf) FILE *ioptr; char *buf;
setbuf muss nach dem Eroeffnen oder vor dem Beginn der E/A- benutzt werden. Wenn buf gleich NULL, so erfolgt die E/A ungepuffert. Ansonsten wird der durch buf angegebene Puffer benutzt. Dabei muss es sich um ein Zeichenfeld mit hinreichender Laenge handeln.

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

fileno(ioptr) FILE *ioptr;
Liefert den mit der durch ioptr bezeichneten Datei verbundenen Filedeskriptor.

fseek(ioptr, offset, ptrname) FILE *ioptr; long offset;
int ptrname;

Es wird die Position des naechsten Bytes in der Datei eingestellt. Wenn ptrname 0 ist, wird die Verschiebung (Offset) bzgl. Dateianfang gemessen. Falls ptrname 1 ist, wird Offset bzgl. der aktuellen Position gemessen. Wenn ptrname den Wert 2 besitzt, wird Offset bzgl. Dateiende gemessen. Die Routine beruecksichtigt eventuelle Pufferung.

long ftell(ioptr) FILE *ioptr;
Die Verschiebung bzgl. Dateianfang wird geliefert. Eventuelle Pufferung wird beruecksichtigt.

getpw(uid, buf) int uid; char *buf;
Die Passwortdatei (passwd) wird nach der angegebenen UID-Nummer durchsucht. Falls ein entsprechender Eintrag vorhanden ist, so wird 0 zurueckgegeben und die dazugehoerige Zeile wird im Zeichenpuffer buf abgelegt. Falls kein Eintrag fuer die UID-Nummer vorhanden ist, so wird 1 zurueckgegeben.

char *malloc(num) int num;
Stellt num Bytes bereit. Das Ergebnis ist ein Zeiger, der den Forderungen nach Erhaltung integraler Grenzen genuegt. Ist kein Speicherplatz vorhanden, wird NULL zurueckgegeben.

char *calloc(num, size) int num, size;
Liefert Speicherplatz der Groesse num * size, wobei eine Initialisierung auf 0 erfolgt. Das Ergebnis ist ein Zeiger, der den Anforderungen integraler Grenzen genuegt. Ist kein Speicherplatz vorhanden, wird NULL zurueckgegeben.

cfree(ptr) char *ptr;
Der durch calloc bereitgestellte Speicherplatz wird freigegeben. Probleme treten moeglicherweise auf, wenn der Zeiger ptr nicht durch calloc bereitgestellt wurde.

2.7. Makros

Es folgen Makros, deren Definitionen in der Datei <ctype.h> enthalten sind:

isalpha(c) liefert einen Wert ungleich 0, wenn c ein Alphazeichen ist.

isupper(c) liefert einen Wert ungleich 0, falls c ein Grossbuchstabe ist.

islower(c) liefert einen Wert ungleich 0, falls c ein Kleinbuchstabe ist.

isdigit(c) liefert einen Wert ungleich 0, falls eine Ziffer

c ist.

isspace(c) liefert einen Wert ungleich 0, falls c eines der Zeichen ' ', '\t', '\n', '\r', '\v', '\f' ist.

ispunct(c) liefert einen Wert ungleich 0, falls c kein Steuerzeichen, Leerzeichen, Buchstabe oder Ziffer ist (d.h., wenn c ein Interpunktionszeichen ist.)

isalnum(c) liefert einen Wert ungleich 0, falls c ein Buchstabe oder eine Ziffer ist.

isprint(c) liefert einen Wert ungleich 0, falls c druckbar ist, d.h. ein Buchstabe, Ziffer oder Interpunktionszeichen ist.

iscntrl(c) liefert einen Wert ungleich 0, falls c ein Steuerzeichen ist.

isascii(c) liefert einen Wert ungleich 0, falls c ein ASCII-Zeichen, d.h. kleiner als der Oktalwert 0200 ist.

toupper(c) liefert den zum Kleinbuchstaben c gehoerigen Grossbuchstaben.

tolower(c) liefert den zum Grossbuchstaben c gehoerigen Kleinbuchstaben.

3. E/A Auf niedrigster Stufe

3.1. Allgemeines

Dieser Abschnitt erlaeutert die unterste Stufe der E/A im System WEGA. Die Stufe gewaehrleistet weder eine Pufferung noch irgendwelche anderen Dienste, sie stellt in der Tat einen direkten Zugang in das Betriebssystem her. Man ist vollstaendig auf sich allein gestellt, hat jedoch andererseits die beste Kontrolle ueber alle Vorgaenge. Und da die Rufe und die Benutzung ganz einfach sind, ist die Angelegenheit gar nicht so schlecht, wie sie auf den ersten Blick erscheint.

3.2. Filedeskriptoren

Im WEGA erfolgt eine Eingabe oder Ausgabe- grundsaetzlich, indem Dateien gelesen oder geschrieben werden, da alle peripheren Geraete sogar Nutzerterminals, Dateien im Dateisystem sind. Das bedeutet, dass jede Kommunikation zwischen einem Programm und peripheren Geraeten durch eine einzige, homogene Schnittstelle behandelt wird.

Im allgemeinen Fall ist es vor dem Lesen oder Schreiben einer Datei noetig, das System von dieser Absicht zu informieren. Dieser Prozess heisst "Oeffnen" der Datei. Soll eine Datei geschrieben werden, so kann es auch noetig sein, sie vorher zu erzeugen. Das System prueft, ob der Nutzer das Recht hierzu besitzt (Existiert die Datei? Hat der Nutzer die Zugriffserlaubnis?). Ist alles in Ordnung, stellt das System eine positive Integerzahl, einen sogenannten Filedeskriptor bereit. Immer dann, wenn eine E/A-Operation durchgefuehrt werden soll, wird der Filedeskriptor anstelle des Dateinamens benutzt, um die betreffende Datei zu identifizieren. (Das ist annaehernd so wie die Verwendung von READ(S,...) und WRITE(6,...) in Fortran.) Alle Informationen ueber eine eroeffnete Datei werden vom System verwaltet; Das Nutzerprogramm bezieht sich auf die Datei grundsaetzlich ueber den Filedeskriptor.

Die in Abschn. 3 eroerterten Filepointer sind im Prinzip aehnlich zu den Filedeskriptoren, die andererseits grundlegender sind. Ein Filepointer ist ein Zeiger auf eine Struktur, die neben anderen Dingen auch den Filedeskriptor fuer die betreffende Datei enthaelt.

Da die Ein- und Ausgabe normalerweise ueber das Terminal eines Nutzers erfolgt, gibt es spezielle Vereinbarungen, um das auf geeignete Weise zu unterstuetzen. Wenn der Kommandointerpreter ("shell") ein Programm abarbeitet, so eroeffnet er drei Dateien mit den Deskriptoren 0, 1 und 2. Diese Dateien heissen Standardeingabe, Standardausgabe und Standardfehlerausgabe und sind normalerweise dem Terminal zugeordnet. Damit kann ein Programm, indem es vom Filedeskriptor 0 liest und in die Filedeskriptoren 1 und 2

schreibt, Terminal-E/A erledigen, ohne sich um das Eroeffnen von Dateien zu kuemmern.

Wenn die E/A mittels < und > umgelenkt wird, wie in

```
prog <infile >outfile
```

so aendert die Shell die Standardzuordnungen fuer die Filedeskriptoren 0 und 1 von dem Terminal auf die betreffenden Files. Aehnliches gilt auch dann, wenn die Eingabe oder Ausgabe mit einer Pipe verbunden ist. Normalerweise bleibt Filedeskriptor 2 mit dem Terminal verbunden, so dass dort die Fehlernachrichten erscheinen. In jedem Fall wird die Dateizuweisung durch Shell geaendert, nicht durch das Programm. Das Programm muss nicht wissen, woher die Eingabe kommt oder wohin die Ausgabe geht, solange Datei 0 fuer die Eingabe bzw. Datei 1 und 2 fuer die Ausgabe benutzt werden .

3.3. Read und Write

Jede Ein- bzw. Ausgabe wird durch die Funktionen read bzw. write durchgefuehrt. Fuer beide Funktionen ist das erste Argument ein Filedeskriptor. Das zweite Argument ist ein Puffer in dem betreffenden Programm, wo die Daten herkommen oder wohin sie gelangen sollen. Das dritte Argument ist die Anzahl der Bytes, die uebertragen werden sollen. Die Aufrufe der Funktion heissen:

```
n_read = read(fd, buf, n);
n_written = write(fd, buf, n);
```

Zurueckgegeben wird jeweils ein Bytezaehler, der die Anzahl der tatsaechlich uebertragenen Bytes angibt. Beim Lesen kann dieser Zaehler kleiner sein als die als Argument spezifizierte Byteanzahl, wenn weniger als n Bytes zum Lesen uebrig bleiben. (Ist die Datei das Terminal, liest read normalerweise nur bis zum naechsten Newline, das ist im allgemeinen weniger als gefordert wurde.) Ein Rueckgabewert von 0 Bytes impliziert das Dateiende und -1 zeigt irgendeinen Fehler an. Beim Schreiben gibt der Rueckgabewert die Anzahl der tatsaechlich geschriebenen Bytes an, es liegt im allgmeinen ein Fehler vor, wenn dieser Wert nicht mit der im Argument angegebenen Zahl uebereinstimmt.

Es gibt keinerlei Festlegungen ueber die Byteanzahl, die angegeben werden kann. Ueblich sind die Werte 1, d.h. ein Zeichen pro Zeiteinheit ("ungepuffert") und 512, dieser Wert entspricht der physischen Blockgroesse vieler peripherer Geraete. Die letztere Anzahl ist am effektivsten, aber auch die zeichenweise E/A ist nicht ausgesprochen aufwendig.

Mit diesen Mitteln koennen wir ein einfaches Programm

schreiben, was die eingegebenen Daten einfach wieder ausgibt. Dieses Programm kopiert irgendetwas irgendwohin, da die Eingabe und Ausgabe auf eine beliebige Datei oder Geraet umgelenkt werden kann.

```
#define BUFSIZE 512 /* best size for WEGA */
main()  /* copy input to output */
{
   char buf[BUFSIZE];
   int  n;
   while ((n = read (0, buf, BUFSIZE)) > 0)
           write(1, buf, n);
   exit(0);
}
```

Ist die Dateigroesse kein vielfaches von BUFSIZE, so wird irgendein read einen kleineren Wert fuer die Anzahl der tatsaechlich gelesenen und durch write zu schreibenden Bytes liefern. Der naechste Aufruf von read wird den Wert 0 liefern.

Es ist aufschlussreich zu sehen, wie read und write benutzt werden koennen, um Routinen einer hoeheren Niveaustufe wie getchar, putchar usw. aufzubauen. Als Beispiel folgt eine Version von getchar fuer ungepufferte Eingabe:

```
getchar() /* unbuffered single character input */
{
   char c;
   return((read(0, &c, 1) > 0) ? c : EOF);
}
```

c muss als Zeichen char vereinbart werden, da read einen Zeichenzeiger akzeptiert.

Die zweite Version von getchar liest in groesseren Einheiten ein und verteilt ein Zeichen nach dem anderen.

```
#define BUFSIZE 512
#define CMASK 0377
getchar() /* buffered version */
{
   static char  buf[BUFSIZE];
   static char  *bufp = buf;
   static int   n = 0;
   if (n == 0) {  /* buffer is empty */
      n = read(0, buf, BUFSIZE);
      bufp = buf;
   }
   return ((--n >= 0) ? *bufp++ & CMASK : EOF);
}
```

3.4. Open, Creat, Close, Unlink

Soll die E/A nicht ueber Standardeingabe-, Standardausgabe- oder Standardfehlernachrichten erfolgen, so muessen die Dateien explizit fuer Schreiben oder Lesen eroeffnet werden. Hierfuer gibt es zwei Eintrittspunkte in System, open und creat.

open ist zu fopen ziemlich aehnlich, abgesehen davon, dass anstelle eines Filepointers ein Filedeskriptor zurueckgeschrieben wird, der eben ein Integerwert ist.

```
int fd;
fd = open(name, rwmode);
```

Wie bei fopen ist das name-Argument eine Zeichenkette, die mit dem externen Dateinamen korrespondiert. Das Argument fuer den Zugriffsmodus jedoch ist verschieden, naemlich 0 fuer Lesen, 1 fuer Schreiben und 2 fuer Lese- und Schreibzugriff. open liefert -1 im Fehlerfall, ansonsten einen gueltigen Filedeskriptor.

Es ist fehlerhaft, eine noch nicht existierende Datei mittels open eroeffnen zu wollen, hierfuer gibt es den Eintrittspunkt creat, der auch verwendet werden kann, um existierende Dateien neu zu ueberschreiben.

```
fd = creat(name, pmode);
```

liefert einen Filedeskriptor, wenn es moeglich war, die Datei name einzurichten, ansonsten -1. Existierte die Datei bereits, so wird die Dateigroesse mit 0 festgelegt; es ist kein Fehler, eine Datei zu erzeugen, die bereits existierte.

Eine neue Datei wird mit dem Schutzmodus pmode eingerichtet. Im WEGA-Filesystem gibt es fuer jede Datei neun Bits Schutzinformationen. In ihnen sind verschluesselt: Lese-, Schreib- und Ausfuehrungserlaubnis fuer den Eingentuemer der Datei fuer die Nutzergruppe sowie fuer alle restlichen Nutzer. Somit reicht eine dreistellige Oktalzahl vollkommen aus, um die Erlaubnisrechte zu spezifizieren. Zum Beispiel legt 0755 die Lese-, Schreib- und Ausfuehrungserlaubnis fuer die Gruppe und alle anderen Nutzer fest.

Um das zu verdeutlichen, folgt nun eine vereinfachte Version des WEGA-Hilfsprogramms cp. Dieses Programm kopiert eine Datei in eine andere. (Die wesentlichste Vereinfachung besteht darin, dass diese Version nur eine Datei kopiert und das zweite Argument kein Verzeichnis sein darf.)

```
#define NULL 0
#define BUFSIZE 512
#define PMODE 0644 /* RW for owner, R for group, others */
```

```
main(argc, argv)   /* cp: copy f1 to f2 */
int argc;
char *argv[];
{
        int f1, f2, n;
        char buf[BUFSIZE];

        if (argc != 3)
                error("Usage: cp from to", NULL);
        if ((f1 = open(argv[1], 0)) == -1)
                error("cp: can t open %s", argv[1]);
        if ((f2 = creat(argv[2], PMODE)) == -1)
                error("cp: can't create %s", argv[2]);

        while ((n = read(f1, buf, BUFSIZE)) > 0)
                if (write(f2, buf, n) != n)
                  error("cp: write error", NULL);
        exit(0);
}

error(s1, s2)           /* print error message and die */
char *s1, *s2;
{
   printf(s1, s2);
   printf("\n");
   exit(1);
}
```

Wie bereits erwaehnt, gibt es einen Grenzwert (typischerweise 15-25) fuer die Anzahl der gleichzeitig offenen Dateien eines Programmes. Dementsprechend muss ein Programm, das viele Dateien verarbeiten soll, darauf vorbereitet sein, Filedeskriptoren wiederzuverwenden. Die Routine close loest die Verbindung zwischen einem Filedeskriptor und einer offenen Datei und gibt den Filedeskriptor zur Verwendung mit einer anderen Datei frei. Die Beendigung eines Programms ueber exit oder die Rueckkehr vom Hauptprogramm "schliesst" alle offenen Dateien. Die Funktion unlink (filename) entfernt die Datei filename aus dem Filesystem.

3.5. Direktzugriff mit lseek

Normalerweise erfolgt die E/A von Dateien sequentiell: jedes read oder write bezieht sich auf die darauffolgende Stelle in der Datei, auf die sich die letzte E/A-Operation bezog. Wenn es jedoch notwendig ist, so kann eine Datei in beliebiger Reihenfolge gelesen oder geschrieben werden. Der Systemruf seek bietet eine Moeglichkeit, in der Datei irgendwohin zu positionieren ohne echt zu lesen oder zu schreiben:

```
        lseek(fd, offset, origin);
```

bewirkt, dass in der durch fd festgelegten Datei eine

Positionierung auf offset erfolgt, wobei offset als Verschiebung zu origin interpretiert wird. Eine nachfolgende Lese- oder Schreiboperation beginnt bei dieser Position. offset ist ein long-int Wert, fd und origin sind int-Werte. Dabei kann origin die Werte 0, 1 oder 2 besitzen, je nachdem ob offset vom Beginn der aktuellen Position oder vom Ende der Datei an gerechnet werden soll. Um zum Beispiel eine Datei fortzuschreiben, muss zunaechst an das Dateiende positioniert werden:

```
        lseek(fd, 0L, 2);
```

Um an den Dateibeginn zu positionieren (rewind):

```
        lseek(fd, 0L, 0);
```

Mittels lseek ist es moeglich, Dateien mehr oder weniger wie grosse Felder zu behandeln, allerdings auf Kosten eines langsameren Zugriffs. Folgende einfache Funktion liest z.B. eine beliebige Anzahl von Bytes ab einer beliebigen Position in einer Datei:

```
get(fd, pos, buf, n) /* read n bytes from position pos */
int fd, n;
long pos;
char *buf;
{
   lseek(fd, pos, 0);   /* get to pos */
   return(read(fd, buf, n));
}
```

3.6. Fehlerbehandlung

Die in diesem Abschnitt erlaeuterten Routinen, tatsaechlich jedoch alle Routinen, die einen direkten Eintrittspunkt im System haben, koennen Fehler zur Folge haben. Normalerweise zeigen sie einen Fehler an, indem sie -1 liefern. Manchmal ist es allerdings wuenschenswert, wenn man etwas genaueres ueber die Fehlerursache erfaehrt, deshalb hinterlassen all diese Routinen eine Fehlernummer in der externen Variablen errno. Die Bedeutung der verschiedenen Fehler ist in Sektion 2 des WEGA-Programmierhandbuchs aufgelistet, so dass ein Programm z.B. feststellen kann, ob ein Versuch des Eroeffnens einer Datei scheiterte, weil sie nicht existierte, oder weil keine Zugriffserlaubnis vorlag. Normalerweise wuenscht man, die Fehlerursache auszudrucken. Die Routine perror druckt eine mit dem Wert von errno in Beziehung stehende Nachricht aus; noch allgemeiner sys_errno ist ein Feld von Zeichenketten, die ueber errno indiziert und ausgedruckt werden koennen.

4. Prozesse

Es ist oft leichter ein Programm zu benutzen, das jemand geschrieben hat, als selbst eines zu erfinden. Dieser Abschnitt beschreibt, wie man ein Programm von einem anderen Programm aus ausfuehren kann.

4.1. Die Systemfunktionen

Die einfachste Moeglichkeit, ein Programm aus einem anderen heraus auszufuehren besteht darin, die Standardbibliotheksroutine system zu verwenden. system besitzt ein Argument, eine Kommandozeichenkette in der gleichen Form wie man sie am Terminal eintippt (abgesehen vom Newlinezeichen am Ende.) Das entsprechende Kommando wird abgearbeitet.

```
main()
{
   system("date");
   /* rest of processing */
}
```

Wenn die Kommandozeichenkette aus mehreren Stuecken aufgebaut werden muss, koennten die Formatierungsmoeglichkeiten der Funktion sprintf nuetzlich sein.

Es sei daran erinnert, dass getc und putc normalerweise ihre E/A puffern, wird die Terminal-E/A nicht korrekt synchronisiert, wenn nicht die Pufferung verhindert wird. Fuer die Ausgabe kann fflush fuer die Eingabe setbuf verwendet werden.

4.2. Prozesserzeugung auf unterster Ebene

Wenn man nicht die Standardbibliothek benutzt, oder wenn man eine differenzierte Steuerung ueber das Geschehen benoetigt, muss man die Rufe anderer Programme mittels primitiverer Routinen als system durchfuehren.

Die Basisoperation zur Ausfuehrung eines anderen Programmes ohne Rueckkehrabsicht heisst execl. Um das Datum wie im obigen Beispiel als letzte Aktion eines Programmes auszudrucken, kann man

```
execl("/bin/date", "date", NULL);
```

benutzen. Das erste Argument von execl ist der Dateiname des Kommandos. Man muss also wissen, wo sich das Kommando im Dateisystem befindet. Das zweite Argument ist per Konvention der Programmname (d.h. die letzte Komponente des Dateinamens); aber das wird selten benutzt und dient meist nur als Platzhalter. Besitzt das Kommando Argumente, so

sind sie anschliessend angefuehrt. Das Ende der Liste wird durch das NULL-Argument markiert.

Der execl-Aufruf ueberlagert das existierende programm durch das neue, fuehrt es aus und beendet es. Es gibt keinen Ruecksprung in das urspruengliche Programm.

Ein etwas realistischerer Fall liegt vor, wenn ein Programm in zwei oder mehrere Phasen zerfaellt, die lediglich ueber temporaere Dateien miteinander kommunizieren. Hierbei ist es ganz natuerlich, wenn der zweite Pass mittels execl vom ersten gerufen wird. Die Ausnahme der Regel, dass das aufrufende Programm niemals die Steuerung zurueckerhaelt, besteht dann, wenn ein Fehler auftrat. Z.B. wenn die Datei nicht gefunden werden konnte oder sie nicht ausfuehrbar ist. Wenn man nicht genau weiss, wo sich date befindet, kann man schreiben

```
execl("/bin/date"/, "date", NULL);
execl("/usr/bin/date", "date", NULL);
fprintf(stderr, "Someone stole 'date'\n");
```

Es gibt eine Variante von execl die execv heisst und dann sinnvoll ist, wenn man vorher nicht genau weiss, wie viele Argumente es sein werden. Der Aufruf lautet:

```
execv(filename, argp);
```

wobei argp ein Feld von Zeigern auf die Argumente ist. Der letzte Zeiger muss NULL sein, so dass execv erkennt, wo die Liste endet. Wie bei execl ist filename die Datei, wo sich das Programm befindet und argp[0] ist der Programmname. (Diese Vereinbarung ist zu dem argv-Feld fuer Programmargumente identisch.)

Wenn man nicht weiss, wo ein Programm zu finden ist oder sich darum nicht kuemmert, kann man die Routinen execlp und execvp verwenden, um es zu finden. Die Funktionen execlp und execvp werden mit den gleichen Argumenten aufgerufen wie execl und execv, jedoch wird in einer Liste von Verzeichnissen nach einer ausfuehrbaren Datei gesucht. Wenn die gefundene ausfuehrbare Datei ein Shellskript ist, wird Shell aufgerufen, um sie auszufuehren. Die Verzeichnisliste wird dem Environment entnommen. Keine dieser Routinen bietet den Komfort der normalen Kommandoabarbeitung. Es wird keine automatische Suche in mehreren Verzeichnissen durchgefuehrt - man muss genau wissen, wo sich das Kommando befindet. Es gibt auch keine Verarbeitung der Metazeichen <, >, *, ? und [] in der Argumentliste. Wenn man diese wuenscht, muss man execl benutzen, um Shell sh, aufzurufen, das dann alles weitere erledigt. Man baue hierzu commandline als Zeichenkette auf, die das komplette Kommando genauso enthaelt, als waere es am Terminal eingegeben worden. z.B.

```
execl ("/bin/sh", "sh", "-c", commandline, NULL);
```

Es wird hierbei angenommen, dass sich Shell an einem bestimmten Platz, naemlich /bin/sh befindet. Das Argument -c besagt, dass das naechste Argument als vollstaendige Kommandozeile zu behandeln ist. Das ist genau das, was beabsichtigt war. Das einzige Problem besteht darin, commandline entsprechend aufzubauen.

4.3. Prozesssteuerung

All das bisher Gesagte ist fuer sich noch nicht recht nuetzlich. Jetzt werden wir zeigen, wie man nach dem Ausfuehren eines Programmes mittels execl oder execv die Steuerung zurueckerhalten kann. Da diese Routinen das alte Programm einfach durch das neue ueberlagern, ist es erforderlich, um das alte zu retten, es zunaechst in zwei Kopien zu zerteilen; eine dieser Kopien kann ueberlagert werden, waehrend die andere auf die Beendigung des neuen ueberlagernden Programmes wartet. Das Zerteilen wird durch die Routine fork vorgenommen.

```
proc_id = fork();
```

erzeugt zwei Kopien, die beide weiter ausgefuehrt werden. Der einzige Unterschied zwischen ihnen ist der Wert proc_id, dem "Prozesskennzeichen". In einem der beiden Prozesse (dem "Kind") ist proc_id 0. Im anderen (dem "Elternprozess") ist proc_id ungleich 0; er entspricht der Prozessnummer des Kindes. Somit besteht der Aufruf und die Rueckkehr von einem anderen Programm in der Hauptsache aus folgendem:

```
if (fork() == 0)
   execl("/bin/sh", "sh", "-c", cmd, NULL);
     /* in child */
```

In der Tat, abgesehen von der Fehlerbehandlung reicht das aus. fork erzeugt zwei Kopien des Programmes. Im Kind ist der von fork gelieferte Wert 0, so dass dort execl gerufen wird, um das Kommando auszufuehren und anschliessend ist Schluss. Im Elternprozess gibt fork einen Wert ungleich 0 zurueck, so dass execl uebergangen wird. (Trat hierbei ein Fehler auf, so gibt fork -1 zurueck.) Meistens jedoch moechte der Elternprozess auf die Beendigung des Kindprozesses warten, bevor er selbst weiter arbeitet. Das kann ueber die Funktion wait erreicht werden:

```
int status;
if (fork () == 0)
   execl(...);
wait (&status);
```

Dabei werden immer noch keine abnormalen Bedingungen behandelt, wie z.B. ein Fehler bei execl oder fork oder die Moeglichkeit, dass mehr als ein Kind simultan ausgefuehrt

wird. (wait liefert das Prozesskennzeichen des Kindes, falls man es mit dem von fork gelieferten Wert vergleichen moechte.) Schliesslich behandelt dieses Fragment auch kein unnormales Verhalten des Kindes (dies ist in status kommentiert). Trotzdem sind diese drei Zeilen das Herz der Systemroutine in der Standardbibliothek, die wir gleich zeigen werden.

Der von wait gelieferte status verschluesselt in seinen niederen 8 Bits den Abbruchstatus: 0 bei normaler Beendigung, ungleich 0 um verschiedene Probleme anzuzeigen. Die hoeherwertigen 8 Bits sind dem Argument des exit-Rufes entnommen, der eine normale Beendigung des Kindprozesses verursachte. Es ist zu empfehlen, dass alle Programme einen sinnvollen Status liefern.

Wird ein Programm von Shell aufgerufen, zeigen die Filedeskriptoren 0, 1 und 2 auf die richtigen Dateien und alle anderen Filedeskriptoren sind verfuegbar. Wenn das Programm ein anderes aufruft, wird gesichert, dass die gleichen Bedingungen vorliegen. Weder fork noch execl bzw. execv haben irgendeinen Einfluss auf die eroeffneten Dateien. Wenn der Elternprozess Ausgaben puffert, die vor den Ausgaben des Kindes ausgegeben werden muessen, so muss der Elternprozess den Puffer leeren, bevor er execl aufruft. Umgedreht wird das gerufene Programm alle Informationen verlieren, die vom Aufrufer gelesen wurden, wenn dieser den Eingabestrom puffert.

4.4. Pipes

Eine Pipe ist ein E/A-Kanal zwischen zwei kooperierenden Prozessen. Ein Prozess schreibt in die Pipe waehrend der andere aus ihr liest. Das System sorgt fuer die Pufferung der Daten und die Synchronisierung der zwei Prozesse. Die meisten Pipes werden erzeugt, wie bei

```
ls | pr
```

wo die Standardausgabe von ls mit der Standardeingabe von pr verbunden wird. Manchmal jedoch ist es am besten, wenn ein Prozess selbst eine solche Verbindung herstellt; in diesem Abschnitt werden wir verdeutlichen, wie man dabei vorgehen kann.

Der Systemaufruf pipe erzeugt eine Pipe. Da sie sowohl fuer Lesen als auch fuer Schreiben verwendet wird, werden zwei Filedeskriptoren zurueckgegeben, die Verwendung sieht etwa folgendermassen aus:

```
int fd[2];
stat = pipe(fd);
if (stat == -1)
/* there was an error ... */
```

fd ist ein Feld von zwei Filedeskriptoren, wobei fd[0] die Leseseite und fd[1] die Schreibseite bezeichnet. Sie koennen ueber read, write und close genauso wie jeder andere Filedeskriptor verwendet werden.

Wenn ein Prozess eine leere Pipe liest, wird er warten, bis Daten eintreffen. Wenn ein Prozess in eine Pipe schreibt, die gefuellt ist, so wird er warten, bis aus der Pipe etwas gelesen worden ist. Wenn die Schreibseite der Pipe geschlossen wird, so liefert ein nachfolgendes read das Dateiende.

Um die Verwendung von Pipes an einem realistischen Beispiel zu illustrieren, wollen wir ein Funktion popen(cmd, mode) schreiben, die einen Prozess cmd erzeugt (genau wie es das System macht) und einen Filedeskriptor liefert, der diesen Prozess entweder lesen oder schreiben wird, entsprechend der mode-Angabe. Dass heisst der Ruf

```
fout = popen("pr", WRITE);
```

erzeugt einen Prozess, der das pr-Kommando ausfuehrt; nachfolgende write-Rufe mit dem Filedeskriptor fout werden ihre Daten diesem Prozess ueber eine Pipe senden.

popen erzeugt zuerst die Pipe durch einen Pipe-Systemruf, dann erfolgt fork, um zwei Kopien von sich selbst zu erzeugen. Das Kind stellt fest, ob es fuer Lesen oder Schreiben bestimmt ist und schliesst die andere Seite der Pipe. Dann ruft es Shell (ueber execl), um den entsprechenden Prozess auszufuehren. Der Elternprozess schliesst analog die Seite der Pipe, die er nicht benutzt. Das Schliessen ist notwendig, damit die Dateiendetests richtig funktionieren. Wenn z.B. ein Kind, das beabsichtigt zu lesen, die Schreibseite der Pipe nicht schliesst, so wird es niemals das Ende des Pipefiles erkennen, und zwar deshalb, weil ein Schreibprozess potentiell aktiv ist.

```
#include <stdio.h>
#define   READ  0
#define   WRITE 1
#define   tst(a, b)     (mode == READ ? (b) : (a))
static    int   popen_pid;
popen(cmd, mode)
char       *cmd;
int        mode;
{
   int p[2];
   if (pipe(p) < 0)
        return(NULL);
   if ((popen_pid = fork()) ==  0) {
        close(tst(p[WRITE], p[READ]));
        close(tst(0, 1));
        dup(tst(p[READ], p[WRITE]));
        close(tst(p[READ], p[WRITE]));
```

```
        execl ("/bin/sh", "sh", "-c", cmd, 0);
        _exit(1); /* disaster has occured if we get here */
    }
    if (popen_pid == -1)
        return (NULL);
    close(tst(p[READ], p[WRITE]));
    return(tst(p[WRITE], p[READ]));
}
```

Die Folge der close-Rufe ist etwas vertrackt. Angenommen, die Aufgabe besteht darin, einen Kindprozess zu erzeugen, der Daten vom Elternprozess lesen soll, dann schliesst der erste close-Ruf die Schreibseite der Pipe und laesst die Leseseite offen. Die Zeilen

```
        close(tst(0, 1));
        dup(tst(p[READ], p[WRITE]));
```

sind der uebliche Weg, den Pipedeskriptor mit der Standardeingabe des Kindes zu verbinden. close schliesst Filedeskriptor 0, d.h. die Standardeingabe. dup ist ein Systemruf, der ein Duplikat eines bereits offenen Filedeskriptors liefert. Filedeskriptoren werden in aufsteigender Folge zugewiesen und der erste verfuegbare wird zurueckgegeben, so dass der Effekt von dup darin besteht, den Filedeskriptor der Pipe (Leseseite) nach Filedeskriptor 0 zu kopieren, damit die Leseseite der Pipe zur Standardeingabe wird. (In der Tat, das ist etwas trickreich, aber es ist ein Standardidiom.) Man beachte, das dies nur funktioniert, wenn die Pipe nicht gleich dem Filedeskriptor 0 zugeordnet ist. Schliesslich wird die alte Leseseite der Pipe geschlossen.

Eine aehnliche Folge von Operationen findet statt, wenn der Kindprozess an den Elternprozess schreiben soll anstatt zu lesen. Es ist eine nuetzliche Uebung, diesen Fall einmal durchzuexerzieren.

Die Aufgabe ist noch nicht erledigt, da wir noch ein Funktion pclose benoetigen, die die Pipe schliesst, die von popen erzeugt worden ist. Der Hauptgrund dafuer, eine separate Funktion anstelle close zu verwenden, besteht darin, dass es wuenschenswert ist, auf die Beendigung des Kindprozesses zu warten. Erstens zeigt der Rueckgabewert von pclose an, ob der Prozess erfolgreich arbeitete. Wenn ein Prozess mehrere Kinder erzeugt, ist es ebenso wichtig, dass nur eine begrenzte Anzahl von Kindern existieren kann, auf die nicht gewartet wird, selbst dann, wenn einige von ihnen geendet haben; das Durchfuehren eines wait-Rufes "beerdigt" den Kindprozess.

```
#include <signal.h>
pclose(fd) /* close pipe fd */
int fd;
{
   register r, (*hstat)(), (*istat)(), (*qstat)();
   int  status;
   extern int popen_pid;
   close(fd);
   istat = signal(SIGINT, SIG_IGN);
   qstat = signal(SIGQUIT, SIG_IGN);
   hstat = signal(SIGHUP, SIG_IGN);
   while ((r = wait(&status) != popen_pid && r != -1);
   if (r == -1)
        status = -1;
   signal(SIGINT, istat);
   signal(SIGQUIT, qstat);
   signal(SIGHUP, hstat);
   return(status);
}
```

Die signal-Rufe garantieren, dass keine Unterbrechungen usw. die wartenden Prozesse behindern, das ist das Thema des naechsten Abschnittes.

So wie die Routine geschrieben wurde, besteht die Einschraenkung, dass nur eine Pipe zu einem bestimmten Zeitpunkt offen sein kann, und zwar wegen der einzelnen geteilten Variablen popen_pid. Sie sollte besser ein Feld sein, das vom Filedeskriptor indiziert wird. Eine popen-Funktion mit etwas anderen Argumenten und Ergebniswert steht in der Standard-E/A-Bibliothek zur Verfuegung.

5. Signale

5.1. Allgemeines

Dieser Abschnitt eroertert die Behandlung von Signalen von der Aussenwelt (wie Unterbrechungen). Da man nichts besonders Sinnvolles tun kann innerhalb C bei Programmfehlern, da sie in der Hauptsache aus nicht erlaubten Speicherplatzverweisen oder aus der Ausfuehrung sonderbarer Anweisungen erwachsen, werden wir nur die Signale der Aussenwelt diskutieren. interrupt, das gesendet wird, wenn DEL eingegeben wurde; quit, das durch CTRL-Backslash generiert wird; hangup, verursacht durch Abschalten des Terminals; terminate, bewirkt durch das Kill-Kommando. Geschieht eines dieser Ereignisse, wird das Signal an alle Prozesse gesendet, die von dem betreffenden Terminal gestartet wurden. Wenn nicht andere Vereinbarungen getroffen worden sind, beendet das Signal den Prozess.

5.2. Signalroutine

Die Standardaktion wird von der signal-routine geaendert. Sie besitzt zwei Argumente: das erste gibt das Signal an und das zweite legt die Behandlung fest. Das erste ist eine Zahl, aber das zweite Argument ist entweder die Adresse einer Funktion oder eine etwas sonderbare Kodierung, die entweder verlangt, dass das Signal ignoriert oder die Standardaktion durchgefuehrt werden soll. Die include-Datei signal.h definiert Namen fuer die verschiedenen Argumente und sollte immer eingefuegt werden, wenn Signale verwendet werden. Somit bewirkt

```
#include <signal.h>
 ...
signal(SIGINT, SIG_IGN);
```

das Ignorieren von Unterbrechungen (interrupts), waehrend

```
signal(SIGINT, SIG_DFL);
```

die Standardaktion der Prozessbeendigung wieder aktiviert. In jedem Fall gibt signal den vorherigen Wert des Signals zurueck. Das zweite Argument von signal kann stattdessen der Name einer Funktion sein (die explizit vereinbart sein muss, wenn der Compiler sie noch nicht zu Gesicht bekam). In diesem Fall wird diese Funktion aufgerufen, wenn das Signal auftritt. Meist wird diese Moeglichkeit genutzt, um vor dem Abbruch noch ausstehende Aktionen durchzufuehren, z.B. um eine temporaere Datei zu loeschen:

```
#include <signal.h>
main()
{
   int onintr();
```

```
        if (signal(SIGINT, SIG_IGN) != SIG_IGN)
                    signal(SIGINT, onintr);
        /* Process ... */
        exit (0);
    }

    onintr()
    {
        unlink(tempfile);
        exit (1);
    }
```

5.3. Interrupts

Warum nun der Test und der doppelte Aufruf von signal? Man erinnere sich, dass solche Signale wie ein Interrupt an alle Prozesse gesendet werden, die von diesem Terminal gestartet worden sind. Wenn ein Programm nicht interaktiv ausgefuehrt werden soll (gestartet durch &), so schaltet Shell fuer dieses Programm Unterbrechungen ab, so dass die Ausfuehrung durch Unterbrechungen, die fuer Vordergrundprozesse bestimmt sind, nicht angehalten wird. Wuerde dieses Programm als erstes veranlassen, dass alle Unterbrechungen grundsaetzlich an die onintr-Routine zu delegieren sind, so waeren dadurch die Bemuehungen von Shell, das Programm vor Unterbrechungen im Hintergrund zu schuetzen, zunichte gemacht.

Die oben gezeigte Loesung besteht darin, den Zustand der Unterbrechungsbehandlung zu testen und Unterbrechungen weiterhin zu ignorieren fuer den Fall, dass sie bereits ignoriert werden. Der geschriebene Kode ist abhaengig von der Tatsache, dass signal den vorhergehenden Zustand eines bestimmten Signals liefert. Werden Signale bereits ignoriert, so sollte das der Prozess weiterhin tun, andernfalls sollten sie abgefangen werden.

Ein etwas intelligenteres Programm koennte zunaechst verlangen, dass eine Unterbrechung abgefangen und als Forderung zur Beendigung der gegenwaertigen Aktivitaeten interpretiert wird, sowie anschliessend die eigene Kommandoverarbeitungsschleife wieder aufgenommen wird. Man denke dabei an einen Texteditor: der Abbruch eines laengeren Druckvorganges sollte natuerlich nicht zur Folge haben, dass die Editorarbeit beendet wird und all das bereits Erledigte verloren geht. Fuer diesen Fall bietet sich folgender Entwurf an:

```
    #include <signal.h>
    #include <setret.h>
    ret_buf    sjbuf;
    main()
    {
        int (*istat)(), onintr();
        istat=signal(SIGINT, SIG_IGN);
```

```
            /* save original status */
        setret(sjbuf); /* save current stack position */
        if (istat != SIG_IGN)
            signal(SIGINT, onintr);

    /* main processing loop */

    }

    onintr()
    {
        printf("\nInterrupt\n");
        longret(sjbuf); /* return to saved state */
    }
```

Die include-Datei setret.h vereinbart den Typ ret_buf als ein Objekt, in dem der Zustand gerettet werden kann. sjbuf ist ein derartiges Objekt, und zwar irgendein Feld. Die setret-Routine rettet dann den Zustand der Dinge. Tritt eine Unterbrechung auf, so wird ein Ruf der onintr-Routine erzwungen, welche eine Nachricht senden und Anzeigen setzen kann usw. longret besitzt als Argument ein Objekt, das von setret gespeichert wurde und gibt die Steuerung an die Stelle nach dem setret-Ruf zurueck. Damit geht die Steuerung an die Stelle der Hauptroutine, wo das Signal aufgesetzt wurde und die Hauptschleife beginnt. (Das Ruecksetzen betrifft auch das Stackniveau.) Man beachte im uebrigen, dass das Signal nach einer erfolgten Unterbrechung wieder gesetzt wird. Das ist noetig, da die meisten Signale, wenn sie auftreten, automatisch auf ihre Standardaktion rueckgesetzt werden.

Einige Programme, die Signale entdecken moechten, koennen nicht so einfach an einem beliebigen Punkt angehalten werden, z.B. mitten in der Aenderung einer verketteten Liste. Wird die Routine nach dem Auftreten eines Signals gerufen, so wird eine Anzeige gesetzt und dann, anstatt exit oder longret zu rufen, am Unterbrechungspunkt fortgesetzt. Die Unterbrechungsanzeige kann spaeter abgetestet werden.

Bei einem solchen Ansatz tritt ein Problem auf. Angenommen das Programm liest gerade vom Terminal, wenn die Unterbrechung gesendet wird. Die angegebene Routine wird gerufen, es wird die Anzeige gesetzt und zurueckgesprungen. Traefe das oben gesagte ein, d.h. es wuerde am Unterbrechungspunkt fortgesetzt, so wuerde weiter vom Terminal gelesen bis der Nutzer eine neue Zeile eintippt. Dieses Verhalten koennte natuerlich Verwirrung stiften, da dem Nutzer moeglicherweise nicht bewusst ist, dass das Programm weiterliest. Sehr wahrscheinlich wuerde er es vorziehen, wenn das Signal sofort wirksam wird. Die Methode derartige Schwierigkeiten zu loesen, besteht darin, dass das Lesen vom Terminal beendet wird, wenn die Ausfuehrung nach dem Signal wieder aufgenommen wird. Anschliessend wird eine Fehleranzeige geliefert, die andeutet, was

geschah.

Folglich sollten Programme, die Signale abfangen und die Steuerung anschliessend wieder aufnehmen, auf Fehler vorbereitet sein, die durch unterbrochene Systemrufe verursacht wurden. (Vorsicht ist beim Lesen vom Terminal sowie bei wait und pause geboten.) Ein Programm, dessen onintr-Routine intflag setzt, das Unterbrechungssignal ruecksetzt und die Steuerung zurueckgibt, sollte gewoehnlich Kode der folgenden Art enthalten, wenn es die Standardeingabe liest:

```
if (getchar() == EOF)
   if (intflag)
        /* EOF caused by interrupt */
   else
        /* true end-of-file */
```

Eine letzte Feinheit, die man sich merken sollte, gewinnt an Bedeutung, wenn das Abfangen von Signalen mit der Ausfuehrung anderer Programme verbunden ist. Angenommen ein Programm faengt Unterbrechungen ab und enthaelt ausserdem eine Methode (wie "!" beim Editor), andere Programme auszufuehren. Der Kode sollte dann etwa so aussehen:

```
if (fork() == 0)
   execl(...);
signal(SIGINT, SIG_IGN); /* ignore interrupts */
wait(&status); /* until the child is done */
signal(SIGINT, onintr); /* restore interrupts */
```

Warum das? Es ist zwar wiederum nicht offensichtlich, aber nicht schwierig. Angenommen, das Programm, das gerufen werden soll, faengt selbst Unterbrechungen ab. Wenn man das Unterprogramm unterbricht, so wird es das Signal erhalten, zu seiner Hauptschleife zurueckkehren und moeglicherweise vom Terminal lesen. Aber das aufrufende Programm wird auch aufhoeren auf das Unterprogramm zu warten und das Terminal lesen. Es ist jedoch sehr unvorteilhaft, wenn zwei Programme von einem Terminal lesen, da das System, bildlich gesprochen, eine Muenze wirft, um zu entscheiden, wer jede Eingabezeile erhalten soll. Ein einfacher Ausweg besteht darin, dass das Elternprogramm Unterbrechungen solange ignoriert, bis das Kindprogramm abgearbeitet ist. Die Begruendung wird von der system-Funktion der Standard-E/A-Bibliothek widergespiegelt:

```
#include <signal.h>
system(s) /* run command string s */
char *s:
{
   int status, pid, w;
   register int(*istat)(), (*qstat)();

   if ((pid = fork()) == 0) {
        execl("/bin/sh", "sh", "-c", s, 0);
        _exit(127);
   }
   istat = signal(SIGINT, SIG_IGN);
   qstat = signal(SIGQUIT, SIG_IGN);
   while ((w = wait(&status)) != pid && w != -1)
        ;
   if (w == -1)
        status = -1;
   signal(SIGINT, istat);
   signal(SIGQUIT, qstat);
   return(status);
}
```

Notizen:

C A S

U8000-Assembler

Vorwort

Diese Unterlage beschreibt die U8000-Assemblersprache des Betriebssystems WEGA. Sie bildet die Grundlage fuer den Assemblerprogrammierer unter WEGA.

Im Abschnitt 1 wird eine kurze Einfuehrung gegeben. Ihr folgt in vier Abschnitten die Beschreibung der Assemblersprache. Der Anhang enthaelt eine Zusammenfassung der Assembler-Steueranweisungen, der Schluesselwoerter, der U8000-Befehlsmnemoniken und der Assemblerfehlernachrichten.

Die Beschreibungen zur Handhabung des Assemblers und des Laders (Linker) sind im WEGA-Programmierhandbuch unter cas(1), ld(1) und sld(1) zu finden.

Inhaltsverzeichnis Seite

1. Allgemeine Informationen

1.1. Ueberblick ueber die Assembler

Der verschiebliche U8000-Assembler fuer das P8000, der cas genannt wird, laeuft unter dem WEGA-Betriebssystem. Er uebersetzt die Grundprogramme der Assemblersprache in die Objekt-Module, die durch P8000 entweder getrennt ausgefuehrt werden koennen oder mit anderen Assembler-Objekt-Modulen verbunden sein kann, um ein komplettes Programm zu bilden. Ein Editor wird zur Schaffung eines Grundmodules der Assemblersprache (Datei) verwendet. Der Grunddateiname sollte mit dem Anhang .s enden. Befehle zum Aufruf des Assemblers sind in cas (1) enthalten. Der Assembler ist ein Assembler mit 2 Arbeitsgaengen. Waehrend des ersten Arbeitsganges formt er die Symboltabelle und schafft eine Verbindungsdatei, die geloescht wird, wenn die Zusammenstellung beendet ist. Symbole, die eine Variablenlaenge haben koennen, erscheinen in der Symboltabelle in jener Reihenfolge, in der sie im Programm der Assemblersprache definiert sind. Waehrend des zweiten Arbeitsganges bildet der Assembler ein verschiebbares Zielmodul im a.out (5) Format und mit dem Default-Dateinamen a.out. Das Merkmal der Verschiebbarkeit eines Assemblers befreit den Programmierer von der Handhabung des Speichers waehrend der Programmentwicklung (da Zielkode im Speicher verschoben werden koennen) und gestattet die Entwicklung von Programmen in Modulen, deren Adressen automatisch aufgeloest werden, sobald die Module verbunden sind.

1.2. Beziehung zum PLZ/ASM-Assembler

Der P8000-Assembler koexistiert mit dem P8000 PLZ/ASM-Assembler. PLZ/ASM-Programme koennen jedoch nicht vom P8000-Assembler zusammengestellt werden, noch koennen P8000-Assembler vom PLZ/ASM-Assembler zusammengestellt werden. Nachfolgend wird der P8000 Assembler einfach Assembler genannt. Jegliche Hinweise auf den PLZ/ASM-Assembler werden deutlich gemacht, um eine Verwechslung der beiden Assembler zu verhindern.

1.3. Bemerkungen zur Implementation

Jegliche Einschraenkungen, die mit einem bestimment Release des Assemblers verbunden sind, werden im WEGA-Programmierhandbuch cas(1) festgehalten.

2. Sprachstruktur

2.1. Einleitung

Dieser Abschnitt beschreibt die Grundstruktur der Assemblersprache, umfasst Zahlen, Ausdruecke und ein- und zweigliedrige Operatoren.

2.2. Zeichenketten

Eine Zeichenkette besteht aus einer Zeichenfolge, die in doppelten Anfuehrungszeichen (") oder einfachen Anfuehrungszeichen (') steht. Konstruktive Zeichenketten sind verknuepft. Zeichenketten koennen kein tatsaechliches Newline enthalten. Tabelle 2-1 beschreibt die Spezialzeichen, die innerhalb einer Zeichenkette benutzt werden koennen.

Character	Definition
\0	null
\n	newline
\t	tab
\b	backspace
\l	linefeed
\r	carriage return
\f	formfeed
\\	backslash
\"	double quote
\'	single quote
\%nn	two hexadecimal digits that form an arbitrary bit pattern

Es folgen Beispiele gueltiger Zeichenketten.

```
"This is a string"
"This is a null terminated string"
"This is a \" double quote within a string"
"This is a \' single quote within a sting"
"This is \n Multi-line \nString"
"Here is a \t tab, \b backspace, and \r cr"
"Here is a \f formfeed \\ backslash and \%AB hex %AB"
```

2.3. Zahlen

Zwei Zahlentypen werden vom Assembler unterstuetzt. Ganze Zahlen und Gleitkommazahlen.

2.3.1. Ganze Zahlen

Ganze Zahlen koennen in Form von Dezimal-Hexadezimal, Oktal- oder Binaerzahlen vertreten sein. Es folgen Beispiele fuer jede Darstellung:

```
5023            decimal
%FA2E           hexadecimal
%(8)7726        octal
%(2)10011101    binary
```

2.3.2. Gleitkommazahlen

Eine Gleitkommazahl besteht aus dem Teil einer ganzen Zahl, einem Bruchstueck und einem Exponententeil. Dem Exponententeil. Dem Exponententeil ist ein "E" oder "e" vorangestellt. Es muss entweder das Dezimal-Komma oder das "E" oder "e" vertreten sein, um eine Gleitkommazahl zu bilden. Nur Dezimaldigits koennen fuer eine Gleitkommazahl verwendet werden.

```
3.0
.023
3.23
3.23E7
4.5e6
2E7
2.
```

Gleitkommazahlen werden stets durch einen Gleitkomma-Konvertierungsoperatoren eingeleitet. Diese Operatoren, die in Tabelle 2.2 zusammengefasst sind, operieren nur bei ganzen Zahlen und Gleitkommazahlen. Sie koennen in Ausdruecken nicht verwendet werden.

Tabelle 2.2 Floating Point Conversion Operators:

Operator	Conversion
^F	Convert to floating double extended
^FD	Convert to floating double
^FS	Convert to floating single

Es folgen Beispiele fuer gueltige und ungueltige Verwendungen:

gueltig	ungueltig
^F 3.5	^F (3+4)
^F10	^FS L1
^FD 7	2 - ^FD 3.5
^FS 10E7	
^FD3.5	
^F .23E4	

2.4. Identifikatoren

Ein Identifikator ist ein nicht numerisches Zeichen, das von einer Variablenzahl numerischer oder nichtnumerischer Zeichen gefolgt wird. Ausser den upper und lower-case-letters kann ein nichtnumerisches Zeichen "_" oder "?" sein. Ausser den Dezimaldigits kann ein numerisches Zeichen "." sein. Identifikatoren koennen bis zu 128 Zeichen sein. Es folgen Beispiele fuer gueltige und ungueltige Identifikatoren.

gueltig	ungueltig
Myname	2label
count1	2_chickens
L1	.dot
done?	
_end	
label.one	

2.4.1. Schluesselworte und lokale Identifkatoren

Zwei Spezialformen der Identifikatoren werden von den Assemblers unterstuetzt. Schluesselworte und lokale Identifkatoren. Schluesselwort-Indikatoren sind eine spezielle Art von Identifikatoren, die dem Assembler als Schluesselworte vorbehalten sind. Eine Art von Schluesselworten, die Assembler-Direktive, wird vom Assembler als solchen direkt anerkannt, da sie stets von einer Periode (".") eingeleitet wird. Der Rest des Schluesselwort-Indikatorensatzes besteht aus Befehlsmnemoniken, flag codes und condition codes. Sie werden im Anhang B aufgefuehrt. Schluesselwort-Indikatoren werden in allen upper-case und lower-case anerkannt:

.SEG	LD
.nonseg	ld
.word	INC
.byte	.EVEN

Identifikatoren werden zu lokalen Marken, wenn sie durch ein "~" eingeleitet werden. Fuer weitere Informationen ueber lokale Marken siehe Abschn. 3.

```
~LI
~1
~jelly
~parm.1
```

2.5. Konstanten

Ein konstanter Wert ist ein Wert, der sich durch das gesamte Programm-Module nicht veraendert. Konstanten koennen als Zeichenketten oder als Identifikator

ausgedrueckt werden, der einen konstanten Wert repraesentiert. Identifikatoren koennen die Form von internen, lokalen oder globalen Marken annehmen, wie in Abschnitt 3 beschrieben.

2.6. Eingliedrige und zweigliedrige Operatoren

Um eine Assembler-Arithmetik auszufuehren, werden Ausdruecke unter Verwendung von eingliedrigen und zweigliedrigen Operatoren in Verbindung mit konstanten und Variablennamen gebildet. (Variablennamen koennen als Teil von Ausdruecken, jedoch nicht die Variablen selbst, verwendet werden.) In Tabelle 2-3 sind die eingliedrigen Operatoren in Rangfolge aufgefuehrt, die zweigliedrigen Operatoren sind in Tabelle 2-4 aufgefuehrt. Eingliedrige Operatoren besitzen Prioritaet ueber zweigliedrige Operatoren, Klammern koennen jedoch verwendet werden, um sich ueber die Prioritaet der Bestimmung in einem Ausdruck hinwegzusetzen.

Tabelle 2.3 Unary Operators:

Operator	Function
+	unary plus
-	unary minus
^B	binary coded decimal
^C	ones complement
^FS	convert to floating single
^FD	convert to floating double
^F	convert to floating extended
^S	segment (see Section 4.3)
^O	offset (see Section 4.3)

Tabelle 2.4 Binary Operators in Order of Precedence:

Operator	Function
*	multiply
/	divide
^<	shift left
^>	shift right
^$	bitwise and
^|	bitwise or
^X	bitwise xor
+	binary plus
-	binary minus

2.7. Ausdruecke - Assemblerarithmetik

Arithmetik wird in einem Programm der Assemblersprache in zwei Wegen durchgefuehrt. Run-time Arithmetik wird ausgefuehrt, waehrend das Programm eigentlich ausfuehrt und wird durch einen Befehl in der Assemblersprache explizit

definiert.

```
SUB R10, R12 // Substrahiere Inhalt des
             // Registers 12 vom Inhalt
             // des Registers 10
```

Assembly-time Arithmetik wird vom Assembler ausgefuehrt, wenn das Programm zusammengestellt wird und die Bestimmung von Ausdruecken in Operanden, wie folgt, einschliesst:

```
LD  R0,  #(22/7 + X)
JP  Z, LOOP1 + 12
ADD R2, #HOLDREG-1
```

Assembly-time Arithmetic ist beschraenkter als die Run-time Arithmetik in Bereichen, wie signed versus unsigned Arithmetik und im Bereich der zugelassenen Werte. Nur die unsigned Arithmetik ist bei der Berechnung der Assembly-time-Ausdruecke gestattet. Run-tim Arithmetik verwendet sowohl signed als auch unsigned Mode, da sie vom spezifizierten Assemblersprachenbefehl determiniert und mit den Operanden durch den Programmierer verbunden ist. Die gesamte Assembly-time Arithmetik wird unter Verwendung der 32-bit Arithmetik, "modulo 4,294,967,296", berechnet. Werte, die groesser oder gleich 4,294,967,296 sind, werden durch 4,294,967,296 dividiert und der Rest der Division wird als Ergebnis verwendet. In Abhaengigkeit von der Anzahl der durch einen bestimmten Befehl erwuenschten Bits, werden nur die am weitesten rechts stehenden 4, 8, 16 oder 32-Bits des resultierenden 32-bit-Wertes verwendet. Sollte das Ergebnis der Assembly-time Arithmetik in 4 Bytes gespeichert werden, wird der Wert "modulo 16" genommen, um ein Ergebnis im Bereich 0 bis 15 zu haben. Soll das Ergebnis in einer 1-Byte-Adresse gespeichert werden, wird der Wert "modulo 256" genommen, um ein Ergebnis im Bereich von 0 bis 255 zu haben. Soll das Ergebnis in einem Wort gespeichert werden, wird der Wert "modulo 65536" genommen, um ein Ergebnis im Bereich von 0 bis 65535 zu haben.

```
LDB  RL4, #X+22 //Result of (X+22) must be in
                //range 0 to 255

JP   X+22       //Modulo 65536. Result is the
                //address 22 bytes beyond X
                //and may wraparound through
                //zero

ADDL RR12, #32000*MAX //Result of 32000*MAX) is
                      //taken modulo 4,294,967,296
```

Alle arithmetischen Ausdruecke besitzen einen mit ihnen verbundenen Mode: absolut, verschiebbar und extern. In den folgenden Erlaeuterungen werden diese Abkuerzungen benutzt:

```
AB - absoluter Ausdruck
RE - verschiebbarer Ausdruck
```

EX - externer Ausdruck

2.7.1. Absolute Ausdruecke

Ein absoluter Ausdruck besteht aus einer oder mehreren Zahlen oder absoluten Konstanten, die mit eingliedrigen oder zweigliedrigen Operatoren verbunden sind. Der Unterschied zwischen zwei verschiebbaren Ausdruecken wird ebenfalls als absolut angesehen. Die verschiebbaren Ausdruecke muessen im selben Bereich derselben Sektion sein. Sind sie es nicht, kann die absolute Differenz auf assembly time nicht determiniert werden. (Fuer weitere Informationen ueber Programmsektionen und - Bereiche siehe Teil 5). Ein absoluter Ausdruck wird wie folgt definiert:

```
AB --> a number or absolute constant
       AB <operator> AB
       '+' AB, '-' AB
       RE '-' RE
```

Die Konstrukteure der Segment-Adresse "<<" und ">>" koennen in einem absoluten Ausdruck zur Bildung eines langen Wertes verwendet werden. Z.B.

```
<<3>>%100
```

entspricht dem langen Wert.

```
%03000100
```

und kann in jedem Ausdruck verwendet werden, wo lange Werte verwendet werden koennen. Zeichenketten koennen ebenfalls als absolute Werte verwendet erden. Jedoch nur die ersten vier Zeichen einer Zeichenkette werden zur Bildung des absoluten Wertes verwendet. Zur Befehls-Assembly-Time wird jede Segment-Direkt-Adresse die in den niederwertigen Bytes des Segmentteils keine Nullen haben, als Fehler markiert. Ausserdem wird das hochwertige Bit fuer Segment-Adressen zu diesem Zeitpunkt gesetzt. Beispiele fuer gueltige absolute Ausdruecke (wo L1 und L2 verschiebbare Marken und c1 ein konstanter Identifikator sind) sind:

```
%(8)2767 + (3 * 5)
c1 * 6 + %(2)01001100
%FEFEABAB + (L1 - L2)
5 ^< 8
3 + <<2>>%100
4 + "ABCD"
```

Beispiele fuer ungueltige absolute Ausdruecke sind:

```
2 + L1
(L1 * 3) - L2
c1 + (L1 -3)
```

2.7.2. Verschiebbare Ausdruecke

Ein verschiebbarer Ausdruck enthaelt genau ein IdentifikatorSubjekt zur Verschiebung nach der Zusammenstellung. Der Ausdruck kann durch die Addition oder Subtraktion eines absoluten Ausdrucks erweitert werden. Plus und Minus sind die einzig zugelassenen Operatoren. Ein verschiebbarer Ausdruck kann wie folgt definiert werden:

```
RE --> a relocatable identifier
        RE '+' AB
        AB '+' RE
        RE '-' AB
        +RE
```

Beispiele fuer gueltige verschiebbare Ausdruecke (wo L1 und L2 verschiebbare Marken und c1 ein konstanter Identifikator sind) sind:

```
L1 + c1
L1 + (%(8)077 - %FE02 / 2) ^> 4
c1 + (L1 - L2) + L2
L1 - (L1 - L2) + c1
```

Beispiele fuer ungueltige Ausdruecke sind:

```
c1 - L1
(%203F) - 100 - L2
(L1 - L2) * L2
L1 / L2
L1 + (L2 - c1)
```

2.7.3. Externe Ausdruecke

Ein externer Ausdruck enthaelt genau einen externen Identifikator, der moeglicherweise durch das Addieren oder Subtrahieren eines absoluten Ausdrucks erweitert sein kann. Ein externer Identifikator wird im gegenwaertigen Modul verwendet, aber in einem anderen Modul definiert. Der Wert eines externen Identifikators ist unbekannt, bis die Module verbunden sind. Ein externer Ausdruck wird wie folgt definiert:

```
EX --> external identifier
        EX '+' AB
        AB '+' EX
        EX '-' AB
        +EX
```

Beispiele fuer gueltige externe Ausdruecke (wo L1 eine verschiebbaere Marke, c1 ein konstanter Identifikator und el eine externe Marke ist) sind:

```
e1 - c1
e1 - (L1 - L2) + 5
c1 + e1
(%304 - 5) + e1
%(2)01100111 * 2 + e1
```

Beispiele fuer ungueltige externe Ausdruecke sind:

```
e1 + (L1 - e1)
%FEFE - e1
c1 * 2 + e1 - L1
2 * e1
e1 ^> 8
```

3. Anweisungen der Assemblersprache

3.1. Einleitung

Dieser Teil beschreibt die Fehler und die Syntax der Anweisungen der Assemblersprache. Die Konventionen, die bei der Assemblerbeschreibung der Syntax verwendet werden, sind folgende:

Parameter, die in winkligen Klammern gezeigt werden, sind Items, die durch eigentliche Daten oder Namen ersetzt werden sollen < section_name >

Optionale Items stehen in Klammern (< expression >)

Parameter, die durch "|" getrennt sind, zeigen an, dass der eine oder der andere Parameter, jedoch nicht beide, verwendet werden kann.

Die moegliche Wiederholung eines Item wird durch das Anhaengen von "+" an das Item angezeigt (um eine oder mehrere Wiederholungen anzukuendigen) oder von "*" (um keine oder mehrere Wiederholungen anzukuendigen) (<expression>)* Jede Wiederholung nach der ersten muss durch ein Komma eingleitet werden.

Andere Spezialzeichen, die im Anweisungs- oder Kommandoformat gezeigt werden, wie :=, (), stehen in einfachen Anfuehrungszeichen und muessen, wie gezeigt, geschrieben werden.

Das Spezialsymbol ":=" bedeutet "ist definiert als" oder "ist zugewiesen". Jede Marke, der ein Zeichen mit dieser Konstruktion zugewiesen wird, kann spaeter nicht neu definiert werden.

3.2. Anweisungen der Assemblersprache

Assemblersprachen-Programme bestehen aus Anweisungen der Assemblersprache, die bis zu 4 Fehler haben koennen:

Das Markenfeld -- definiert symbolisch eine Stellung in einem Programm.

Das Operatorfeld -- spezifiziert die durch die Anweisung auszufuehrende Handlung.

Das Operandfeld -- beinhaltet die Daten oder die Adresse der Daten, mit denen operiert wird.

Das Kommentarfeld -- beinhaltet einen Kommentar, um die Handlung der Anweisung zu dokumentieren.

Tabelle 3.1 fasst diese Felder, die im Rest dieses Teils beschrieben werden, zusammen. Jedes Feld muss von den

anderen Feldern durch ein oder mehrere Trennzeichen getrennt sein. Ein Trennzeichen kann eines der folgenden sein:

Leerzeichen
Tabulator
Semikolon

Ein Komma ist zur Trennung der Komponenten im Operandenfeld erforderlich. Jede Anweisung der Assemblersprache wird durch ein Newline- oder Carriage-Return-Zeichen beendet. Wenn eine Anweisungslaenge die Zeilenlaenge ueberschreitet, kann sie auf der naechsten Zeile durch die Verwendung des Zeilenfortsetzungszeichens "\" fortgesetzt werden. Es folgt ein Muster einer Anweisung der Assemblersprache:

```
Label   Operator Operand(s)     Comment
L1:     LD       R0, R1         //Load contents of
                                //Register 0 in Reg. 1
```

Tabelle 3.1 Summary of Language Statement Fields:

Field	Field Types
Label	Internal Global Local External Common
Operator	Directive Direct Assignment Data Declarator Instruction
Operands	Address Data Condition Code
Comment	

Beachte, dass die Reihenfolge der Felder, die im Beispiel gezeigt wird, nicht erforderlich ist. Waehrend Kommentare stets im letzten Feld einer Anweisung stehen, (wenn sie verwendet werden), muessen die Marken den Operatoren nicht unbedingt voranstehen. Wenn der Operator z.B. eine Direktive ist, kann eine Marke folgen:

```
.extern L1
```

3.3. Marken

Eine Marke identifiziert eine Anweisung in einem Programm und gestattet, dass in dieser Anweisung symbolische Verweise angebracht sind. Konstanten, Befehle,

Direktiven und Datenerlaeuterer koennen alle mit Marken versehen sein. Jede Anweisung, auf die durch eine andere Anweisung verwiesen wird, muss mit einer Marke versehen sein. Es kann pro Anweisung mehr als eine Marke existieren. Die folgenden Markentypen treffen auf diese Beschreibung zu:

```
internal
global
local
```

2 zusaetzliche Markentypen, extern und common, werden mit den Assembler-Direktiven bzw. .extern und .comm definiert. Man kann sie durch den Fakt unterscheiden, dass man im gegenwaertigen Modul (Datei) verweisen kann, sie aber als globale in einem anderen Modul definiert sind.

```
external
common
```

3.3.1. Interne Marken

Eine interne Marke besteht aus einem Identifikator, der von ":" gefolgt wird. Eine interne Marke schraenkt den Zugriff auf einen Identifikator durch das Modul, in dem er definiert ist, ein.

```
start:  LD R0,R1          //eine interne Marke fuer
                          //einen Befehl

count_1: .word %200       //eine interne Marke fuer
                          //eine Datenvereinbarung

begin: .psec mysection    //eine interne Marke fuer
                          //eine Assembler-Direktive
```

3.3.2. Globale Marken

Eine globale Marke besteht aus einem von "::" gefolgten Identifikator. Es gestattet den Zugriff auf den Identifikator durch andere Module als denen, wo er definiert ist.

```
L1::L2::.word %ABCD //two global labels for
                    //a data declaration

L1::
L2:: .word %ABCD  //same as preceding example

done?:: PUSH @R15, R0   //a global label for an
                        //instruction

_start::.psec   //a global label for a directive
```

```
foobar::= %20   //a global constant with
                //value %20
```

Eine Marke selbst auf einer Zeile wird als eine Null-Anweisung betrachtet. Solche Anweisung ist mit der naechsten Nichtnull-Anweisung im Programm verbunden.

```
start::         //Null-Anweisung besteht nur aus
                //einer Marke

begin:: .code   //kennzeichnet den Anfang des
                //Code-Bereiches
```

3.3.3. Lokale Marken

Eine lokale Marke besteht aus einem durch "~" eingeleiteten Identifikator (das den Identifikator zu einem lokalen Symbol macht) und wird von ":" gefolgt. (Lokale Marken sind nur in lokalen Bloecken, wie im Teil 5, Programmstruktur beschrieben, gueltig)

```
~L1:~L2:: LD R0, #20    //two local labels for
                        //an instruction

~num:= %100             //a local constant with
                        //value 100 (hex)

~count:.odd     //a local label for a directive
```

3.3.4. Externe Marken

Extern spezifiziert, dass auf eine Marke im gegenwaertigen Modul verwiesen werden kann, sie aber als globale in einen anderen Modul definiert ist. Externe Marken sind mit der externen Direktive, .extern, definiert.

```
.extern prod, done?  //externe Marken sind
.extern datum, _end  //vereinbart
```

3.3.5. Common Marken

Common Marken bestehen aus der.comm-Direktive, die von einem konstanten Ausdruck gefolgt wird, der die Byte-Anzahl der mit dem common Symbol (en) und einem Komma verbundenen Speicherung anzeigt. Diese werden von einer Liste von Identifikatoren gefolgt, die durch Kommas getrennt sind. Zum Zeitpunkt der Verbindung werden common Symbole mit demselben Namen, aber aus verschiedenen Dateien untersucht. Die common Marke mit dem groessten Ausmass wird als nicht initialisierte Dateien untersucht. Die common Marke mit dem dem groessten Ausmass wird als nichtinitialisierte Daten (BSS Speicherung) bestimmt. Wird eine globale Definition mit demselben Namen gefunden) verweisen alle common Marken

auf die globale Definition

```
        .comm 20, data1, data2  //2 common Symbole
                                //der Groesse 20
        .comm 5+3, myname  //common Symbol der Groesse 8
```

3.4. Operatoren

Das Operatorfeld spezifiziert die durch die Anweisung auszufuehrende Handlung. Dieses Feld kann eines der folgenden enthalten:

```
        directive
        direct assignment
        data dedarator
        instruction
```

3.4.1. Assembler-Directiven

Eine Assembler-Directive lenkt entweder die Operation des Assemblers oder bestimmt die Speicherung, hat jedoch selbst keinen ausfuehrbaren Code zur Folge. Eine Periode "." geht jeder Assembler-Directive voran. Tabelle 3.2 gibt eine funktionale Zusammenfassung der Direktiven und einen Hinweis auf den Teil, der eine Beschreibung der Direktiven und Beispiele fuer deren Anwendung enthaelt.

Tabelle 3.2 Functional Summary of Assembler Directives:

Category	Directives	See
Data Stroage and Initialization Directives	.byte .word .long .quad .extend .addr .blbk .blkw .blkl	Section 3
Label Control Directives	.comm .extern	
Segment Control Directives	.seg .nonseg	Section 4
Program Section Directives	.psec .csec .asec .data .bss .code	Section 5

Location Counter Control Directives	.even .odd
Listing Directive	.line

3.4.2. Direkte Zuweisung

Die Anweisung einer direkten Zuweisung gestattet den Symbolen die Verbindung mit Konstanten, Marken oder Schluesselworten. Die Anweisung einer direkten Zuweisung ist ein Symbol (gewoehnlich eine Marke), das von "=" und einem der folgenden gefolgt wird:

```
32-bit absolute constant
32, 64, or 80 bit floating point constant
Relocatable expression
Location Counter
Keyword (for keyword redefinition)
```

32-Bit Absolute Konstanten:

Einer internen, globalen oder lokalen Marke kann der Wert eines 32-bit-konstanten Ausdrucks zugewiesen werden.

```
c1:=20          //der internen Marke c1 wurde der
                //konstante Wert 20 zugewiesen.

c3::=2+3*5      //der globalen Marke c3 wurde der
                //konstante Wert 17 zugewiesen

~c4:=L1-L2      //der lokalen Marke ~c4 wurde die
                //absolute Differenz aus Marke L1
                //und L2 zugewiesen.

c5:=<<4>>%1020  //der internen Marke c5 wurde der
                //lange Wert %04001020 zugewiesen.
```

Gleitkommakonstanten:

Einer internen oder lokalen Marke (jedoch keiner globalen Marke) kann der Wert einer 32, 64 oder 80 Bit Gleitkommakonstanten zugewiesen werden. Die Gleitkommakonstante kann ein Ausdruck oder eine Gleitkommazahl sein, angefuehrt von einem floating point type conversion unary operator, wie in Teil 2 beschrieben. Gleitkommakonstanten koennen nur Gleitkommazahlen ersetzen.

```
L3:=^F 3        //Der internen Marke L3 wurde die
                //erweiterte Gleitkommadarstellung
                //von 3 zugewiesen.

glbl:=^FS 3.5   //Der internen Marke glbl wurde die
                //einfache Gleitkommardarstellung
                //von 3.5 zugewiesen.
```

```
~loc2:=^FD 2.23E7 //Der lokalen Marke
                  //~loc2 wurde die
                  //doppelte Gleitkommadasrstellung
                  //von 2.23E7 zugewiesen.
```

Verschiebbare Ausdruecke und Symbole:

Einer internen, globalen oder lokalen Marke kann der Wert eines verschiebbaren Ausdrucks zugewiesen werden.

```
c1:=.+2         //Der internen Marke c1 wurde der
                //Wert des gegenwaertigen Adress-
                //zaehlers plus 2 zugewiesen.

g3::=L47-%30    //Der globalen Marke g3 wurde die
                //Adresse der Marke L47-%30
                //zugewiesen.

~dum:=60+start  //Der lokalen Marke ~dum wurde der
                //Wert 60 plus die Adresse der
                //Marke Start zugewiesen.
```

Anmerkung: Sind ".", "L47" und "start" im Segment-Mode zusammengestellt, so sind sie ganze Segment-Adressen.

Die Adresszaehlersteuerung:

Dem Adresszaehlersymbol "." kann der Wert eines konstanten Ausdrucks, eines verschiebbaren Ausdrucks oder eines Adresszaehler-Relativ-Ausdrucks zugewiesen sein.

```
.=.+10          //Der Adresszaehler waechst um 10
.=20            //Dem Adresszaehler ist der Wert 20
                //zugewiesen.

.=.-(3+5)       //Der Adresszaehler wird
                //um 8 verringert

.=L2+10         //Der Adresszaehler wird 10 Bytes
                //hinter dem Symbol L2 gesetzt
```

Redefinition von Schluesselworten:

Eine lokale oder interne, jedoch keine globale Marke, kann zum Zwecke der Redefinition von Schluesselworten mit einem Schluesselwort verbunden sein. Die Redefinition von Schluesselworten gibt der Marke alle Attribute des Schluesselwortes,dem sie zugewiesen wurde.

```
location:=.     //Die interne Marke location ist
                //ein Synonym fuer "."

sdefault:=.psec //Die interne Marke sdefault ist
                //ein Synonym fuer .psec
```

```
        ~wval:=.word    //Die lokale Marke ~wval ist ein
                        //Synonym fuer .word

        ~pl:=R0         //Die lokale Marke ~pl ist jetzt
                        //ein Synonym fuer das Register R0.
```

3.4.3. Der Datendeklarator

Die Anweisung einer Datendeklaration weist die Speicherung zu und initialisiert sie. So eine Anweisung besteht aus einer Datendeklarations-Direktive, die von einer Marke (optional) angefuehrt und von einer Serie konstanter und verschiebbarer Ausdruecke gefolgt wird. Die 9 Datendeklarations-Direktiven sind:

```
        .byte
        .word
        .long
        .quad
        .extend
        .addr
        .blbb
        .blkw
        .blkl
```

```
 .byte(<number>'('<expression>')'|<expression>|'"'string'"')*
```

weist die Speicherung zu und initialisiert sie mit einem spezifizierten Byte-Wert(en), der eine Serie von konstanten und verschiebbaren Ausdruecken oder eine ascii-Zeichenkette sein kann. Die Zahl ist der Wiederholungsfaktor. Wenn eine Zahl spezifiziert ist, muss der Ausdruck in Klammern stehen; Zeichenketten stehen in doppelten Anfuehrungszeichen.

```
name:.byte "Hans Meier" //weist fuer die ascii-Reprae-
                        //sentation einer genannten Zei-
                        //chenkette die Speicherung zu

place:.byte "Anytown"\  //setzt eine lange Zeichenkette
            "Europe"    //auf der naechsten Zeile fort

L4::.byte 3, "joe"      //weist 4 Bytes mit dem Ausgangs-
                        //wert 3 und ascii-Zeichenkette
                        //joe zu.
```

```
 .word (<number> '('<expression>')'|<expression>)*
```

Weist die Speicherung zu und initialisiert sie mit einem spezifizierten Wortwert (en), der eine Serie von konstanten und verschiebbaren Ausdruecken sein kann. Die Zahl ist der Wiederholungsfaktor. Ist eine Zahl spezifiziert, muss der Ausdruck in Klammern stehen.

```
count:.word %20          //weist ein Wort mit dem Aus-
                         //gangswert %20 zu

L2:.word 20, 3+5, 5      //weist 3 Worte mit den Ausgangs-
                         //werten 20, 8 und 5 zu
```

.long (<number> '('<expression>'|<expression>)*

Weist die Speicherung zu und initialisiert sie mit dem Namen spezifizierten langen Wert(en), der eine Serie von konstanten und verschiebbaren Werten sein kann. Die Zahl ist der Wiederholungsfaktor. Ist eine Zahl spezifiziert, muss der Ausdruck in Klammern stehen.

```
.long 10 (%ABCDABCD)     //weist 10 lange Wert mit dem
                         //Ausgangswert %ABCDABCD zu
```

.quad (<number> '('<expression>')'|<expression>)*

Reserviert 64 Speicherungs-Bits. Nur das doppelte Voranstellen einer Gleitkommazahl fuellt die zugewiesene Speicherung voellig aus. Fuellt der Wert die zugewiesene Speicherung nicht voellig aus, wird keine Zeichenextension durchgefuehrt. Die Zahl ist der Wiederholungsfaktor. Ist die Zahl spezifiziert, muss der Ausdruck in Klammern stehen.

```
.quad %FFFFFFFF          //initialisiert die niederwer-
                         //tigen 32 Bits des quad mit
                         //%FFFFFFFF

.quad ^FS3.5             //initialisiert die niederwer-
                         //tigen 32 Bits des quad mit
                         //dem Gleitkommawert 3.5

.quad ^FD3.4             //initialisiert das gesamte quad
                         //mit der doppelten Gleitkomma
                         //zahl 3.4
```

.extend (<number> '('<expression>')'|<expression>)*

Weist 80 Bits der Speicherung zu. Nur ein erweitertes Voranstellen der Gleitkommazahlen fuellt die zugewiesene Speicherung voellig aus. Fuellt der Wert die zugewiesene Speicherung nicht voellig aus, wird keine Zeichenextension durchgefuehrt. Die Zahl ist der Wiederholungsfaktor. Ist eine Zahl spezifiziert, muss der Ausdruck in Klammern stehen.

```
.extend 10 (^F1.234E5)   //weist 10 erweiterte Gleit-
                         //kommazahlen mit dem Wert
                         //1.234E5 zu
```

.addr (<number> '('<expression>')'|<expression>)*

Wenn es im Nicht-Segment-Mode zusammengestellt wird, weist es die Speicherung zu und initialisiert sie mit dem spezifizierten 16-Bit-Wert, der eine Serie von konstanten und verschiebbaren Ausdruecken sein kann. Die Zahl ist ein Wiederholungsfaktor. Wird eine Zahl verwendet, muessen die Ausdruecke in Klammern stehen. Wenn im Segment-Mode zusammengestellt wird, weist es die Speicherung zu und initialisiert sie mit einem 32-Bit-Wert.

```
        .addr  L2        //weist  2  (Nicht-Segment)   oder
                         //4   (Segment)   Bytes   fuer  die
                         //Adresse L2 zu
```

.blkb <expression>

Weist Speicherung in Bytes zu. Die Anzahl der Bytes wird durch den Ausdruck spezifiziert. Es erfolgt keine Initialisierung.

```
        .blkb 20         //weist Speicher fuer 20
                         //Bytes zu
```

.blkw <expression>

Weist Speicherung in Worten zu. Die Anzahl der Worte wird durch den Ausdruck spezifiziert. Es erfolgt keine Initialisierung.

```
        .blkw (3+5)      //weist Speicher fuer 8
                         //Worte zu
```

.blkl <expression>

Weist Speicherung in langen Worten zu. Die Anzahl der langen Woerter wird durch den Ausdruck spezifiziert. Es erfolgt keine Initialisierung.

```
        cl:=20           //definiert die Konstante
        .blkl (2*c1)     //weist Speicher fuer 40
                         //lange Worte zu
```

Kommas sind fuer die Initialisierungslisten erforderlich; beachte diesen Daten-Deklarator:

```
        .byte 2-3
```

Er besitzt einen Wert, 2-3. Sollen 2 Werte initialisiert werden, verwende ein Komma:

```
        .byte 2, -3
```

3.4.4. Befehle

Ein Befehl ist die Assemblersprachenmnemonik, die eine spezifische, stattzufindende Handlung beschreibt.

3.4.5. Pseudobefehle

Die Mehrheit der Code im Assemblersprachprogramm werden normalerweise Assembler-Direktiven, Daten-Deklaratoren, direkte Zuweisungsanweisungen, die Assemblersprachbefehle und Gleitkommabefehle sein. Das P8000 ist in der Lage, eine Sprung- und Aufrufoptimierung auszufuehren. Trifft der Assembler auf Pseudosprung- und Aufrufbefehle (JPR und CALLR), bestimmt der den Bereich des Sprungs oder Aufrufs und erzeugt die relative (Kurz-) Form des Befehls (JR und CALR), wo immer es moeglich ist. Kann er die relative Form des Befehls nicht erzeugen, erzeugt er die absolute (long-) Form des Befehls (JP oder CALL).

Die Sprungoptimierung wird dem Programmierer explizit durch einen JPR-Steuerungsbefehl mit der folgenden Form geliefert:

```
JPR [cc] ',' <jpr_expr>
```

wobei cc jeder Konditionscode ist, der mit einem JP oder JR-Befehl verwendet werden kann

```
<jpr_expr>=><label>[('+'|'-') <const_expr>]
```

wobei <label> eine interne, globale oder lokale Marke und <const_expr> ein konstanter Ausdruck ist.

Ein JPR (<jpr_expr>)Ausdruck ist ein verschiebbarer Ausdruck, der genau einen verschiebbaren Wert (<label>) enthaelt. Das Ziel von einem JPR muss eine Programm-Marke mit einer optionalen Konstanten sein, die zu ihm addiert oder von ihm substrahiert ist. Eine bestimmte Form von <label> + <const_expr> kann jedoch nicht optimiert werden. Diese Form kann man am besten am folgenden Beispiel erlaeutern:

```
L1:     JPR     L2-300
        .
        .
        .
L3:     JPR     L99
        .
        .
        .
L2:
```

Ist L2-L1 kleiner als 300 Bytes, so ist JPR auf L1 eigentlich ein Rueckwaertssprung. Das Ziel rueckt

eigentlich weiter weg, wenn JPR auf L3 optimiert wird. Dieser Fall ist fuer die Handhabung sehr kostspielig und selten genug, um ihn nicht zu optimieren. Bestimmte JPR und CALLR-Befehle koennen zu einem Kurz-(relativen) Befehl nicht optimiert werden. Die folgenden Anweisungstypen koennen nicht optimiert werden:

1. Ein JPR oder CALLR-Befehl, dessen Ziel sich nicht in derselben Sektion befindet.

2. Ein JPR oder CALLR-Befehl, dessen Ziel sich nicht im selben Modul (extern) befindet.

Waehrend der Zusammenstellung ist es moeglich, auf eine Anweisung zu treffen, die nicht zusammengestellt werden kann, sofern die Sprungoptimierung nicht erfolgt ist. Wurde die Sprungoptimierung durchgefuehrt, bevor die Zielmarke fuer einen bestimmten Sprung gefunden wurde, wird der Sprung lang gemacht. Die folgenden Bedingungen bewirken, dass die Sprungoptimierung vor dem Ende des ersten Arbeitsganges des Assemblers erfolgt.

1. Direkte Zuweisung des Adresszaehlers (wie bei .=.+20)

2. Ein konstanter Ausdruck, der die Differenz von 2 verschiebbaren Werten (z.B. L1-L2) enthaelt, wenn es einen optimierbaren Sprung zwischen den beiden verschiebbaren Werten gibt.

Die Aufrufoptimierung

Der Assembler liefert ebenfalls die Aufrufoptimierung durch einen CALLR-Steuerungsbefehl, der einen relativen Aufruf immer dort erzeugen wird, wo es moeglich ist. Der Aufruf-Steuerungsbefehl hat folgende Form:

```
CALLR <jpr_expr>
```

wobei <jpr_expr> ein einfacher, verschiebbarer Ausdruck, wie zuvor beschrieben, ist. Aufrufe werden unter denselben Bedingungen optimiert, unter denen eine Sprungoptimierung bewirkt wird.

3.5. Operanden

Operanden liefern die Informationen, die ein Befehl zur Ausfuehrung der Handlung benoetigt. In Abhaengigkeit vom spezifizierten Befehl, kann dieses Feld keine oder mehrere Operanden haben. Ein Operand kann sein:

- zu verarbeitende Daten (direkte Daten)
- die Adresse einer Position, von der die Daten genommen werden sollen (source address)
- die Adresse einer Programmadresse, zu der die Programm-

steuerung gefuehrt werden soll
- ein Bedingungskode, der zum Lenken eines Ablaufs der Programmsteuerung verwendet wird.

Obwohl es eine Reihe gueltiger Operandenkombinationen gibt, muss man an die Grundkonvention denken: Der Zeilenoperand geht dem Sourceoperanden stets voran. Mit Ausnahme der direkten Daten und Konditionscode werden alle Operanden als Adressen ausgedrueckt: Register, Speicher und I/O-Adressen. Z.B. kann ein Operand einen Register bezeichnen, dessen Inhalte den Inhalten eines anderen Registers hinzugefuegt werden, um die Adresse der Speicheradresse zu bilden, die die source data (Basis-Index-Adressierung) beinhaltet. Adressmode und Operatoren sind Gegenstand des 4. Teils.

3.6. Kommentare

Kommentare werden verwendet, um den Programmcode als einen Wegweiser der Programmlogik zu dokumentieren und das Debugging des gegenwaertigen und zukuenftigen Programms zu vereinfachen. 2 Typen von Kommentaren sind verfuegbar: Der end-of-line-Kommentar und der multi-line-Kommentar. Der end-of-line-Kommentar beginnt mit dem Zeichen "//" und endet beim naechsten Carriage-Return.

```
        LD R0, R1       //das ist ein end-of-line-Kommentar
```

Der multi-line-Kommentar beginnt mit dem Zeichen "/*", endet mit dem Zeichen "*/" und erstreckt sich ueber eine oder mehrere Zeilen.

```
        LD R0, R1       /*das ist ein Beispiel
                        ** fuer einen
                        multi-line-Kommentar*/
```

Anmerkung:
Da kurze Offset-Adressen verschiebbar sein koennen, werden sie zur Verbindungszeit auf die Gueltigkeit hin ueberprueft. Beispiele fuer kurze Offset-Adressoperatoren:

4. Adressierungsarten und Operatoren

4.1. Einleitung

Dieser Teil beschreibt die Adressierungsarten und Operatoren des P8000 und umfasst Beispiele der Assemblerbefehle, die jene verwenden.

4.2. Adressierungsarten

Daten koennen durch 8 verschiedene Adressierungsarten spezifiziert werden:

- Direkte Daten
- Register
- Indirekte Register
- Direkte Adresse
- Indizierte Adresse
- Relative Adresse
- Basisadresse
- Basisindizierte Adresse

Es werden in Operanden Spezialzeichen zur Identifizierung bestimmter Adressenarten verwendet. Diese Zeichen sind:

"R" - einer Wortregisterzahl vorangestellt
"RH" oder "RL" - einer Byteregisterzahl vorangestellt
"RR" - einer Registerpaarnummer vorangestellt
"RQ" - einer Register-Vierfach-Zahl vorangestellt
"@" - einem Indirekt-Register-Hinweis vorangestellt
"#" - direkten Daten vorangestellt
"()" - wird verwendet, um den Verschiebungsteil einer indizierten Basis oder basisindizierten Adresse einzuklammern
"." - kennzeichnet die gegenwaertige program counter location, wird gewoehnlich bei der relativen Adressierung verwendet.

Die Verwendung dieser Zeichen wird in den folgenden Abschnitten beschrieben.

4.2.1. Direkte Daten

Direkte Daten sind der einzige Mode, der keine Register- oder Speicheradresse anzeigt, obwohl sie fuer die Absichten dieser Eroerterung als Adressierungsmode betrachtet werden. Der Operandenwert, der durch den Befehl im Adressierungsmode der Direkten Daten verwendet wird, ist der Wert, der im Operandenfeld selbst geliefert wird. Direkte Daten werden durch das Spezialzeichen " " eingeleitet und sind entweder ein konstanter Ausdruck (der Zeichenkonstanten und Symbole enthaelt, die die Konstanten repraesentieren) oder ein verschiebbarer Ausdruck. Direkte Datenausdruecke werden unter Verwendung der 32-Bit-

Arithmetik berechnet. In Abhaengigkeit vom verwendeten Befehl wird der durch die am weitesten rechts stehenden 4, 8, 16 oder 32 Bits repraesentierte Wert eigentlich verwendet. Eine Fehlernachricht wird fuer Werte erzeugt, die den gueltigen Bereich fuer den Befehl ueberschreiten.

```
        LDB   RH0, #100 //Load decimal 100 into byte
                        // register RH0

        LDL   RR0, #%8000 * REP_COUNT
                   //Load the value resulting from
                   //the multiplication of hexadeximal
                   //8000 and the value of constant
                   //REP_COUNT into register pair RR0
```

Wird ein Variablenname oder ein Adressausdruck mit "#" praefigiert, ist der verwendete Wert die Adresse, die durch die Variable oder das Ergebnis der Ausdrucksberechnung repraesentiert wird, nicht aber die Inhalte der dazugehoerigen Datenadresse. Im Nicht-Segment Mode haben alle Adressausdruecke einen 16-Bit-Wert zur Folge. Fuer Segment-Adressen bildet der Assembler automatisch das passende Format fuer eine lange Offset-Adresse, die die Segment-Zahl und das lange Offset in einem 32-Bit-Wert einschliesst. Es wird empfohlen, symbolische Namen ueberall dort zu verwenden, wo es moeglich ist, da die dazugehoerige Segment-Zahl und das Offset fuer den symbolischen Namen vom Assembler automatisch gehandhabt wird und ihm spaeter Werte zugewiesen werden koennen, wenn das Modul fuer die Ausfuehrung verbunden oder beladen wird. In jenen Faellen, da ein spezifisches Segment erwuenscht ist, kann die folgende Bezeichnung verwendet werden (der Segment-Bezeichner steht in doppelten winkligen Klammern):

```
        <<segment>>offset.
```

wo "segment" ein konstanter Ausdruck ist, der fuer einen 7-Bit-Wert berechnet und "offset" ein konstanter Ausdruck ist, der fuer einen 16-Bit-Wert berechnet. Die Bezeichnung wird vom Assembler zu einer langen Offset-Adresse erweitert.

4.2.2. Registeradressierung

Bei der Registeradressierungsart ist der Operandwert der Inhalt des General-purpose Registers. Es gibt auf P8000 4 verschiedene Registergroessen:

```
        Wortregister (16 Bits)
        Byteregister ( 8 Bits)
        Registerpaar (32 Bits) und
        Registervierfaches (64 Bits)
```

Ein Wortregister wird durch "R" angezeigt, dass von einer Zahl zwischen 0 und 15 (dezimal) gefolgt wird, die mit den

16 Registern des Apparates korrespondiert. Man kann unter Nutzung der Bytregisterkonstruktion "RH" oder "RL", gefolgt von einer Zahl zwischen 0 und 7, entweder auf das hochwertige und niederwertige Byte der ersten 8 Register Zugriff erhalten. Auf jedes Paar von Wortregistern kann man als ein Registerpaar Zugriff erhalten, wenn man "RR", gefolgt von einer geraden Zahl zwischen 0 und 14, verwendet. Register quadruples entsprechen 4 konsekutiven Wortregistern und sind durch die Bezeichnung "RQ", gefolgt von einer der Zahlen 0, 4, 8 oder 12, zugaenglich. Wird eine merkwuerdige Registerzahl mit einem Registerpaar-Bezeichner gegeben, oder eine andere Zahl als 0, 4, 8 oder 12 wird fuer ein Register quadruple gegeben, hat das einen Assembler-Fehler zur Folge. Im allgemeinen haengt die Groesse eines in eine Operation verwendeten Register von dem bestimmten Befehl ab. Byte-Befehl, die mit dem Suffix "B" enden, werden mit Byte-Registern verwendet. Wortregister werden mit Wortbefehlen verwendet, die keinen Spezialsuffix besitzen. Registerpaare werden mit langen Wortbefehlen verwendet, die mit dem Suffix "L" enden. Registerquadruples werden nur mit 3 Befehlen verwendet (DIVL, EXTSL und MULTL), die einen 64-Bit-Wert verwenden. Ein Assemblerfehler wird gemacht, wenn die Groesse eines Registers nicht richtig mit dem bestimmten Befehl korrespondiert.

```
LD   R5, #%3FFF //Load register 5 with the
                //hexadecimal value 3FFF

LDB  RH3, #%F3  //Load the high order byte of
                //word register 3 with the
                //hexadecimal value F3

ADDL RR2, RR4   //Add the register pairs 2-3 and
                //4-5 and store the result in 2-3

MULTL RQ8, RR12 //Multiply the value in register
                //pair 10-11 (low order 32 bits of
                //register quadruple 8-9-10-11) by
                //the value in register pair 12-13
                //and store the result in register
                //quadruple 8-9-10-11
```

4.2.3. Indirekte Registeadressierung

Bei der indirekten Registeradressierungsart ist der Operandenwert der Inhalt der Lokation, dessen Adresse im spezifizierten Register festgehalten ist. Ein Wortregister wird verwendet, um die Adresse in der Nicht-Segment-Art zu bewahren, hingegen ein Registerpaar in der Segment-Art verwendet werden muss. Es kann jede general-purpose Wortregister (oder Registerpaar im Segmentmode) verwendet werden, ausser R0 oder RR0. Die indirekte Registeradressierungsart wird ebenfalls mit I/O-Befehlen verwendet und zeigt stets eine 16-Bit-I/O-Adresse an. Es

kann jeder general-purpose Register ausser R0 verwendet werden. Eine indirekte Registeradressierung ist durch das Symbol @ entweder von einem Wortregister oder einem Registerpaar-Bezeichner gefolgt, spezifiziert. Fuer die indirekte Registeradressierungsart wird der Wortregister durch ein "R" spezifiziert, das von einer Zahl zwischen 1 und 15 gefolgt wird und ein Registerpaar wird durch "RR" spezifiziert, das von einer geraden Zahl zwischen 2 und 14 gefolgt wird.

```
        JP  @R2          //Pass control (jump) to the
                         //program memory location
                         //addressed by register 2
                         //(non-segmented mode)

        LD  @R3, R2      //Load contents of register
                         //2 into location addressed by
                         //register 3 (non-segmented mode)

        LD  @RR2, #30    //Load immediate decimal value 30
                         //into location addressed by regis-
                         //ter pair 2-3 (segmented mode)

state2: CALL @R3         //Call indirect through register 3
                         //(non-segmented mode)
```

4.2.4. Direkte Adressierung

Der Operandwert, der vom Befehl in der direkten Addressierungsart verwendet wird, ist der Inhalt der Adresse, die durch die Adresse im Befehl spezifiziert ist. Eine direkte Adresse kann als ein symbolischer Name eines Speichers oder I/O Adresse spezifiziert werden oder als ein Ausdruck, der fuer eine Adresse berechnet. Fuer den Nicht-Segment-Mode und alle I/O-Adressen ist die Adresse ein 16-Bit-Wert. Im Segment-Mode ist die Speicheradresse entweder ein 16-Bit-Wert (kurzes Offset) oder ein 32-Bit-Wert (langes Offset). Alle Ausdruecke der Assemblytime Adresse werden unter Verwendung der 32-Bit-Arithmetik berechnet, nur mit den 16 am weitesten rechts stehenden Bits des Ergebnisses, das fuer die Nicht-Segment-Adressen verwendet wird.

```
        LD  R10, datum   //Load the contents of the
                         //location addressed by datum
                         //into register 10

        LD struct+8, R10//Load the contents of register
                         //10 into the location addressed
                         //by adding 8 to struct

        JP  C, %2F00     //Jump to location %2F00 if the
                         //carry flag is set (non-segmented
                         //mode)
```

```
        INB  RH0, 77    //Input the contents of the I/O
                        //location addressed by decimal
                        //77 into RH0

L2::    INC count, #2   //Increment instruction with
                        //direct address "count" and
                        //immediate value 2
```

Fuer Segment-Adressen bildet der Assembler automatisch ein geeignetes Format, das die Segment-Zahl und das Offset einschliesst. Es wird empfohlen, symbolische Namen zu verwenden, wo immer es moeglich ist, da die korrespondierende Segment-Zahl und das Offset fuer den symbolischen Namen durch den Assembler automatisch gehandhabt werden und ihm spaeter Werte zugewiesen werden koennen, wenn das Modul fuer die Ausfuehrung verbunden oder beladen ist. Fuer jene Faelle, da ein spezifisches Segment erwuenscht ist, kann die folgende Bezeichnung verwendet werden (der Segment-Bezeichner steht in winkligen Klammern):

```
        <<segment>>offset
```

wo "segment" ein konstanter Ausdruck ist, der fuer einen 7-Bit-Wert berechnet und "offset" ein konstanter Ausdruck ist, der fuer einen 16-Bit-Wert berechnet. Die Bezeichnung wird durch den Assembler zu einer langen Offset-Adresse erweitert. Um eine kurze Offset-Adresse zu erzwingen, ist ein kurzer Offset-Operator verfuegbar, der nur mit einer direkten Adresse im Segment-Mode verwendet werden kann. Der kurze Offset-Operator ist ein Paar vertikaler Balken "|", das die Adresse einschliesst. Fuer eine gueltige Adresse muss das Offset im Bereich von 0 bis 225 sein; die Zieladresse schliesst die Segmentzahl und ein kurzes Offset in einem 16-Bit-Wert ein.

Ein Beispiel fuer den kurzen Offset-Operator:

```
        .seg            //enter segmented mode
        L1:.word %ABAB  //declare data

        .code           //enter code area
          LD R0, |L1|   //load register 0 from
                        //short address L1
          CP R0, %0D    //compare with %0D
          JP EQ, |L2+10| //jump to short address
                         //L2 + 10
          ADD R0, R2    //add R0 to R2
        L2:  RET        //return
```

4.2.5. Indexierte Adressierung

Eine indizierte Adresse besteht aus einer Speicheradresse, die durch die Inhalte eines angelegten Wortregisters (Index) ersetzt wird. Dieser Ersatz wird der

Speicheradresse hinzugefuegt und die resultierende Adresse verweist auf jene Adresse, dessen Inhalte vom Befehl verwendet werden. Im Nicht-Segment-Mode ist die Speicheradresse als ein Ausdruck spezifiziert, der fuer einen 16-Bit-Wert berechnet. Im Segment-Mode ist die Speicheradresse als ein Ausdruck spezifiziert, der entweder fuer einen 16-Bit-Wert (kurzes Offset-Format) oder zu einem 32-Bit-Wert (langes Offset-Format) berechnet. Alle Assembly-time-Adressausdruecke werden unter Verwendung der 32-Bit-Arithmetik berechnet, wobei nur die 16 am weitesten rechts stehenden Bits des Ergebnisses fuer Nicht-Segment-Adressen verwendet werden. Dieser Adresse folgt der Index, ein in Klammern stehender Wortregister-Bezeichner. Fuer die indizierte Adressierung wird ein Wortregister durch "R" spezifiziert und von einer Zahl zwischen 1 und 15 gefolgt. Alle general-purpose Wortregister, ausser L0, koennen verwendet werden.

```
LD  R10, table(R3)      //Load the contents of the
                        //location addressed by table
                        //plus the contents of reg-
                        //ister 3 into register 10

LD  240+38(R3), R10     //Load the contents of reg-
                        //ister 10 into the location
                        //addressed by 278 plus the
                        //contents of register 3
                        //(non-segmented mode)

ADD  R2, tab(R4)        //Load register 2 with
                        //contents of register 2
                        //added to contents of the
                        //address "tab" indexed
                        //by the value in register 4
```

Fuer Segment-Adressen schafft der Assembler automatisch das geeignete Format fuer die Speicheradresse, die die Segmentzahl und das Offset einschliesst. Wie bei der direkten Adressierung sollten symbolische Namen verwendet werden, wo immer es moeglich ist, so dass Werte spaeter zugewiesen werden koennen, wenn das Modul fuer die Ausfuehrung beladen oder verbunden ist. In jenen Faellen, da ein spezifisches Segment erwuenscht ist, kann die folgende Bezeichnung verwendet werden (der Segment-Bezeichner steht in doppelten winkligen Klammern):

```
<<segment>>offset
```

wo "segment" ein konstanter Ausdruck ist, der fuer einen 7-Bit-Wert berechnet und "offset" ein konstanter Ausdruck ist, der fuer einen 16-Bit-Wert berechnet. Diese Bezeichnung wird durch den Assembler zu einer langen Offset-Adresse erweitert.

4.2.6. Relative Adressierung

Die relative Adressierungsart wird durch ihre Befehle impliziert. Sie wird durch die Call-Relative (CALR), Decrement und Jump if und Zero (DJNZ), Jump Relative (JR), Load Adress Relative (LDAR) und Load Relative (LDR)-Befehle verwendet und ist die einzige Art, die diesen Befehlen zur Verfuegung steht. Der Operand repraesentiert in diesem Falle einen Ersatz, der den Inhalten des Programmzaehlers hinzugefuegt wird, um die Zieladresse zu bilden, die zum gegenwaertigen Befehl relativ ist. Der Original-Inhalt des Programmzaehlers wird genommen, um die Adresse des Befehlsbytes, der dem Befehl folgt, zu sein. Die Groesse und der Bereich des Ersatzes haengt vom spezifischen Befehl ab und wird mit jedem Befehl im P8000-Handbuch beschrieben. Der Ersatzwert kann auf zweierlei Art ausgedrueckt werden. Im ersten Fall liefert der Programmierer einen spezifischen Ersatz in Form von ".+n", wo n ein konstanter Ausdruck im fuer die den spezifischen Befehl bestimmten Bereich ist und "." die Inhalte des Programmzaehler am Anfang des Befehls repraesentiert. Der Assembler subtrahiert die Groesse des relativen Befehls automatisch vom konstanten Ausdruck, um den Ersatz zu gewinnen.

```
JR  OV, .+K //Add value of constant K to program
            //counter and jump to new location if
            //overflow has occurred!

JR  .+4     //Jump relative to program counter "."
            //plus 4
```

Bei der zweiten Methode kalkuliert der Assembler den Ersatz automatisch. Der Programmierer spezifiziert nur einen Ausdruck, der fuer eine Zahl oder eine Programmarke berechnet, wie bei der direkten Adressierung. Die Adresse, die durch den Operand spezifiziert ist, muss fuer den Befehl in einem gueltigen Bereich liegen und der Assembler subtrahiert den Wert der Adresse des folgenden Befehls automatisch, um den tatsaechlichen Ersatz zu gewinnen.

```
DJNZ R5, loop   //Decrement register 5 and jump to
                //loop if the result is not zero

LDR R10, data   //Load the contents of the location
                //addressed by data into register10
```

4.2.7. Basisadressierung

Eine Basisadresse besteht aus einem Register, der die Basis und einen 16-Bit-Ersatz enthaelt. Der Ersatz wird der Basis hinzugefuegt und die resultierende Adresse zeigt die Adresse an, deren Inhalte vom Befehl verwendet werden. Beim Nicht-Segment-Mode wird die Basisadresse in einem Wortregister bewahrt, der durch "R", gefolgt von einer Zahl zwischen 1 und 15, spezifiziert ist. Jeder general-purpose

Wortregister ausser R0, kann verwendet werden. Der Ersatz wird als ein Ausdruck spezifiziert, der fuer einen 16-Bit-Wert berechnet, eingeleitet durch ein "#"-Symbol und in Klammern stehend. Beim Segment-Mode wird die Segment-Basisadresse in einem Wortregisterpaar bewahrt, das durch "RR" gefolgt von eine geraden Zahl zwischen 2 und 14, spezifiziert wird. Jedes general-purpose Registerpaar, ausser RR0, kann verwendet werden. Der Ersatz wird als ein Ausdruck spezifiziert, der fuer einen 16-Bit-Wert berechnet, eingeleitet durch ein "#"-Symbol und in Klammern stehen.

```
LDL     RR2     //Load into register pair 2-3 the
                //long word value found in the
                //location resulting from adding
                //255 to the address in register
                //1  (non-segmented mode)

LD  RR4(#%4000), R2 //Load register 2 into the loca-
                    //tion addressed by adding %4000
                    //to the segmented address found
                    //in register pair 4-5
                    // (segmented mode)

LD  R0, R2(#10) //Load register 0 from 10 bytes
                //past the base address in
                //register 2 (non-segmented mode)
```

4.2.8. Basis-Index-Adressierung

Die Basis-Index-Adressierung entspricht der Basisadressierung, ausser dass der Ersatz (Index) so wie die Basis in einem Register bewahrt werden. Die Inhalte der Register werden zusammenaddiert, um die im Befehl verwendete Adresse zu dekreminieren. Beim Nicht-Segment-Mode wird die Basisadresse in einem Wortregister bewahrt, der durch ein "R", gefolgt von einer Zahl zwischen 1 und 15, spezifiziert ist. Der Index wird in einem Register bewahrt, der in gleicher Weise spezifiziert ist und in Klammern steht. Jeder general-purpuse Wordregister kann sowohl fuer die Basis als auch fuer den Index verwendet werden, ausser R0. Beim Segment-Mode wird die Segment-Basisadresse in einem Registerpaar bewahrt, das durch "RR", gefolgt von einer geraden Zahl zwischen 2 und 14, spezifiziert ist. Jedes general-purpose Registerpaar, ausser RR0, kann verwendet werden. Der Index wird in einem Wortregister bewahrt, der durch "R", gefolgt von einer Zahl zwischen 1 und 15, spezifiziert ist. Jeder general-purpose Wortregister, ausser R0, kann verwendet werden.

```
LD  R3, R8(R15) //Load the value at the location
                //addressed by adding the address
                //in R8 to the displacement in
                //R15 into register 3 (nonseg-
                //mented mode)
```

```
        LDB  RR14(R4),RH2 //Load register RH2 into the
                          //loacation addressed by the
                          //segmented addressed by the
                          //segmented address in RR14
                          //indexed by the value in R4
                          //(segmented mode)

init:   LD R0, R2(R4)     //Load into register 0
                          //the base address
                          //in register 2 indexed
                          //by the value in register 4
                          //(non-segmented mode)
```

4.3. Operatoren fuer Segment-Adressierungsarten

Zwei Spezialoperationen, in Tabelle 4.1 zusammengefasst, erleichtern die Manipulation der Segment-Adressen. Waehrend Adressen als Einzelwert mit einem vom Programmieren zugewiesenen symbolischen Namen behandelt werden koennen, ist es gelegentlich nuetzlich, die Segmentzahl oder das mit dem symbolischen Namen verbundene Offset zu determinieren.

Der eingliedrige Operator "^S" wird bei einem Adressausdruck angewandt, der einen mit einer Adresse verbundenen symbolischen Namen enthaelt und einen 16-Bit Wert zurueckgibt. Dieser Wert ist die 7-Bit-Segmentzahl, die mit dem Ausdruck und einem 1-Bit im signifikantesten Bit des High-Order-Bytes und allen Null-Bits im Low-Order-Byte verbunden ist.

Der "^S"-Operator kann nur im Segment-Mode verwendet werden.

Der eingliedrige Operator "^O" wird bei einem Adressausdruck angewandt und gibt einen 16-Bit-Wert zurueck, der den mit dem Ausdruck verbundenen Offset-Wert darstellt. Der Offset-Operator kann sowohl im Segment- als auch im Nicht-Segment-Mode verwendet werden, hat jedoch im Nicht-Segment-Mode keine Wirkung.

Auf Grund der Spezialeigenschaften dieser Adressoperatoren, koennen keine anderen Operatoren auf einen Subausdruck angewandt werden, der einen Segment- oder Offsetoperator enthaelt, obwohl andere Operatoren im Subausdruck verwendet werden koennen, in dem jeder angewandt werden kann.

```
        .seg              //segmented mode
        L1:.word %ABCD    //declare data

        .code             //enter the code area
        LD R4, #^S L1     //load the segment value
        LD R5, #^O L1     //load the offset  value
        LD R3, @RR4       //load indirect through RR4
```

```
        .nonseg          //non-segmented mode
        L2:.word %BBCC   //declare data

        .code            //enter the code area
        LD R5, #^O L1    //offset operator has
        LD R5, #L1       //no effect; the first
                         //instruction is
                         //equivalent to second
                         //instruction
```

Tabelle 4.1 Segmented Addressing Mode Operators:

Operator		Function
^S	Access	segment portion of address
^O	Access	offset portion of address

4.4. Direktiven fuer die Adressierungsarten

Zwei Direktiven gestatten dem Programmierer zu determinieren, ob der Zusammenstellungsprozess im Segment- oder Nicht-Segment-Mode stattfindet.

```
        .seg
```

weist den Assembler an, die Zusammenstellung im Segment-Mode zu beginnen. Bei default stellt der Assembler im Nicht-Segment-Mode zusammen. Jedes Programm, das eine .seg-Direktive enthaelt, wird als Segment-Programm aufgenommen.

```
        .nonseg
```

weist den Assembler an, zur Zusammenstellung im Non-Segment-Mode zurueckzukehren.

5. Die Programmstruktur

5.1. Einleitung

Die Strukturierung von Programmen und das Konzept der Verschiebbarkeit sind Gegenstand des 5. Teils.

5.2. Module

Ein Assembler-Sprachprogramm besteht aus einem oder mehreren getrennt kodierten und zusammengestellten Modulen (auch Dateien) genannt. Diese Module sind in ausfuehrbaren Programmen unter Verwendung der Modulverkettung und der Verschiebungsmoeglichkeit des operierenden Systems verbunden. Module werden aus Assemblersprachanweisungen gebildet, die Daten definieren oder Handlungen, wie im Teil 3 beschrieben, ausfuehren. Der Assembler erzeugt verschiebbare Objekt-Module. Dieses Merkmal der Verschiebbarkeit des Assemblers befreit den Programmierer von der Betaetigung des Speichers waehrend der Programmentwicklung. Die Verschiebbarkeit wird durch mehrere Direktiven, die unten erlaeutert werden, unterstuetzt, welche mitbestimmen, wo Daten und Handlungsaussagen in den Speicher geladen werden.

5.3. Sektionen und Bereiche

Zusaetzlich zur logischen Strukturierung, geschaffen durch Module, ist es moeglich, das Programm in Sektionen einzuteilen, die in verschiedenen Bereichen des Speichers dargestellt werden koennen, wenn das Modul fuer die Ausfuehrung verbunden oder beladen wird. Der Programmierer z.B. kann beschliessen, einen Satz von Datenstrukturen oder -anweisungen zu gruppieren, um sie gemeinsam am selben Modul zu manipulieren. Es kann aber auch erwuenscht sein, den Objekt-Code fuer die Anweisungen von den Daten in einem System, wo der read-only Speicher fuer die Anweisungen und der read/write Speicher fuer die Daten verwendet wird, physikalisch zu trennen. Jede Sektion koennte einem unterschiedlichen Adressleerzeichen zugewiesen werden. Im Segment-Mode koennte jede Sektion in einem unterschiedlichen Segment dargestellt oder mehrere Sektionen von unterschiedlichen Modulen in demselben Segment verbunden werden. Ein Einzelmodul kann mehrere Sektionen enthalten, von denen jede einem unterschiedlichen Bereich im Speicher zugewiesen wird. Alternativ koennen Teile einer einzelnen Sektion auf mehrere Module verteilt und die Teile automatisch in einem einzelnen Bereich durch den Verketter verbunden werden. Es gibt eine 1-zu-1-Darstellung zwischen Sektionen und Segmenten im Segment-Mode. Im Nicht-Segment-Mode existiert die Faehigkeit dieser 1-zu-1-Darstellung nicht, obwohl die Sektionen den Teilen eines Nutzerprogramms gestatten, logisch gruppiert zu werden, wie beim Segment-Mode. Gegenwaertig koennen der

Code, Daten und bss-Bereiche eines Moduls auf dem P8000 getrennt manipuliert werden. Sind alle Daten in einem Modul in einer Sektion enthalten, ist es moeglich, jene Sektion zur Verbindungszeit zu manipulieren. Die Faehigkeit, Sektionen durch den Namen zu manipulieren, egal welchen Inhalts sie sind, wurde bisher noch nicht ausgefuehrt. Der Assembler gestattet die Spaltung eines Programms in bis zu 3 Sektionstypen: die Programmsektion, die absolute Sektion, die common Sektion. Jede Sektion kann bis zu 3 Bereiche enthalten: einen Code-Bereich, einen Daten-Bereich und einen bss (nichtinitialisierten Datenbereich). Die Code- und Datenbereiche koennen jede legale Assembleranweisung enthalten, bss jedoch kann nur nichtinitialisierte Daten enthalten. Ausserdem ist der Code-Bereich auf 64K begrenzt; die kombinierten Daten- und bss-Bereiche koennen 64K nicht ueberschreiten. Es gibt drei Bereichsassemblerdirektiven:

.code

weist den Assembler an, zum Code-Bereich der gegenwaertigen Sektion zu wechseln.

.data

weist den Assembler an, zum Datenbereich der gegenwaertigen Sektion zu wechseln.

.bss

weist den Assembler an, zum nichtinitialisierten Datenbereich der gegenwaertigen Sektion zu wechseln.

```
        .code           //gib den Code-Bereich der gegen-
                        //waertigen Sektion ein

        .data           //gib den Datenbereich der gegen-
                        //waertigen Sektion ein

        .bss            //gib den bss-Bereich der gegen-
                        //waertigen Sektion ein.
```

5.3.1. Programmsektionen

Eine Programmsektion enthaelt jede legale Assemblersprachanweisung. Jedes Modul muss eine namenlose Sektion enthalten, kann aber auch zusaetzlich bezeichnete Programmsektionen enthalten. Ein Sektionsname besteht aus einem gueltigen Identifizierer. Bei default ist ein Module im Datenbereich der namenlosen Programmsektion am Anfang der Zusammenstellung. Die Assemblerdirektive .psec zeigt zum einen den Anfang einer Programmsektion und gestattet zum anderen den Austausch unter den Programmsektionen.

.psec [<section_name>]

zeigt den Beginn einer Programmsektion an oder weist den Assembler an, zur spezifizierten Programmsektion oder zur default-Programmsektion zu wechseln, falls die Sektion nicht spezifiziert ist. Immer, wenn eine neue Programmsektion eingegeben wird, befindet sich das Modul durch default im Datenbereich jener Sektion. Bei der Rueckkehr zu einer Sektion befindet sich das Modul im zuletzt eingegebenen Bereich jener Sektion. Es folgt ein Beispiel fuer die Verwendung der .psec-Direktive mit den Betriebsdirektiven.

```
        .psec arithmetic//enter the data area (default) of
                        //the program section named
                        //arithmetic

        count: .word 0  //declare a word named count
                        //with initial value 0

        .code           //enter code area of program
                        //section arithmetic

        LD R0, count    //load the count into register 0

        INC R0          //increment the count by 1

        LD store, R0    //load the new count into bss
                        //symbol store

        .bss            //enter the bss area

        store: .word    //word value store

        .psec           //return to the unnamed (default)
                        //program section. The program
                        //returns to the last previously
                        //enter area of the
                        //default program section
```

5.3.2. Absolute Sektionen

Eine absolute Sektion ist eine Sektion, deren Speicherform die absolute Adresse der Sektion im Speicher reflektiert. Da es pro Modul nur eine absolute Sektion geben kann, hat sie keinen Namen.

Absolute Sektionsdirektiven

.asec

weist den Assembler an, zur absoluten Sektion des gegenwaertigen Moduls zu wechseln. Das Modul ist durch default im Datenbereich der Sektion. Beispiele ihrer Verwendung folgen:

```
        .asec           //enter the absolute section
```

```
        .extern proc1, proc2\//define external symbols
                proc3, proc4

        .=10
        jumptab::.addr proc1\ //absolute location 10
                proc2, proc3\ //define jump table
                proc4\        //containing addresses
                              //of external routines
```

5.3.3. Common Sektionen

Common Sektionen gestatten den Verweis auf Sektion desselben Namens in mehreren unterschiedlichen Modulen. Zur Verbindungszeit werden solche Sektionen unter Verwendung der Groesse der groessten Sektion fuer das verschmolzene vereinigt. Ein Common Sektionsname besteht aus einem gueltigen Identifizierer. Common Sektionsdirektiven

```
 .csec <section_name>
```

zeigt den Anfang einer Common Sektion an oder weist den Assembler an, zur spezifizierten Common Sektion zu wechseln. Das folgende Beispiel zeigt die Verwendung von Common Sektionen in 3 unterschiedlichen Modulen:

Module 1, named file1.s, contains:

```
        .csec mycommons
        .=.+10
        L1::.word %FEFE         //a word at location 10
```

Module 2, named file 2.s, contains:

```
        .csec mycommons
        .=.+20
        L2::.word %ABAB         //a word at location 20
```

Module 3, named file3.s, contains:

```
        .csec mycommons
        .=.+10
        L3:.word %CDCD          //a word at location 10
```

Wenn diese 3 Dateien mit dem Kommando

```
        ld -O final file1.0 file2.0 file3.0
```

verbunden werden, entspricht die daraus resultierende common Sektion einem einzelnen Modul, das den folgenden Code enthaelt:

```
        .csec mycommons
        L1::L3:: .word %CDCD    //a word at location 10
        L2::.word %ABAB         //a word at location 20
```

Anmerkung:
Der Wert auf Position 10 in der letzten Datei hat Vorrang vor dem Wert auf Position 10 in der ersten Datei auf Grund der Reihenfolge in der die Dateien in der Kommandozeile des Verketters plaziert sind.

5.4. Lokale Bloecke

Lokale Bloecke gestatten die weitere Strukturierung der Assemblersprachprogramme. Lokale Bloecke stehen zwischen den Symbolen "{" und "}". Sie koennen ineinandergesetzt sein. Die Symbole "{" und "}" muessen die einzigen Zeichen auf der Zeile sein. Lokale Marken in Sektion 3 beschrieben, koennen nur in lokalen Bloecken verwendet werden. Der Anwendungsbereich irgendeines lokalen Symbols ist der am nahesten eingeschlossene lokale Blockdelmiter. Z.B.

```
    {                   //start local block

        ~L1:=20         //declare local constant
        ld r0, #~L1     //reference local constant

    ~L2:ld r2, #%10             //declare local label

        ld r0, #~L2     //reference the local label

        {               //start questet local block
            ~L1:=10     //declare local constant

            ld r0, #~L1 //reference ~L1 equals 10

        }               //end nested local block

        ld r0, #~L1     //reference ~L1 equals 20

    }                   //end local block
```

5.5. Adresszaehler

Der Assembler geht der Adresse der gegenwaertigen Anweisung mit einem Adresszaehler nach, ebenso, wie es ein ausfuehrendes Programm mit seinem Programmzaehler tut. Es gibt einen Adresszaehler, der mit jedem der 3 moeglichen Bereiche einer Sektion verbunden ist: Daten, Code und bss. Der Zaehlwert repraesentiert ein 16-Bit-Offset im gegenwaertigen Bereich. Das Offset kann ein absoluter Wert sein, wenn der Bereich in eine absolute Sektion faellt, oder es kann ein verschiebbarer Wert sein, wenn der Bereich in ein Programm oder eine Common Sektion faellt. Ist es ein absoluter Wert, so reflektiert der Adresszaehler die absolute Speicheradresse der gegenwaertigen Anweisung. Ist es ein verschiebbarer Wert, reflektiert der Adresszaehler das verschiebbare Offset der Anweisung. Das verschiebbare Offset kann zur Verbindungszeit ausgeglichen werden, in

Abhaengigkeit davon, wo die Sektion schliesslich zugewiesen wurde. Das Adresszaehlersymbol "." kann in jedem Ausdruck verwendet werden und repraesentiert die Adresse des ersten Bytes des gegenwaertigen Befehls oder der Direktive.

5.5.1. Die Steuerung des Adresszaehlers

Zwei Assemblerdirektiven ermoeglichen die Steuerung des Adresszaehlers:

.even

Erhoeht den Adresszaehler um 1, wenn er eine ungerade Adresse bewahrt. Das zeigt keine Wirkung, wenn der Adresszaehler eine gerade Adresse bewahrt.

.odd

Erhoeht den Adresszaehler um 1, wenn er eine gerade Adresse bewahrt. Das zeitigt keine Wirkung, wenn der Adresszaehler eine ungerade Adresse bewahrt.

5.5.2. Zeilennummern-Direktive

Durch den Assembler wurde eine zusaetzliche Direktive geliefert:

.line <number> ['"'(filename)'"']

setzt die gegenwaertige Zeilennummer zur spezifizierten Nummer. Ein optionaler Dateiname kann geliefert werden, um den Namen der Datei (Modul), die bearbeitet wird, anzuzeigen. Im Zusammenhang mit der Erfassungsmoeglichkeit des P8000 kann diese Direktive fuer Fehlermeldungen verwendet werden.

Anhang A - Zusammenfassung der Assembler-Direktiven

A.1. Einleitung

Dieser Anhang fasst die Assemblerdirektiven zusammen. Die grammatischen Regeln, die ihrer Verwendung auferlegt sind, werden im Teil 3 beschrieben.

.seg

Weist den Assembler an, die Zusammenstellung im Segment-Mode zu beginnen. Bei default stellt der Assembler im Nicht-Segment- Mode zusammen. Jedes Programm, das eine .seg-Direktive enthaelt, wird als Segment-Programm angenommen.

.nonseg

Weist den Assembler an, die Zusammenstellung im Nicht-Segment-Mode zu beginnen.

.even

Erhoeht den Adresszaehler um 1, wenn er eine ungerade Adresse bewahrt. Das zeitigt keine Wirkung, wenn der Adresszaehler eine gerade Adresse bewahrt.

.odd

Erhoeht den Adresszaehler um 1, wenn er eine gerade Zahl enthaelt. Das zeitigt keine Wirkung, wenn der Adresszaehler eine ungerade Adresse bewahrt.

.line <number> ['"' (filename) '"']

setzt die gegenwaertige Zeilennummer zur spezifizierten Nummer. Ein optionaler Dateiname kann geliefert werden, um den Namen der Datei (Modul), die bearbeitet wird, anzuzeigen. Im Zusammenhang mit der Erfassungsmoeglichkeit des P8000 kann diese Direktive fuer Fehlermeldungen verwendet werden.

.comm <expression> <label>+

Definiert eine Marke als Common Marke.

.extern <label>+

Definiert eine Marke als externe Marke.

.code

Weist den Assembler an, den Codebereich der gegenwaertigen Sektion einzugeben.

.data

Weist den Assembler an, den Datenbereich der gegenwaertigen Sektion einzugeben.

.bss

Weist den Assembler an, den nichtinitialisierten Datenbereich der gegenwaertigen Sektion einzugeben.

.psec [<section_name>]

Weist den Assembler an, zur spezifizierten Sektion zu wechseln.

.csec <section_name>

Weist den Assembler an, zur spezifizierten Common Sektion zu wechseln.

.asec

Weist den Assembler an, zur absoluten Sektion zu wechseln.

.byte(<number>'('<expression>')'|<expression>|'"'string'"')*

Weist den Speicher zu und initialisiert ihn mit dem spezifizierten Byte-Wert(n), der eine Serie von konstanten und verschiebbaren Ausdruecken oder eine ascii-Zeichenkette sein kann. Die Zahl ist der Wiederholungsfaktor. Ist eine Zahl spezifiziert, muss der Ausdruck in Klammern stehen, die Zeichenketten stehen in doppelten Anfuehrungszeichen.

.word (<number> '('<expression>')'|<expression>)*

Weist den Speicher zu und initialisiert ihn mit einem spezifizierten Wortwert(en), der eine Serie von konstanten und verschiebbaren Ausdruecken sein kann. Die Zahl ist der Wiederholungsfaktor. Ist eine Zahl spezifiziert, muss der Ausdruck in Klammern stehen.

.long (<number>'('<expression>')'|<expression>)*

Weist den Speicher zu und initialisiert ihn mit den spezifizierten langen Wert(en), der eine Serie von konstanten und verschiebbaren Ausdruecken sein kann. Die Zahl ist der Wiederholungsfaktor. Ist eine Zahl spezifiziert, muss der Ausdruck in Klammern stehen.

.quad (<number>'('<expression>')'|<expression>)*

Behaelt 64 Bits des Speichers vor. Nur das doppelte Voranstellen der Gleitkommazahl fuellt den zugewiesenen Speicher voellig aus. Fuellt der Wert den zugewiesenen Speicher voellig aus, wird keine Zeichenextension ausgefuehrt. Die Zahl ist der Wiederholungsfaktor. Ist eine Zahl spezifiziert, muss der Ausdruck in Klammern stehen.

.extend (<number> '('<expression>')'|<expression>)*

Behaelt 80 Bits des Speichers vor. Nur das erweiterte Voranstellen von Gleitkommazahlen fuellt den zugewiesenen Speicher voellig aus. Fuellt der Wert den zugewiesenen Speicher nicht voellig aus, erfolgt keine Zeichenextension. Die Zahl ist der Wiederholungsfaktor. Ist eine Zahl spezifiziert, muss der Ausdruck in Klammern stehen.

.addr (<number> '('<expression>')'|<expression>)*

Wenn die Zusammenstellung im Non-Segment-Mode erfolgt, weist es den Speicher zu und initialisiert ihn mit dem spezifizierten 16-Bit-Wert, der eine Serie von konstanten und verschiebbaren Ausdruecken sein kann. Die Zahl ist der Wiederholungsfaktor. Wird die Zahl verwendet, muss der Ausdruck in Klammern stehen. Erfolgt die Zusammenstellung im Segment-Mode, weist es den Speicher zu und initialisiert ihn mit einem 32-Bit-Wert.

.blkb <expression>

Weist Speicher in Bytes zu. Die Zahl der Bytes wird durch den Ausdruck spezifiziert. Es erfolgt keine Initialisierung.

.blkw <expression>

Weist Speicher in Worten zu. Die Zahl der Worte wird durch den Ausdruck spezifiziert. Es erfolgt keine Initialisierung.

.blkl <expression>

Weist Speicher in langen Worten zu. Die Zahl der langen Worte wird durch den Ausdruck spezifiziert. Es erfolgt keine Initialisierung.

Anhang B - Schluesselworte und Sonderzeichen

Schluesselworte

Bestimmte Spezialsymbole sind dem Assembler vorbehalten und koennen durch den Programmierer nicht als Symbole neu definiert werden. Diese sind die Namen der Konditionscode, Registersymbole und Assemblersprachbefehle.

CONDITION CODES

C	LE	NE	PE	UGT
EQ	LT	NOV	PL	ULE
GE	MI	NZ	PO	ULT
GT	NC	OV	UGE	Z

CONTROL REGISTER SYMBOLS

FCW	PSAP
FLAGS	PSAPOFF
NSP	PSAPSEG
NSPOFF	REEFRESH
NSPSEG	

Flag Names

S	C
V	P
Z	VI
NVI	

FLOATING POINT KEYWORDS

AFF	FOP1	INV	OUFLG	RP	USER
CMPFLG	FOP2	INX	PCI	RZ	WARN
DBL	FOV	IN	PCZ	SCON	
DE	INTFLG	IX	PROJ	SGL	
DZ	INVFLG	NAN	RM	SYSFLG	
FFLAGS		NORM			

FLOATING POINT CONDITION CODES

FEQ	FGU	FLU
FGE	FLE	FNEU
FGEU	FLEU	FORD
FGT	FLT	FUN

ASSEMBLY LANGUAGE INSTRUCTIONS

ADC	EI	FMULS	LD	POPL	SLLB
ADCB	EX	FNEG	LDA	PUSH	SLLL
ADD	EXB	FNEGD	LDAR	PUSHL	SOTDR
ADDB	EXTS	FNEGS	LDB	RES	SOTDRB
ADDL	EXTSB	FNORM	LDCTL	RESB	SOTIR
AND	EXTSL	FNORMD	LDCTLB	RESFLG	SOTIRB
ANDB	FABS	FNORMS	LDD	RET	SOUT
BIT	FABSD	FNXM	LDDB	RL	SOUTB
BITB	FABSS	FNXMD	LDDR	RLB	SOUTD
CALL	FADD	FNXMS	LDDRB	RLC	SOUTDB
CALR	FADDD	FNXP	LDI	RLCB	SOUTI
CLR	FADDS	FNXPD	LDIB	RLDB	SOUTIB
CLRB	FCLR	FNXPS	LDIR	RR	SRA
COM	FCP	FREM	LDIRB	RRB	SRAB
COMB	FCPD	FRESFLAG	LDK	RRC	SRAL
COMFLG	FCPF	FRESTRAP	LDL	RRCB	SRL
CP	FCPS	FCSL	LDM	RRDB	SRLB
CPB	FCPX	FSETFLAG	LDPS	SBC	SRLL
CPL	FCPXD	FSETMODE	LDR	SBCB	SUB
CPD	FCPXF	FSETTRAP	LDRB	SC	SUBB
CPDB	FCPZ	FSIGQ	LDRL	SDA	SUBL
CPDR	FCPZX	FSQR	MBIT	SDAB	SWAP
CPDRB	FDIV	FSQRD	MREQ	SDAL	TCC
CPI	FDIVD	FSQRS	MRES	SDL	TCCB
CPIB	FDIVS	FSUB	MSET	SDLB	TEST
CPIR	FEXM	FSUBD	MULT	SDLL	TESTB
CPIRB	FEXPL	FSUBS	MULTL	SET	TESTL
CPSD	FINT	HALT	NEG	SETB	TRDB
CPSDB	FINTD	IN	NEGB	SETFLG	TRDRB
CPSDR	FINTS	INB	NOP	SIN	TRIB
CPSDRB	FLD	INC	OR	SINB	TRIRB
CPSI	FLDBCD	INCB	ORB	SIND	TRTDB
CPSIB	FLDCTL	IND	OTDR	SINDB	TRTDRB
CPSIR	FLDCTLB	INDB	OTDRB	SINDR	TRTIB
CPSIRB	FLDD	INDR	OTIR	SINDRB	TRTIRB
DAB	FLDIL	INDRB	OTIRB	SINI	TSET
DBJNZ	FLDIQ	INI	OUT	SINIB	TSETB
DEC	FLDP	INIB	OUTB	SINIR	XOR
DECB	FLDPD	INIR	OUTD	SINIRB	XORB
DI	FLDPS	INIRB	OUTDB	SLA	
DIV	FLDS	IRET	OUTI	SLAB	
DIVL	FMUL	JP	OUTIB	SLAL	
DJNZ	FMULD	JR	POP	SLL	

Pseudobefehle

JPR CALLR

Bei der Definition von Symbolen muessen Nutzer auch folgende Formen vermeiden:

Rn	, wo n eine Zahl zwischen 0 und 15 ist
RHn oder Rln	, wo n eine Zahl zwischen 0 und 7 ist
RRn	, wo n jede der geraden Zahl zwischen 0 und 14 ist
RQn	, wo n jede Zahl 0, 4, 8, 12 ist
Fn	, wo n eine Zahl zwischen 0 und 7 ist.

Sonderzeichen

Die untenstehende Aufstellung von Sonderzeichen schliesst Delimiter und Spezialsymbole ein. Der Unterschied zwischen ihnen besteht darin, dass Delimiter keine semantische Signifikanz (z.B. koennen 2 Zeichen eine beliebige Anzahl von sie trennenden Leerzeichen haben) besitzen, Spezialsymbole hingegen besitzen semantische Bedeutung (z.B. wird verwendet, um den direkten Wert anzuzeigen). Die Klasse der Delimiter schliesst das Leerzeichen (blank), den Tabulator, die Zeilenspeisung, Carriage Return, Semikolon (;) und Komma (,) ein. Die in den Symbolen /* eingeschlossene Kommentarkonstruktion wird auch als Delimiter angesehen. Die Spezialsymbole und deren Verwendung sind wie folgt:

+	Binary addition; unary plus
-	Binary subtraction; unary minus
*	Unsigned multiplication
/	Unsigned division
^<	Shift left
^>	Shift right
^$	Bitwise and
^|	Bitwise or
^x	Bitwise xor
:	Internal label terminator
::	Global label terminator
~	Local label indicator
:=	Constant and variable initialization
%	Nondecimal number base specifier
#	Immediate data specifier
@	Indirect address specifier
()	Enclose expressions selectively; enclose octal or binary number base indicator; enclose index part of indexed, based, and based indexed address
.	Location counter indicator
//	Begin comment
<< >>	Denotes segmented address

| |	Enclose short offset segmented address
^S	Access segment portion of address
^O	Access offset portion of address
^B	Binary-coded decimal
^C	Ones complement
^FS	Convent to floating single
^FD	Convert to floating double
^F	Convert to floating extended

Anhang C - Fehlernachrichten des Assemblers

Der Anhang C beschreibt die Warn- und Fehlernachrichten des Assemblers.

Warnungen:

Fehler, die Warnnachrichten verursachen, stoeren die Operation des Assemblers nicht, jedoch sollten sie korrigiert werden, bevor die a out Datei ausgefuehrt wird.

Operand too large
Value too large
: Eine zu grosse Zahl fuer einen Datentyp oder ein Befehlsfeld wurde verwendet; z.B.: ".byte %ffff"

Syntaxfehler:

Die meisten Syntax-Fehlernachrichten sind selbsterklaerend; jene die es nicht sind, werden hier aufgelistet. Gibt es mehr als einen Syntaxfehler pro Zeile, wird nur der erste gemeldet. Werden Syntaxfehler entdeckt, wird keine a.out-Datei geschaffen. Schlage den entsprechenden Teil dieses Handbuchs bezueglich der richtigen Syntax nach.

Expecting carriage return or linefeed.
: Extrazeichen (Identifizierer, Ausdruecke, Interpunktion usw.) sind am Ende der Anweisung zu finden. Die gebraeuchlichsten Situationen, wo das erfolgt, wird unten aufgelistet.

  Eine Marke wird von etwas gefolgt, das kein Anweisungsbeginn oder ":=" Operator ist.

  Ein Opcode wird von etwas gefolgt, das kein Operand, keine Gleitkommaabrundung oder Unendlichkeitsmode ist.

  Eine ".psec"-Direktive wird von etwas gefolgt, das kein Identifikator ist.

  Ein ".byte", ".word", ".long" oder ".addr" wird von etwas gefolgt, das kein Adressausdruck oder address expression repeat count ist.

  Kommas fehlen zwischen Operanden, Abrundungs-Modes, Unendlichkeitsmodes, Adressausdruecken in ".byte", "word", ".long" oder ".addr"-Anweisungen, Identifikatoren in ".comm" und ".extern"-Anweisungen oder zwischen dem Ausdruck und dem ersten Identifikator in einer ".comm-Anweisung".

  Es fehlen zweigliedrige Operatoren wie "+", "-", "^X" in Ausdruecken.

  Adressmodes muessen genauso interpunktiert sein, wie im

Teil 4 dieses Handbuchs beschrieben. In bestimmten Faellen ruft eine fehlende linke Klammer diese Fehlernachricht hervor.

Die ".line"-Direktive hat ein optimales Zeichenkettenargument fuer den Dateinamen. Alles andere als eine Zeichenkette nach der Zeilennummer hat diesen Fehler zur Folge.

Expecting beginning of line
Das erste Symbol in der Anweisung ist ein Symbol, das legal keine Anweisung beginnen kann.

Expecting beginning of program
Das erste Symbol in der ersten Zeile des Programms ist ein Symbol, das legal keine Anweisung beginnen kann.

Semantische oder fatale Fehler:

Nur der erste semantische Fehler fuer eine Zeile wird gemeldet. Eine a.out-Datei kann geschaffen werden, doch sie wird korrupt sein. Semantische Fehler sind haeufig mit Syntaxfehlern verbunden. Korrigiere erst die Syntaxfehler.

Block specifier must be constant
Der Ausdruck, der zur Bestimmung der Groesse eines Speicherblocks fuer eine ".blkb", ".blkl", oder ".blkw"-Anweisung verwendet wurde, war verschiebbar.

Bss cannot be initialized
Dieses zeigt an, dass der Programmierer den bss-Bereich mit einer ".bss"-Direktive eingegeben und Befehle oder initialisierte Daten dort placiert hat. Der ".bss"-Bereich kann nur nichtinitialisierten Speicher wie in ".blkb", ".blkl", ".blkw", usw. enthalten.

Invalid assignment
Es wurde der Versuch unternommen, einen Symbolnamen mit einem externen oder nichtdefinierten Wert in einer ":=" direkten Zuweisung zu verbinden.

Invalid constant
Ein Ausdruck, der konstant sein sollte, wurde geschaffen, um verschiebbar oder extern zu sein.

Invalid operand combination
Die Operandenkombination, die mit einem spezifischen Befehl verwendet wurde, war ungueltig. Siehe P8000-Handbuch bezueglich der spezifischen Befehle zur Schaffung der gueltigen Operand-Kombinationen.

Invalid section name
Eine ".psec", ".csec" oder ".asec"-Direktive wurde mit einem Namen verwendet, der bereits woanders als Marke definiert wurde. Sektionsnamen koennen nur mit Sektionsdirektiven verwendet werden.

Invalid token
Es wurde ein ungueltiges Zeichen entdeckt, wie z.B. ein ungeeigneter Identifier oder eine Gleitkommazahl.

Mixed relocatable and absolute
Ein relatives Befehlsziel war absolut, wobei der Befehl in einer verschiebbaren Sektion war oder umgekehrt.

Nesting too deep
Eingenistete Bloecke, durch Klammern "{" und "}" angezeigt, ueberschreiten die gegenwaertig implementierte Einnistungstiefe.

Out of nodes
Der Assembler ging fuer die Initialisierung ueber das Leerzeichen hinaus. Das zeigt gewoehnlich an, dass zu viele Werte in einer Datenanweisung verwendet wurden. Es wird empfohlen, die Anweisung in mehrere Anweisungen zu zerlegen.

Register must be 0-7
Ein Byte-Register wurde mit einer anderen Registernummer als 0-7 verwendet. Nur die ersten 7 Register koennen als Byte-Register verwendet werden.

Symbol redefined
Ein zuvor definiertes Symbol wurde neu definiert.

Segment overflow
Mehr als 64K-Bytes von code, data und bss wurden in einer einzelnen Sektion placiert.

Too many segments
Mehr als 256 Sektionen (z.B., ".psec", ".csec" oder ".asec") wurden definiert.

Undefined symbol
Es wurde auf ein Symbol verwiesen, jedoch keine Definition gefunden. Wird die -u Option des Assemblers verwendet, werden all jene Hinweise extern ohne eine explizite ".extern"-Anweisung.

Unkown Keyword
Es wurde ein mit "." beginnendes Symbol verwendet, jedoch kein Schluesselwort dieses Namens gefunden. Nur Assembler-Schluesselworte duerfen mit "." beginnen.

Both sides of <binary operator> must be constants
Invalid addition (or subtraction) expression
Invalid expression type for ^S (or ^O, or | |) operator
Invalid expression type for left (or right) side
of <binary operator>
Die Regeln fuer verschiebbare, kontante und externe Ausdruecke koennen verletzt worden sein. Auch einige Operatoren unterliegen zusaetzlichen Beschraenkungen (z.B. "^S", "^O") und die Regeln fuer Operatoren

koennen verletzt worden sein.

Bad relocation bits
Cannot determine expression type
Erroneous expression type
Nodes allocated at end of statement
Too many bits assembled for word
Unexpected tag in intermediate file
Unexpected tag in symbol file
Unknown area
Unknown expression type
Unknown scope
Unknown tag in intermediate file
Im allgemeinen sind diese Fehler mit Syntaxfehlern oder vorangegangenen semantischen Fehlern verbunden. Korrigiere zuerst die anderen Fehler, bevor du versuchst, diese zu bestimmen. Erfolgt einer dieser Fehler ohne andere Fehler, kann es ein interner Fehler des Assemblers sein.

Fatale Fehler:

Diese Fehler veranlassen den Assembler zur sofortigen Unterbrechung.

Invalid option
Es wurde eine unbekannte Kommandozeilenoption aufgerufen.

Yacc stack overflow
Zur Zergliederung der Grammatik wurden zuviele Stadien verwendet.

Anhang D - Direktiven zur Unterstuetzung von Debuggern

Die folgenden Direktiven sind zur Unterstuetzung von Debuggern und werden nur von Compilern erzeugt. Sie sind nicht zur Verwendung durch Assembler- sprachprogrammierer gedacht.

.stable <nummer>
Allocate space for source code line number with associated assembly Language code.

.stabn <constant_expr> ',' <constant_expr> ',' <constant_expr> ',' '"' string '"'
Allocate symbol table entry for non-relocatable symbol debug information.

.stabp <constant_expr> ',' <constant_expr> ',' <constant_expr> ',' <constant expr> ',' '"' string '"'
Allocate symbol table for parameter debug information.

.stabr <constant_expr> ',' <constant_expr> ',' '"' string '"'
Allocate symbol table entry for relocatable symbol debug information.

Notizen:

S C R E E N / C U R S E S

Bibliothek zur Bildschirmarbeit

Vorwort

Diese Unterlage beschreibt ein Paket von C-Bibliotheksfunktionen, die es dem Anwender erlauben:

(1) einen Bildschirminhalt mit vernuenftiger Optimierung zu aktualisieren,

(2) eine Terminaleingabe in einer bildschirmorientierten Art zu erhalten und

(3) unabhaengig von den obigen Punkten, den Cursor optimal von einem Punkt zu einem anderen Punkt zu bewegen.

Diese Routinen benutzen alle die Datenbasis /etc/termcap.

Inhaltsverzeichnis Seite

1. SCREEN-Interface-Library

1.1. Einleitung

Eine erfolgreiche Methode zur Vermittlung von Informationen an den Computerbenutzer besteht in der Ausgabe von Displays auf dem Terminal-Bildschirm. Erhaelt der Nutzer kurze Displays, so kann er schnell und leicht antworten. Vergegenwaertigt man sich die Taetigkeiten, die am Computer ablaufen, so gestattet das System sowohl dem Nutzer als auch dem Apparat, effektiver zu werden.

Beim System WEGA ist das am haeufigsten benutzte Bildschirm-Utility der Editor vi. Mit vi werden Textseiten angezeigt, um eine zuegige Herausgabe und Bewegung innerhalb der Datei zu ermoeglichen. Die Flexibilitaet von vi, dem Bildschirm-Editor, wird sofort offensichtlich, wenn man ihn mit dem zeilenorientierten Editor ed. vergleicht. Ausserdem ist vi unabhaengig vom Terminal; er verwendet die Datei /etc/termcap (the terminal capability data base), um sich auf eine Reihe von Terminal-Typen einzustellen.

Die Screen Interface Library ist eine Funktionsbibliothek zur Entwicklung von Software mit dieser bildschirmorientierten oder Display-Herangehensweise. Die Routinen in der Bibliothek verwenden /etc/termcap fuer die Terminal-Unabhaengigkeit und die Curses-Bibliothek (/usr/lib/libcurses.a), wegen ihrer Faehigkeit, den Bildschirm auf den neuesten Stand zu bringen. Siehe Abschnitt 3 Curses in dieser Beschreibung.

Die Screen Interface Library schliesst Routinen zum Setzen des Terminals, die Cursor-Bedienung durch die Nutzung von Pfeiltasten (arrow keys), das Eingeben von einzelnen Zeichen, das Hervorheben des Punktes, auf den der Cursor zeigt, die Seitennumerierung, das Rollen, das Speichern und die Wiederabrufung von Displays, die Ermittlung des Wortes, auf dem der Cursor liegt, die Handhabung von Hilfsdateien und die generelle Handhabung der letzten Zeile, ein. Die individuellen Routinen werden im WEGA-Programmierhandbuch beschrieben. Der Rest der Unterlage beschreibt ihre Anwendung.

1.2. Beschreibung

Die Screen Interface Library liefert dem Programmierer eine Sammlung von Routinen zur Manipulation des Bildschirms. Die Terminaldisplay-Informationen sind Kolumnen von Informationen (Dateien, Namen oder eine Liste), die unter Nutzung von Cursor-Routinen geschaffen wurden.

Die Screen Interface Library wird als eine Erweiterung des Curser-Pakets angesehen, da die Routinen selbst das Cursor-Paket nutzen. Das schliesst ein, dass die Screen

Interface Library terminalunabhaengig ist, d.h., dass ein Aufruf der Curses-Routine, initscr(), fuer ein Bildschirmprogramm gemacht werden muss. Diese Routine findet den Terminaltyp und fuehrt eine Terminal-Initialisierung aus.

Ein anderes Merkmal der Screen Interface Library ist, dass die Routinen das Window-Konzept uebernommen haben, dass im Curses-Paket eingefuehrt wurde. Ein Window wird definiert als eine innere Darstellung, die einen Ausschnitt des Terminal-Bildschirms enthaelt. Die neuen Windows werden durch folgenden Curses-Aufruf definiert:

```
win = newwin(lines, cols, begin_y, begin_x);
```

wo win ein Pointer zur Struktur WINDOW (in /usr/include/curses.h definiert) ist.

Wie im Curses-Paket haben die meisten Screen Interface Routinen 2 Versionen, eine Gesamtbildschirmdarstellung und eine Window-Darstellung; diese sind in /usr/include/screen.h definiert, d.h., wenn die Gesamtbildschirmdarstellung der Routine genutzt wird, so ist der WINDOW-pointer stdscr. Z.B. liefert die Routine getword ein Wort vom ganzen Bildschirm

```
getword(string);
```

und wgetword liefert ein Wort vom Window

```
wgetword(win, string);
```

Hier ist eine Auflistung der Screen Interface Library Routinen:

(1) goraw()
setzt Standard Output auf cbreak mode

(2) gonormal()
setzt das Terminal zurueck in seine normale Stellung

(3) getkey()
liefert eine einzelne Taste, die auf der Tastatur bestaetigt wird

(4) wgetword(win, string)
liefert ein Wort vom Display

(5) wmesg(win, string, data)
Ausgabe einer Nachricht im Standardmode auf der letzten Zeile von win

(6) wmvcursor(win, c, top, bottom)
benutzt das Zeichen c und bewegt den Cursor entsprechend in win

(7) wforword(win, top, bottom)
bewegt sich zum Anfang des naechsten Wortes

(8) wbackword(win, top, bottom)
bewegt sich zum Anfang des vorausgehenden Wortes

(9) wforspace(win, tip, bottom)
bewegt sich zum naechsten Leerzeichen

(10) wbackspace(win, top, bottom)
bewegt sich zum vorausgehenden Leerzeichen

(11) wright(win, top, bottom)
bewegt den Cursor nach rechts

(12) wleft(win, top, bottom)
bewegt den Cursor nach links

(13) whighlight(win, flag)
hebt das Wort an der jetzigen Stelle hervor
oder laesst es zuruecktreten

(14) wsavescrn(win)
rettet den Inhalt von win

(15) wresscrn(win)
ruft ein vorausgegangenes, gespeichertes Window
ab und ueberschreibt win

(16) wscrolf(win)
rollt vorwaerts in win

(17) wscrolb(win)
rollt rueckwaerts in win

(18) wpagefor(win, fp, top)
eine Seite weiter in win unter Nutzung der Datei,
die mit dem Pointer fp verbunden ist

(19) wpageback(win)
eine Seite rueckwaerts in win

(20) wcolon(win)
Handhabung von Punkt-Kommandos (z.B., :q fuer quit)

(21) whelp(win, file)
Handhabung von Help-Kommandos unter Benutzung
der Hilfsdatei file

Eine optionale Moeglichkeit waehrend der Cursorbewegung im Standout mode ist das Hervorheben des Punktes, auf den der Cursor zeigt. Das wird durch das Setzen des globalen flags hilite, zu TRUE erreicht. Beachte auch, dass alle Fehler und informatorischen Nachrichten im Standout mode auf der

letzten Zeile des Windows erscheinen. Eine detaillierte Erlaeuterung fuer jede Screen Interface Library Routine ist im Abschnitt 1.4 enthalten.

1.3. Anforderung und Handhabung

1.3.1. Programmierung

Wenn der Programmierer die Screen Interface Library benutzt, muss er dafuer sorgen, dass die folgenden Zeilen im Grundprogramm enthalten sind:

```
#include <curses.h>
#include <screen.h>
```

Diese header Dateien enthalten die globalen Definitionen und Variablen, die sowohl von der Curses als auch der Screen-Interface-Library verwendet werden. Ausserdem sind die Pseudofunktionen fuer den Standardbildschirm definiert, d.h., dass der Routineparameter win, als Aequivalent der Curses-Variablen, stdscr, angenommen wird. Z.B. ist der Funktionsaufruf

```
help (file)
```

definiert als

```
whelp(stdscr, file)
```

Wenn das Bildschirmprogramm uebersetzt wird, sollte die folgende Kommandozeile verwendet werden:

```
cc [flags] file.c -lscreen -lcurses -ltermlib
```

1.3.2. Normale Beendigung

Bei der normalen Beendigung einer Routine in der Screen Interface Library wird der Wert OK (=1) oder ein spezifischer Wert (OK ist in /usr/include/curses.h definiert) zurueckgegeben. Spezifische Werte (falls es welche gibt), die von jeder Routine zurueckgegeben werden, sind im Abschnitt 1.4. angegeben.

1.3.3. Abnorme Beendigung

Bei der abnormen Beendigung einer Routine in der Screen Interface Library wird der Wert ERR (=0) zurueckgegeben. (ERR ist in /usr/include/curses.h definiert).

1.4. Zusammenfassung der Bibliotheksroutinen

Dieser Abschnitt enthaelt die Beschreibungen der Routinen, die in der Screen Interface Library verfuegbar

sind. Wenn sie anwendbar, so gibt es 2 verfuegbare Aufrufsequenzen (eine fuer Window und eine fuer stdscr), wobei die w-Version der Routine fuer eine Window angewendet werden kann. Die alternative Version der Routine setzt den Standard oder ganzen Bildschirm voraus.

goraw()
Diese Routine setzt standard output auf CBREK mode; ausserdem setzt es das Curses flag; _rawmode auf TRUE (=1).

gonormal()
Diese Routine setzt standard output zurueck zu seinem normalen Mode und _rawmode zurueck zu FALSE (=0).

getkey()
Die Routine liefert ein Zeichen von der Tastatur.

Wenn eine der Pfeiltasten eingetippt wird, so wird die Standarddefinition, die in /usr/include/screen.h zu finden ist, zurueckgegeben. Die folgende Auflistung enthaelt die Festlegungen fuer die Pfeiltasten:

links - h, CTRL-h, backspace
nach unten, - j, CTRL-j
nach oben, -k, CTRL-k
rechts, - l, CTRL-l, space

Im Falle, da keine der Pfeiltasten eingetippt wird, wird das eingetippte Zeichen zurueckgegeben. Ausserdem wird das Zeichen \r zurueckgegeben, wenn carriage return eingetippt wird. Das geschieht, weil einige Terminals ein line feed oder das Zeichen \n fuer den Pfeil nach unten erzeugen; deshalb muss ein Unterschied gemacht werden zwischen der RETURN-Taste (die abgebildet ist to line feed) und der Pfeiltaste nach unten.

```
wgetword (win, str)
WINDOW *win;
char *str;
```

oder

```
getword(str)
char *str;
```

Diese Routine liefert ein Wort vom Display und gibt str aus; wenn das Wort im Display hervorgehoben ist, wird das standout-mode-bit, in jedem Zeichen korrigiert.

```
wmesg(win, str, data)
WINDOW *win;
char *str;
char *data;
```

oder

```
mesg(str, data)
char *str;
char *data;
```

Diese Routine gibt die mittels printf formatierte Nachricht str auf der letzten Zeile des win oder stdscr aus. Es ist hier vermerkt, dass newline oder \n nicht erforderlich ist, da die Nachricht auf der letzten Zeile des Window gegeben wird. Alle zusaetzlichen Daten, die ausgegeben werden sollen (z.B. fuer %s in mesg.), sind in der Variablen data gespeichert. Wenn es keine zusaetzlichen Daten gibt (z.B. str ist eine einfache Nachricht), sollte data NULL enthalten. Nachdem die Nachricht ausgegeben wurde, kehrt der Cursor zur gegenwaertigen Position zurueck.

```
wmvcursor(win, c, top, bottom)
WINDOW *win;
char c;
int top, bottom;
```

oder

```
mvccursor(c, top, bottom)
char c;
int top, bottom;
```

Diese Routine verwendet das gegebene Zeichen c, um den Cursor zweckmaessig auf win zu bewegen innerhalb der Display-Grenzen, vorgegeben durch die obere und untere Zeile. Die gueltigen Werte fuer c (und folglich die gueltigen Cursor-Bewegungen) sind herunter, herauf, vorwaerts (oder Wort) oder rueckwaerts wie in /usr/include/screen.h definiert (siehe Abschnitt 1.5). Wird die untere Grenzlinie ueberschritten, wird der Cursor zum Kopf der rechten oder der am weitesten links stehenden Spalte bewegt; deshalb gibt es einen Cursorumlauf (wraparound). Nachdem die Bewegung ausgefuehrt ist, liefert die Routine OK. Wenn c eine ungueltige Cursorbewegung darstellt, liefert die Routine den Wert ERR. Wenn das globale flag, hilite, gesetzt ist, wird das Hervorheben des Wortes auf der gegenwaertigen Position automatisch ausgefuehrt.

```
wforword(win, top, bottom)
WINDOW *win;
int top, bottom;
```

oder

```
forword(top, bottom)
int top, bottom;
```

Diese Routine bewegt den Cursor zum Anfang des

naechsten Wortes (nach rechts) auf dem Display. Wenn der Cursor sich auf der unteren Zeile der am weitesten rechts stehenden Spalte befindet, wird der Cursor umgelegt an den Kopf der am weitesten links stehenden Spalte.

```
wbackword(win, top, bottom)
WINDOW *win;
int top, bottom;
```

oder

```
backword(top, bottom)
int top, bottom;
```

Diese Routine bewegt den Cursor zum Beginn des vorausgehenden Wortes (nach links) auf dem Display. Wenn sich der Cursor an der am weitesten links stehenden Stelle der oberen Zeile befindet, wird der Cursor auf das letzte Wort der am weitesten rechts stehenden Spalte umgelegt.

```
wforspace(win, bottom)
WINDOW *win;
int top, bottom;
```

oder

```
forspace(top, bottom)
int top, bottom;
```

Diese Routine bewegt den Cursor soweit nach rechts, bis ein Leerzeichen auf dem Display erreicht ist. Wenn sich der Cursor auf der oberen Zeile der am weitesten rechts stehenden Spalte befindet, wird der Cursor umgelegt zum Kopf der am weitesten links stehenden Spalte.

```
wbackspace(win, top, bottom)
WINDOW *win;
int top, bottom;
```

oder

```
backspace(top, bottom)
int top, bottom;
```

Diese Routine bewegt den Cursor auf dem Display soweit nach links, bis ein Leerzeichen erreicht ist. Wenn der Cursor auf der am weitesten links befindlichen Position der oberen Zeile ist, wird der Cursor umgelegt auf das letzte Wort der am weitesten rechts befindlichen Spalte.

```
wright(win, top, bottom)
WINDOW *win;
int top, bottom;
```

oder

```
right(top, bottom)
int top, bottom;
```

Diese Routine bewegt den Cursor eine Stelle nach rechts. Wenn der Cursor auf der unteren Zeile der am weitesten rechts befindlichen Spalte ist, wird er umgelegt an den Kopf der am weitesten links befindlichen Spalte.

```
wleft(win, top, bottom)
WINDOW *win;
int top, bottom;
```

oder

```
left(top, bottom)
int top, bottom;
```

Diese Routine bewegt den Cursor eine Stelle nach links. Wenn der Cursor auf der am weitesten links befindlichen Stelle der oberen Zeile ist, wird der Cursor umgelegt auf das letzte Wort der am weitesten rechts befindlichen Spalte.

```
whighlight(win, flag)
WINDOW *win;
int flag;
```

oder

```
highlight(flag)
```

Diese Routine bringt das Wort an der gegenwaertigen Cursorposition in den standout mode (d.h. das Wort wird hervorgehoben), wenn das flag TRUE ist. Ist das flag FALSE, wird der standout mode fuer das Wort ausgeschaltet.

```
wsavescrn(win)
WINDOW *win;
```

oder

```
savescrn()
```

Diese Routine rettet den Inhalt des win im globalen WINDOW scrn (zu finden in /usr/include/screen.h), der im Speicher der Routine zugewiesen ist. Wenn es mit der Zuweisung Schwierigkeiten gibt, wird ERR zurueckgegeben. Andernfalls wird das globale flag scrnflg zu TRUE gesetzt, der Inhalt von win wird aufbewahrt und die Routine liefert OK.

```
wresscrn(win)
WINDOW *win;
```

oder

```
resscrn()
```

Diese Routine ueberprueft das globale flag, scrnflg (set to TRUE in wsavescrn) und kontrolliert, ob sich der Inhalt des WINDOW scrn dem win anpasst. Sollte das der Fall sein, wird der Inhalt von scrn auf win ueberschrieben und die Routine liefert OK; andernfalls liefert die Routine ERR.

```
wscrolf(win)
WINDOW *win;
```

oder

```
scrolf()
```

Diese Routine rollt win vorwaerts; das wurde bisher noch nicht ausgefuehrt.

```
wscrolb(win)
WINDOW *win;
```

oder

```
scrolb()
```

Diese Routine rollt win rueckwaerts; das wurde bisher noch nicht ausgefuehrt.

```
wpagefor(win, fp, top)
WINDOW *win;
FILE *fp;
int top;
```

oder

```
pagefor(fp, top)
FILE *fp;
int top;
```

Diese Routine gibt eine Seite der Datei heraus, die mit dem Pointer fp. verbunden ist. Wird die Zahl der Zeilen im win ueberschritten, wird das Prompt

```
Type ^f for next page
```

ausgegeben. Wird ^f nicht eingegeben, kehrt die Routine zurueck, andernfalls wird die naechste Datenseite herausgegeben.

```
wpageback(win)
```

```
WINDOW *win;
FILE *fp;
```

oder

```
pageback()
FILE *fp;
```

Diese Routine gibt die vorangegangene Seite der Datei heraus, die mit dem Pointer fp verbunden ist; das wurde bisher noch nicht ausgefuehrt.

```
wcolon(win)
WINDOW *win;
```

oder

```
colon()
```

Diese Routine behandelt die Punktkommandos (der Doppelpunkt wird auf der letzten Zeile von win ausgegeben). Ein Zeichen, das dem carriage return folgt, ist das erwartete typein; die Routine liefert das eingegebene Zeichen.

```
whelp(win, file)
WINDOW *win;
char *file;
```

oder

```
help(file)
char *file;
```

Diese Routine oeffnet die gegebene Hilfsdatei file und gibt deren Inhalt aus. Der urspruengliche Bildschirm, der der Ausgabe folgt, wird wiederhergestellt. Sollte die Hilfsdatei nicht geoeffnet werden koennen oder Schwierigkeiten bei der Wiederherstellung des urspruenglichen Bildschirms auftreten, liefert die Routine ERR, andernfalls wird OK geliefert.

1.5. Programmierinfornmationen

Dieser Teil beinhaltet die Beschreibung der Inhalte der header-Datei /usr/include/screen.h und die in der Screen Interface Library vorhandenen globalen Nachrichten. Hier sind einfache Definitionen der Datei-Bezeichnungen fuer standard-input, standard-output, and standard-error.

```
#define     INFILE  0
#define     OUTFILE 1
#define     ERRFILE 2
```

Das folgende Macro wird von den Bibliotheks-Routinen zur

Interpretation der CTRL-Zeichen verwendet.

```
    #define     CRTL(c)   ('c' & 0x1f)
```

Diese Definitionen sind die Standardtasten, die im Screen-Interface-Paket verwendet werden.

```
    #define     LEFT      'h'
    #define     RIGHT     'l'
    #define     UP        'k'
    #define     DOWN      'j'
    #define     BACKWARD  'b'
    #define     FORWARD   'f'
    #define     WORD      'w'
    #define     COLON     ':'
    #define     HELP      '?'
    #define     PAGEFOR   CTRL(f)
    #define     PAGEBACK  CTRL(b)
```

Die folgende Liste enthaelt die Festlegungen fuer die Funktionen, die stdscr verwenden; diese Definitionen setzen einfach voraus, dass die Variable win stdscr entspricht, wenn Programme fuer den Standard-Terminal-Bildschirm aufgestellt werden.

Anmerkung:
Die Auflistung dieser Aufstellung wurde infolge der Leerzeichen-Begrenzung modifiziert. Siehe /usr/include/screen.h hinsichtlich des Originaltextes.

```
/* pseudo functions for standard screen */

#define mesg(s, d)
           wmesg(stdscr, s, d)
#define mvcursor(c, top, bottom)
           wmvcursor(stdscr, c, top, bottom)
#define forword(top, bottom)
           wforword(stdscr, top, bottom)
#define backword(top, bottom)
           wbackword(stdscr, top, bottom)
#define forspace(top, bottom)
           wforspace(stdscr, top, bottom)
#define backspace(top, bottom)
           wbackspace(stdscr, top, bottom)
#define right(top, bottom)
           wright(stdscr, top, bottom)
#define left(top, bottom)
           wleft(stdscr, top, bottom)
#define highlight(flag)
           whighlight(stdscr, flag)
#define getword(str)
           wgetword(stdscr, str)
#define colon()
           wcolon(stdscr)
#define help(file)
           whelp(stdscr, file)
```

```
#define savescrn()
          wsavescrn(stdscr)
#define resscrn()
          wresscrn(stdscr)
#define scrolf()
          wscrolf(stdscr)
#define scrolb()
          wscrolb(stdscr)
#define pagefor()
          wpagefor(stdscr)
#define pageback()
          wpageback(stdscr)
```

Diese globalen Variablen werden von der Screen Interface Library verwendet. Wenn die Variable hilite gesetzt ist, setzen die Bibliotheks-Routinen voraus, dass der Teil, auf den der Cursor zeigt, im standout mode erscheint. Die Variable scrnflg, wird durch die Routine wsavescrn immer dann gesetzt, wenn dem WINDOW scrn ein Speicher zugewiesen wird. Das Anwenderprogramm kann diese Variable nutzen, um den zugewiesenen Speicher vor Austritt zu loeschen.

```
    int  hilite;
    int  scrnflg;
```

Die Variable scrn wird von den Routinen wsavescrn und wresscrn zur Speicherung und Wiederabrufung des Bildschirms verwendet.

```
    WINDOW  *scrn;
```

Die folgenden globalen Nachrichten werden auch in der Bibliothek definiert. Um diese Nachrichten zu verwenden, ist die Nachrichtenvariable extern zu vereinbaren (z.B. extern char *crmsg).

```
    char  *crmsg = "Type <CR> to continue";
    char  *qmsg  = "Type ?<CR> for help";
```

1.6. Hinweise fuer den Benutzer

Dieser Teil soll eine Richtschnur fuer den Benutzer darstellen, der die Screen Interface Library Routinen verwendet. Das Folgende ist ein Ueberblick darueber, was ein typisches Bildschirmprogramm enthalten soll, das ein Display herausgibt und eine Cursor-Bewegung erlaubt; bei diesem Beispiel wird stdscr vorausgesetzt.

```
#include <curses.h>
#include <screen.h>
#include <signal.h>

        .
        .
        .
```

```
/* top line of the display */
#define  TOP 2

/* name of help file */
#define  HELPFILE "/usr/lib/screen/helpfile"

#define  SOMELENGTH 14

        :
        :

/* bottom line of the display */
int  bottom;

        :

extern  char *qmsg, *crmsg;

        :

main(argc, argv)
int argc;
char **argv;
{
        /* parse options */

                :

        /* set or reset hilite as default */

                :

        /* initialize for the Screen Interface Lib. */
        initialize();

        /* get data for display */
        getdata();

        /* handle keyboard input */
        keyinput();
}

/* basic initialize routine */
initialize()
{
        /* interrupt routine */
        int done();

        /* set up terminal for Curses */
        initscr();

        /* set up interrupt signal */
        signal(SIGINT, done);

        /* go to cbreak mode */
        goraw();
```

```
}

/* this routine merely gets the display information
 * and stores it in the internal buffer */
getdata()
{
        .
        .
        .
}

/* handle keyboard input (in this routine, assume
 * only 'right arrow', ':q' and '?<CR>'
 * are the valid commands) */
keyinput()
{
        char c;
        /* output display */
        dsp();
        move(TOP, 0);
        refresh();

        /* loop until ':q' is typed */
        do
        {
          if(hilite) highlite(TRUE);
          while(TRUE)
          {
            c = getkey();
            if (c == RIGHT || c == COLON || c == HELP)
                break;
            /* if invalid cursor movement,
             * output '?<CR>' message */
            if (mvcursor(c, TOP, bottom) == ERR)
                mesg(qmsg, NULL);
          }
          /* perform command */
          c = cmd(c);
        }
        while (c != 'q');

        /* clean up before exit */
        done();
}

/* output the display */
dsp()
{
        /* set up the screen for the display */
        erase();
        refresh();
        move(0, 0);
        printw("Some Title\n\n");
        refresh();

        /* print out the internal buffer information
         * (either printw or addstr can be used) */
```

```
        printw("here\n");
        printw("there\n");
        printw("everywhere\n");

                :
                :

        /* set up bottom of the display */
        bottom = stdscr->_cury - 1;
        refresh();
}

/* perform command after 'right arrow',
 * ':' or '?' is typed */
cmd(c)
char c;
{
        int  i, j;
        char str[SOMELENGTH];

        /* get the cursored item */
        getword(str);

        /* perform command */
        switch(c)
        {
          case RIGTH:
          /* this code performs the command
           * on the cursored item */
          i = stdscr->_cury;
          j = stdscr->_curx;
          mesg("here it is - type <CR>", NULL);
          while (getkey() != '\r');
                :
                :

          /* redisplay */
          dsp();
          move(i, j);
          break;

          case COLON:
           if (colon() == 'q') return('q');
           break;

          case HELP:
           if (help(HELPFILE) == ERR)
                mesg("Cannot display help file", NULL);
           break;
        }
        return(c);
}
```

```
/* clean up routine */
done()
{
        signal(SIGINT, done);
        if (hilite) highlight(FALSE);
        move(stdscr->_maxy-1, 0);
        refresh();
        if (scrnflg) delwin(scrn);
        gonormal();
        endwin();
        exit(0);
}
```

2. SCREEN-Paket

Mit diesem Paket kann der C-Progammierer die gebraeuchlichsten Typen der terminalunabhaengigen Funktionen ausfuehren: die Bewegungsoptimierung und die optimale Aktualisierung des Bildschirms.

Dieses Paket ist in drei Teile aufgegliedert: (1) die Aktualisierung des Bildschirms, (2) die Aktualisierung des Bildschirms mit Nutzereingabe und (3) die Optimierung der Cursor-Bewegung.

Die Aktualisierung des Bildschirms und die Eingabe koennen ohne jegliche Programmierkenntnisse ueber die Bewegungsoptimierung oder die Datenbasis selbst vorgenommen werden. Die Bewegungsoptimierung kann auch ohne eines der beiden anderen Teile verwendet werden.

In dieser Schrift wird die folgende Terminologie verwendet:

Window: Eine innere Darstellung, die beliebige Abbilder enthaelten kann. Es ist ein Ausschnitt des Terminal-Bildschirms. Diese Unterteilung kann den gesamten Terminal-Bildschirm oder jeden kleineren Teil bis hin zu einem einzelnen Zeichen innerhalb jenes Bildschirms einschliessen.

Terminal: Auch Terminal Screen genannt, ist das gegenwaertige Bild auf dem Bildschirm des Terminals.

Screen: Eine Untermenge von Windows, so gross wie der Terminal-Bildschirm. Eines dieser, stdscr wird fuer den Programmierer automatisch bereitgestellt.

2.1. Benutzung

Um die Bibliothek verwenden zu koennen, ist es notwendig, bestimmte definierte Typen und Variablen zu haben. Deshalb muss der Programmierer

```
#include <curces.h>
```

am Kopf des Grundprogrammes haben. Die header-Datei <curses.h> schliesst <sgtty.h> und <stdio.h> ein. Es ist fuer den Programmierer ueberfluessig (schadet aber nichts), sie im Grundprogramm noch einmal zu definieren.

Der Uebersetzungsaufruf muss folgende Form aufweisen:

```
cc [flags] file ... -lcurses -ltermlib
```

2.1.1. Die Aktualisierung des Bildschirms

Eine Datenstruktur (WINDOW) beschreibt ein Window-

Abbild, einschliesslich dessen Ausgangsposition auf dem Bildschirm (y, x auf der oberen linken Ecke) und dessen Groesse. Eine dieser Strukturen, curscr (current screen) genannt, ist ein Bildschirmabbild von dem, was gegenwaertig auf dem Bildschirm zu sehen ist. Eine andere Struktur, stdscr (standard screen) wird implizit bereitgestellt, um Veraenderungen vornehmen zu koennen.

Ein Window ist eine reine innere Darstellung. Es wird zum Aufbau und zur Speicherung eines moeglichen Bildes eines Teils des Terminal-Bildschirms genutzt. Es steht in keiner notwendigen Beziehung zu dem, was tatsaechlich auf dem Terminal-Bildschirm ist. Es ist mehr ein Feld von Zeichen, in dem man Veraenderungen vornehmen kann.

Wenn ein Window beschreibt, wie ein Teil des Terminals aussehen soll, laesst die Routine refresh() (oder wrefresh()) das Terminal auf der Flaeche, die vom Window eingenommen wird, so aussehen, wie jenes Window. Veraenderungen auf einem Window ziehen keine Terminal-Veraenderungen nach sich. Die eigentliche Aktualisierung erfolgt nur durch das Aufrufen von refresh() oder wrefresh(). Das gestattet dem Programmierer die Behandlung mehrerer verschiedener Vorstellungen vom Aussehen eines Teils des Terminal-Bildschirms. Ausserdem koennen Aenderungen am Window in beliebiger Reihenfolge vorgenommen werden, ungeachtet der Cursorbewegung. Auf Wunsch kann der Programmierer dann sagen: "Lass es wie dieses aussehen" und das Paket waehlt den besten Weg dafuer aus.

Die Routinen koennen mehrere Windows verwenden, doch zwei werden automatisch gegeben. curscr weiss, wie das Terminal aussieht und stdscr weiss, wie der Programmierer das Terminal als naechstes haben will. Dem Benutzer ist curscr niemals direkt zugaenglich. Aenderungen werden am entsprechenden Bildschirm vorgenommen und dann wird die Routine refresh() (oder wrefresh()) aufgerufen.

2.1.2. Konventionen

Viele Funktionen arbeiten mit stdscr als dem impliziten Bildschirm. Um dem stdscr z.B. ein Zeichen hinzuzufuegen, rufe addch() mit dem gewuenschten Zeichen auf. Wenn ein anderes Window verwendet werden soll, wird die Routine waddch() (fuer window-specific addch()) bereitgestellt. Die Routine ist ein "#define" Macro mit Argumenten, die stdscr implizit verwenden. Die Konvention des Voranstellens von Funktionsnamen mit einem "w", wenn sie fuer spezifische Windows gelten, ist konsistent. Die einzigen Routinen, die nicht an dieser Konvention festhalten, sind die, bei denen ein Window spezifiziert werden muss.

Um die gegenwaertigen (y, x) Koordinaten von einem Punkt zum anderen zu bewegen, werden die Routinen move() und

wmove() bereitgestellt. Dennoch ist es oft erwuenscht, erst zu bewegen und dann die I/O-Operation auszufuehren. Den meisten I/O Routine-Bezeichnungen kann der Praefix "mv" vorangestellt sein und die gewuenschten (y, x) Koordinaten werden den Funktionsargumenten hinzugefuegt. Z.B. koennen die Aufrufe

```
move(y, x);
addch(ch);
```

durch

```
mvaddch(y,x,ch);
```

ersetzt werden.

```
wmove (win, y, x);
waddch (win, ch);
```

kann durch

```
mvwaddch (win, y, x, ch);
```

ersetzt werden. Beachte, dass der Window description pointer win, vor den hizugefuegten (y, x) Koordinaten steht. Sollten win pointer benoetigt werden, so stellen sie stets die ersten Parameter dar, die aufgefuehrt werden.

2.1.3. Terminal-Umgebung

Dem Programmierer stehen viele Variablen, die die Terminalumgebung beschreiben, zur Verfuegung. Das sind:

Typ	Name	Beschreibung
WINDOW	*curscr	gegenwaertige Version des Bildschirms (Terminal-Bildschirm)
WINDOW	*stdscr	Standard-Bildschirm, die meisten Aktualisierungen werden gewoehnlich hier vorgenommen
char	*Def_term	default terminal type, wenn der Typ nicht bestimmt werden kann
bool	My_term	verwendet die Terminal-Spezifizierung in Def_term als Terminal, unanwendbar auf den realen Terminal-Typ
char	*ttytype	vollstaendige Bezeichnung des gegenwaertigen Terminals
int	LINES	Anzahl der Zeilen des Terminals
int	COLS	Anzahl der Spalten des Terminals
int	ERR	Error-flag wird zurueckgegeben, wenn Routine misslungen ist.
int	OK	Error-flag, wird zurueckgegeben, wenn Routine erfolgreich war.

Es gibt noch verschiedene andere "#define" Konstanten und Typen, die verwendbar sind:

reg storage class "register" (for example, reg int;)

bool boolean type, actually a "char"
(for example, bool doneit;)

TRUE boolean "true" flag (1)

FALSE boolean "false" flag (0)

2.1.4. Bildschirm-Initialisierung

Um das Screen-Paket verwenden zu koennen, muessen die Routinen ueber die Terminal-Kennwerte Bescheid wissen und Platz fuer curscr und stdscr muss zugewiesen werden. Diese Funktionen werden von initscr() ausgefuehrt. Da es den Windows Platz zuweisen muss, kann es zu einem Ueberlauf des Speicherplatzes kommen. Sollte das eintreten, so gibt initscr() ERR zurueck. Die Routine initscr() muss aufgerufen werden, bevor irgend eine andere die Windows betreffende Routine verwendet wird. Andernfalls wird ein core dump erzeugt, so bald entweder auf curscr oder stdscr zugegriffen wird. Routinen, die den Terminalstatus aendern, wie nl() und crmode(), sollten nach initscr() aufgerufen werden.

2.1.5. Bildschirm-Rollen

Wenn die Bildschirmwindows zugewiesen wurden, koennen sie fuer den Lauf eingerichtet werden. Verwende scrollok(), um dem Window das Rollen zu erlauben. Verwende leaveck() fuer den Cursor, damit er nach der letzten Aenderung dort belassen wird. Andernfalls bewegt refresh den Cursor zu den (y, x) Koordinaten des Window, nachdem er es aktualisiert hat. Neue Windows werden durch newwin() und subwin() erzeugt. Die Routine delwin() befreit von alten Windows. Um die offizielle Groesse des Terminals per Hand zu aendern, setze die Variablen LINES und COLS und rufe dann initscr() auf. Das macht man am besten, bevor initscr() das erste Mal aufgerufen wird. Es kann aber auch ausgefuehrt werden nachdem initscr() existierende stdscr und/oder curscr streicht, oder bevor es neue erzeugt.

2.1.6. Bildschirm-Akualisierung

Die Grundfunktionen zur Aenderung dessen, was auf einem Window ablaeuft, sind addch() und move(). Die Routine addch() fuegt ein Zeichen an den gegewaertigen (y, x) Koordinaten hinzu und gibt ERR zurueck, falls ein unerlaubtes Rollen des Window verursacht wurde. (z.B. Setze ein Zeichen in die untere rechte Ecke eines Terminals ein, das automatisch rollt, falls das Rollen nicht zugelassen ist.)

Die Routine move() veraendert die gegenwaertigen (x, y) Koordinaten. Verursacht move() die Bewegung der Koordinaten aus dem Window heraus, wenn das Rollen nicht zulaessig ist, wird ERR zurueckgegeben. Wie bereits unter 2.1.2. erwaehnt, koennen beide zu mvaddch() verbunden werden, um beides bei einem Funktionsaufruf zu tun. Die anderen Output-Funktionen, wie addstr() und printw() rufen addch() auf, um dem Window Zeichen hinzuzufuegen.

Nachdem das Window in gewuenschter Weise modifiziert wurde, rufe refresh() auf, um es anzuzeigen. Um das Herausfinden der Aenderungen zu optimieren, setzt refresh() voraus, dass kein Teil des Window veraendert wird, bis nicht das letzte refresh jenes Window auf dem Terminal veraendert wurde; d.h., ein Teil des Terminals nicht mit einem ueberlappenden Window erneuert wurde. Andernfalls wird die Routine touchwin() fuer das Hervorrufen einer Veraenderung des gesamten Windows bereitgestellt, um refresh() zu veranlassen, ueberpruefe die gesamte subsection des Terminals auf Veraenderung hin.

Wenn wrefresh() mit curscr aufgerufen wird, wird der gegenwaertige Bildschirm angezeigt. Das ist nuetzlich fuer die Ausfuehrung eines Kommandos zum Neuentwurf des Bildschirms, falls es erforderlich ist.

2.1.7. Bildschirm-Eingabe

Die Eingabe ist im wesentlichen das Spiegelbild der Ausgabe. Die Komplementaerfunktion zu addch() stellt getch() dar, dass, wenn echo gesetzt ist, addch() aufruft, um das ganze Zeichen auszugeben. Sollte das Terminal nicht im raw oder cbreak mode sein, setzt getch() es in den cbreak-mode und liest das Zeichen ein.

2.1.8. Exit-Processing

Um bestimmte Optimierungen vorzunehmen, muessen vor dem Beginn der Bildschirmroutinen einige Dinge getan werden. Diese Funktionen werden in gettmode() und setterm() ausgefuehrt, die von initscr() aufgerufen werden. Die Routine endwin() reinigt nach diesen Routinen. Sie gibt die Terminal-modes wieder so zurueck, wie sie waren, als initscr erstmalig aufgerufen wurde. Folglich muss endwin() nach jedem Aufruf von initscr() vor Austritt aufgerufen werden.

2.1.9. Interne-Beschreibung

Die Cursor-Optimierungsfunktionen dieses Screen Pakets koennen ohne die Screen-Aktualisierungsfunktionen verwendet werden. Die Screen-Aktualisierungsfunktionen werden dort verwendet, wo Teile des Bildschirms veraendert werden, der

Rest jedoch unveraendert bleibt. Fuer grafische Programme, die entworfen wurden, um auf den zeichenorientierten Terminals zu laufen, ist es schwer, diese Funktionen ohne unnoetigen program-overhead zu verwenden. Ein bestimmtes Mass an Vertrautheit mit den Programmierungsproblemen und einigen speziellen Punkten von C sind Voraussetzung fuer das Verstaendnis der folgenden Beschreibung des Geschehens auf den unteren Stufen.

Die /etc/termcap Datenbasis beschreibt die Eigenschaften des Terminals, jedoch ist auch ein bestimmtes Mass an Dechiffrierung notwendig. Der benutzte Algorithmus ist von vi. Er liest die Terminal-Faehigkeiten von /etc/termcap in einer Schleife in eine Serie von Variablen ein, deren Bezeichnungen zwei Grossbuchstaben mit mnemonischem Wert sind. Z.B. stellt HO einen string dar, der den Cursor zur "home" Position bewegt. Siehe Anhang A fuer die vollstaendige Liste und termcap(5) fuer die Gesamtbeschreibung.

Es gibt zwei Routinen zur Handhabung von terminal setup in initscr(). Die erste, gettmode, setzt einige Variablen, die auf terminal modes basieren und durch gtty und stty (siehe ioctl(2)) zugaenglich sind. Die zweite, setterm(), liest in den Beschreibungen von der /etc/termcap Datenbasis. Das folgende Beispiel zeigt, wie diese Routinen verwendet werden:

```
        if (isatty(0))
        (
                gettmode();
                if (sp=getenv("TERM")
                        setterm(sp);
        )
        else
                setterm(Def_term);
        _puts(TI);
        _puts(VS);
```

isatty() bestimmt, ob der Dateidescriptor 0 ein Terminal ist. Es macht einen gtty auf dem descriptor und ueberprueft den return-Wert. Dann setzt gettmode() die terminal modes von einem gtty Aufruf. Die Routine getenv() wird dann aufgerufen, um die Bezeichnung des Terminals zu erhalten. Ein Pointer wird zu einem string, das die Terminalbezeichnung, die im Zeichen-Pointer sp gespeichert ist, zurueckgegeben und zu setterm() gefuehrt. Die Routine setterm() liest dann in den Faehigkeiten, die mit jenem Terminal von /etc/termcap verbunden sind.

Wenn isatty() false zurueckgibt, wird das implizite Terminal Def_term verwendet. Die TI und VS-Sequenzen initialisieren das Terminal durch Aufruf von puts; dieses Macro verwendet tputs() (siehe termlib(3)) zur Ausgabe eines string. Die Routine endwin() macht die vorangegangenen Operationen rueckgaengig.

Das schwierigste ist die richtige Bewegungsoptimierung. Wenn man betrachtet, wie viele unterschiedliche Merkmale verschiedene Terminals aufweisen (tabs, backtabs, non-destructive space, home sequenzes, absolute tabs ...), so kann es zweifellos eine nicht unbedeutende Aufgabe sein, zu entscheiden, wie man von hier nach dort gelangt. Der Editor vi verwendet viele dieser Merkmale und die vom Editor genutzten Routinen nehmen viele Code-Seiten auf. Gluecklicherweise stehen diese Routinen hier zur Verfuegung.

Nach der Nutzung von gettmode() und setterm(), um die Terminal-Beschreibung zu erhalten, befasst sich die Funktion mvcur() mit dieser Aufgabe. Ihre Anwendung ist einfach; sage ihr, wo sie sich jetzt befindet und wohin sie gehen soll. Z.B.:

```
mvcur (0, 0, LINES/2 COLS/2)
```

bewegt den Cursor von der home-Position (0, 0) zur Mitte des Bildschirms. Um eine absolute Adressierung durchzusetzen, verwende die Funktion tgoto() der termlib(3)-Routinen oder mvcur(), wobei der Cursor dann irgendwohin positioniert ist. Z.B., um von irgendwoher absolut an die untere linke Ecke des Bildschirms zu addressieren, behaupte einfach, in der oberen rechten Ecke zu sein;

```
mvcur (0, COLS-1, LINES-1, 0)
```

3. CURSES-Funktionen

In den folgenden Definitionen bedeutet "[*], dass die Funktion tatsaechlich ein "#define" Macro mit Argumenten (in /usr/include/curses.h zu finden) ist. Die Argumente werden angegeben, um Reihenfolge und Typ zu zeigen.

```
addch(ch) [*]
char ch;

waddch(win, ch)
WINDOW *win;
char ch;
```

fuegt das Zeichen ch im Window an den gegenwaertigen (y, x) Koordinaten hinzu. Wenn das Zeichen eine newline ('\n') ist und newline mapping an ist, wird die Zeile bis zum Ende geloescht und die gegenwaertigen (y, x) Koordinaten werden zum Beginn der naechsten Zeile hin geaendert. Wenn newline mapping aus ist, wird die Zeile bis zum Ende geloescht und die Koordinaten veraendert. Ein return ('\r') bewegt zum Beginn der Zeile auf dem Window. Tabs ('\t') werden auf die Leerzeichen in der normalen tabstop-Position jedes achten Zeichens ausgedehnt. Es wird ERR zurueckgegeben, wenn es ein unerlaubtes Rollen des Bildschirms verursachen wuerde.

```
addstr(str) [*]
char *str;

waddrstr(win, str)
WINDOW *win;
char *str;
```

fuegt string, str, im Window an den gegenwaertigen (y, x) Koordinaten hinzu. Es wird ERR zurueckgegeben, wenn es ein unerlaubtes Rollen des Bildschirms verursachen wuerde. In diesem Fall setzt addstr so viel ein, wie es kann.

```
box(win, vert, hor)
WINDOW *win;
char vert, hor;
```

zeichnet einen Rahmen um das Window, vert als Zeichen fuer das Zeichnen der vertikalen Seiten und hor fuer das Zeichnen der horizontalen Zeilen verwendend. Wenn das Rollen nicht erlaubt ist und das Window die untere rechte Ecke des Terminals umfasst, bleiben die Ecken frei um ein Rollen zu verhindern.

```
clear() [*]

wclear(win)
WINDOW *win;
```

fuellt das gesamte Window mit Leerzeichen. Wenn win ein Bildschirm ist, setzt es das clear flag, das das Senden

einer clear-screen Sequenz beim naechsten refresh-Aufruf verursacht. Es bewegt ebenfalls die gegenwaertigen (y, x) Koordinaten nach (0, 0).

```
clearok(scr, boolf) [*]
WINDOW *scr;
bool boolf;
```

setzt fuer den Bildschirm scr das clear flag. Wenn boolf TRUE ist, erzwingt das beim naechsten refresh ein Loeschen des Bildschirms, oder es haelt ihn davon ab, wenn boolf FALSE ist. Das funktioniert nur auf dem Bildschirm und im Gegensatz zu clear aendert es den Inhalt des Bildschirms nicht. Wenn scr curscr ist, verursacht der naechste refresh-Aufruf ein clear-screen, sogar dann, wenn das zu refresh uebergebene Window kein Bildschirm ist.

```
clrtobot() [*]

wclrtobot(win)
WINDOW *win;
```

loescht das Window von den gegenwaertigen (y,x) Koordinaten bis zum untersten Ende. Das erzwingt keine clear-screen Sequenz beim naechsten refresh. Es gibt kein zugehoeriges "mv" Kommando.

```
clrtoeol() [*]

wclrtoeol(win)
WINDOW *win;
```

loescht das Window von den gegenwaertigen (y, x) Koordinaten zum Ende der Zeile. Es gibt kein zugehoeriges "mv" Kommando.

```
cbreak() [*]

nocbreak() [*]

crmode() [*]

nocrmode() [*]
```

setzt das Terminal in oder setzt es zurueck von cbreak mode. Die Makros mit dem ungluecklich gewaehlten Namen crmode und nocrmode dienen der Kompatibilitaet mit frueheren Versionen der Bibliothek.

```
delch() [*]

wdelch(win)
WINDOW *win;
```

streicht das Zeichen auf den gegenwaertigen (y, x) Koordinaten. Jedes danach stehende Zeichen auf der Zeile

rueckt nach links und das letzte Zeichen wird frei.

```
deleteln() [*]

wdeleteln(win)
WINDOW *win;
```

streicht die gegenwaertige Zeile. Jede unter der gegenwaertigen Zeile stehende Zeile rueckt hoch und die unterste Zeile wird frei. Die gegenwaertigen (y, x) Koordinaten bleiben unveraendert.

```
delwin(win)
WINDOW *win;
```

streicht die Existenz des Windows. Alle Resourcen werden fuer die zukuenftige Benutzung von calloc frei gemacht (siehe malloc(3)). Wenn ein Window ein ihm ueber subwin() zugewiesenes Subwindow in sich hat und das aeussere Window getilgt wird, wird das Subwindow nicht veraendert, auch wenn es unwirksam gemacht wird. Deshalb muessen vor ihren aeusseren Windows die Subwindows getilgt werden.

```
echo() [*]

noecho() [*]
```

setzt das Terminal zu echo oder nicht echo der Eingabezeichen.

```
endwin()
```

beendet die Window-Routine vor Austritt und bringt das Terminal wieder auf die Stufe, wo es sich vor initscr() (oder gettmode() und setterm()) befand. Es sollte stets vor Austritt aufgerufen werden. Dies ist besonders wichtig zum Ruecksetzen des Terminalstatus bei Eintreten eines Traps ueber signal(2), da sonst ein unliebsamer Terminalstatus nach Beendigung des Programmes zurueckbleiben kann.

```
erase() [*]

werase(win)
WINDOW *win;
```

loescht das Window, ohne clear flag zu setzen. Das ist analog zu clear(), ausser, dass es niemals eine clear-screen Sequenz bei einem refresh() entstehen laesst. Es gibt kein zugehoeriges "mv" Kommando

```
getch() [*]

wgetch(win)
WINDOW *win;
```

liefert ein Zeichen vom Terminal und gibt es (falls

notwendig) auf dem Window wieder. ERR wird zurueckgegeben, falls es ein unerlaubtes Rollen des Bildschirms verursachen wuerde. Andernfalls wird das Zeichen zurueckgegeben. Wenn noecho gesetzt ist, bleibt das Window unveraendert. Um die Kontrolle ueber das Terminal zu behalten, ist es notwendig, noecho, cbreak oder rawmode gesetzt zu haben. Sollte das nicht der Fall sein, setzt jede aufgerufene Routine zum Lesen von Zeichen cbreak mode und setzt es zurueck in orginial mode, wenn es beendet ist.

```
getstr(str) [*]
char *str;

wgetstr(win, str)
WINDOW *win;
char *str;
```

liefert einen string vom Terminal, gibt ihn auf dem Window wieder und setzt ihn an die von str angezeigte Stelle. Falls erforderlich, setzt es terminal modes und ruft getch (oder wgetch(win)) auf, um das zum Einsetzen in den string benoetigte Zeichen zu erhalten, bis auf ein newline oder EOF gestossen wird. Das newline wird aus string entfernt. Es wird ERR zurueckgegeben, wenn es ein unerlaubtes Rollen des Bildschirms verursachen wuerde. Es wird davon ausgegangen, dass die durch str bezeichnete Stelle in der Lage ist, die eingegebenen Zeichen und ein abschliessendes

```
gettmode()
```

liefert die terminal modes. Das wird normalerweise durch initscr aufgerufen.

```
getyx(win, y, x) [*]
WINDOW *win;
int     y, x;
```

laedt die gegenwaertigen (y, x) Koordinaten des win in die Variablen y und x. Da es ein Macro und keine Funktion ist, kann nicht die Adresse von y und x nicht uebergeben werden.

```
inch() [*]

winch(win) [*]
WINDOW *win;
```

liefert das Zeichen auf den gegenwaertigen (y, x) Koordinaten im gegebenen Window. Das veraendert das Window nicht. Es gibt kein zugehoeriges "mv" Kommando.

```
initscr()
```

initialisiert die Screen-Routinen. Es muss aufgerufen werden, bevor irgendeine andere Routine verwendet wird. Es initialisiert die Terminal-Typ-Daten und keine der Routinen kann ohne es operieren. Wenn standard input kein Terminal

ist, setzt es die Spezifizierung auf das Terminal, auf dessen Bezeichnung durch Def_term (urspruenglich "dumb") gezeigt wird. Falls Boolean My_term true ist, wird Def_term stets verwendet.

```
insch(c) [*]
char c;

winsch(win, c)
WINDOW *win;
```

fuegt c an die gegenwaertigen (y, x) Koordinaten ein. Jedes Zeichen, welches danach in der Zeile steht, wird nach rechts versetzt und das letzte Zeichen verschwindet. Es wird ERR zurueckgegeben, falls es ein unerlaubtes Rollen des Bildschirms verursachen wuerde.

```
insertln() [*]

winsertln(win)
WINDOW *win;
```

fuegt eine Zeile ueber der jetzigen Zeile ein. Jede darunter liegende Zeile wird nach unten versetzt und die unterste Zeile verschwindet. Die gegenwaertige Zeile wird zum Freiraum und die gegenwaertigen (y, x) Koordinaten bleiben unveraendert.

```
leaveok(win, boolf) [*]
WINDOW *win;
bool boolf;
```

setzt das Boolean flag fuer das Belassen des Cursors nach der letzten Aenderung. Ist boolf TRUE, wird der Cursor nach der letzten Aktualisierung auf dem Terminal belassen und die gegenwaertigen (y, x) Koordinaten fuer win entsprechend veraendert. Falls FALSE vorliegt, wird der Cursor zu den gegenwaertigen (y, x) Koordinaten bewegt. Dieses flag (urspruenglich FALSE) behaelt seinen Wert, bis er vom Benutzer geaendert wird.

```
longname(termbuf, name)
char *termbuf, *name;

fullname(termbuf, name)
char *termbuf, *name;
```

setzt den Namen mit der vollen Bezeichnung des Terminals, die von der termcap-Liste beschrieben wird, in termbuf ein. Das steht in der globalen Variablen ttytype zur Verfuegung. Termbuf wird gewoehnlich mittels termlib Routine tgetent() gesetzt. fullname ist das gleiche wie longname, ausser dass der vollstaendigste Name zurueckgegeben wird, der laenger sein kann.

```
move(y, x) [*]
```

```
        int y, x;

        wmove(win, y, x)
        WINDOW *win;
        int y, x;
```

veraendert die gegenwaertigen (y, x) Koordinaten des Windows zu (y, x). Es wird ERR zurueckgegeben, wenn es den Bildschirm zum unerlaubten Rollen veranlassen wuerde.

```
        mvcur(lasty, lastx, newy, newx)
        int lasty, lastx, newy, newx;
```

bewegt den Cursor des Terminals von (lasty, lastx) zu (newy, newx) in einer Annaeherung der optimalen Form. Es ist moeglich, diese Optimierung ohne den Vorteil der Screen-Routinen zu verwenden. Mit den Screen-Routinen, sollte sie nicht vom Benutzer aufgerufen werden. Move und refresh sollten verwendet werden, um die Cursorposition zu bewegen, so dass die Routinen Kenntnisse ueber die Bewegung besitzen. Diese Routine verwendet die Funktionen, die vom ex editor geliehen wurden.

```
        mvwin(win, y, x)
        WINDOW *win;
        int y, x;
```

bewegt die Home-Position des Window win von seinen jetzigen Anfangskoordinaten zu (y, x). Sollte das einen Teil oder das gesamte Window ueber den Rand des Terminal-Bildschirms hinaussetzen, gibt mvwin() ERR zurueck und aendert nichts. Fuer Subwindows gibt mvwin ERR zurueck, wenn versucht wird, es aus dem Hauptwindow herauszubewegen. Wird ein Hauptwindow bewegt, werden alle Subwindows mit verschoben.

```
        nl() [*]

        nonl() [*]
```

setzt das Terminal in den nl-mode oder setzt es vom nl-mode zurueck, d.h., startet/stoppt das System der Umwandlung von <RETURN> zu <LINE-FEED>. Sollte die Umwandlung nicht vollzogen werden, kann refresh eine groessere Optimierung bringen. So wird empfohlen, jedoch nicht verlangt, es auszuschalten.

```
        overlay(win1, win2)
        WINDOW *win1, *win2;
```

legt win1 ueber win2. Der Inhalt von win1 (soviel, wie darauf passen wird) wird auf win2 auf deren Angangs-(y, x)-Koordinaten plaziert. Das erfolgt in einer nicht zerstoerenden Weise, d.h., die Leerzeichen des win1 lassen den Inhalt der entsprechenden Stellen auf win2 unberuehrt.

```
        overwrite(win1, win2)
```

```
WINDOW *win1, *win2;
```

ueberschreibt win1 auf win2. Der Inhalt von win1 (soviel, wie darauf passen wird) wird auf win2 auf deren Anfangs-(y, x)-Koordinaten plaziert. Das erfolgt in einer zerstoerenden Weise (Leerzeichen auf win1 werden zu Leerzeichen auf win2)

```
printw(fmt, arg1, arg2, ...)
char *fmt;

wprintw(win, fmt, arg1, arg2, ...)
WINDOW *win;
char *fmt;
```

fuehrt ein printf auf dem Window, auf den gegenwaertigen (y, x) Koordinaten beginnend, aus. Es verwendet addstr, um das string auf dem Window einzufuegen. Es ist oftmals ratsam, das Feld mit den Optionen von printf zu verwenden, um zu vermeiden, dass irgendetwas aus frueheren Aufrufen im Window verbleibt. Es wird ERR zurueckgegeben, wenn ein unerlaubtes Rollen des Windows verursacht wuerde.

```
raw() [*]

noraw() [*]
```

setzt das Terminal in oder setzt das Terminal zurueck von raw mode. Das schaltet auch newline mapping aus (siehe nl()).

```
refresh() [*]

wrefresh(win)
WINDOW *win;
```

synchronisiert den Terminalbildschirm mit dem erwuenschten Window. Ist das Window kein Bildschirm, wird nur der von ihm eingenommene Teil aktualisiert. Es wird ERR zurueckgegeben, wenn es den Bildschirm zum unerlaubten Rollen veranlassen wuerde. In diesem Fall aktualisiert es alles, was es kann, ohne ein Rollen hervorzurufen.

In einem Spezialfall, wenn wrefresh mit dem Window curscr aufgerufen wird, bewirkt dies ein Loeschen des Bildschirms und dessen Neubeschreibung (Neuaufbau des Bildes). Dies ist sehr nuetzlich, wenn ein anderer Nutzer in das Bild hineingeschrieben hat (z.B. durch write(1)).

```
savetty() [*]

resetty() [*]
```

savetty() speichert das gegenwaertige Terminal-characteristic-Flag. resetty gibt das wieder, was savetty gespeichert hat. Diese Funktione werden von initscr() und

endwind() automatisch ausgefuehrt.

```
scanw(fmtt, arg1, arg2, ...)
char *fmt;

wscanw(win, fmt, arg1, arg2, ...)
WINDOW *win;
char *fmt;
```

fuehrt unter Nutzung von fmt ein scanf vom Window aus. Das geschieht durch die aufeinanderfolgende Verwendung von getch()'s (oder wgetch(win)'s). Es wird ERR zurueckgegeben, wenn es das unerlaubte Rollen des Bildschirms verursachen wuerde.

```
scroll(win)
WINDOW *win;
```

rollt das Window eine Zeile nach oben. Das wird vom Benutzer normalerweise nicht verwendet.

```
scrollok(win, boolf) [*]
WINDOW *win;
bool boolf;
```

setzt das scroll flag fuer das gegebene Window. Wenn boolf FALSE ist, ist das Rollen nicht gestattet. Das ist implizit gesetzt.

```
setterm(name)
char* name;
```

setzt die Terminal-Kennwerte so, um wie die vom Terminal genannte Bezeichnung zu sein. Das wird normalerweise von initscr() aufgerufen.

```
standout() [*]

wstandout(win)
WINDOW *win;

standend() [*]

wstandend(win)
WINDOW *win;
```

startet und stoppt das Setzen von Zeichen im Standout mode auf win. Die Routine standout() verursacht, dass jedes Zeichen, das dem Bildschirm hinzugefuegt wird, im standout mode auf das Terminal (falls es jene Faehigkeit besitzt) gesetzt wird und standend() beendet diesen Vorgang. Die Sequenzen SO und SE (oder US und UE, falls sie nicht definiert sind) werden verwendet (siehe Anhang A).

```
touchwin(win)
WINDOW *win;
```

simuliert, dass jede Stelle auf dem Window veraendert wurde. Das wird gewoehnlich nur fuer refreshes mit ueberlappenden Windows benoetigt.

```
touchline(win, y, startx, endx)
WINDOW *win;
int    y, startx, endx;
```

Diese Funktion ist aehnlich der Funktion touchwin, jedoch fuer eine einzelne Zeile. Sie setzt die first_change_Marke auf startx, wenn startx vor dem letzten echen Austausch liegt, und setzt die last_change_Marke auf endx, wenn diese nach dem letzen Austausch liegt.

```
touchoverlap(win1, win2)
WINDOW *win1, *win2;
```

simuliert eine Veraenderung des Windows win2 in dem Bereich, der mit win1 ueberlappt. Gibt es keine Ueberlappung, wird keine Veraenderung simuliert.

```
WINDOW *

newwin(lines, cols, begin_y, begin_x)
int lines, cols, begin_y, begin_x;
```

gestaltet neue Windows mit lines Zeilen und cols Spalten, auf der Position (begin_y, begin_x) beginnend. Wenn entweder lines oder cols 0(zero) ist, wird jene Dimension zu (LINES-begin_y) oder COLS-begin_x) respektive gesetzt. Verwende demzufolge newwin(0,0,0,0), um ein neues Window mit der Dimension LINES x COLS zu erhalten.

```
WINDOW *

subwin(win, lines, cols, begin_y, begin_x)
subwin(win, lines, cols, begin_y, begin_x)
WINDOW *win;
int lines, cols, begin_y, begin_x;
```

gestaltet ein neues Window mit lines Zeilen und cols Spalten, beginnend auf der Position (begin_y, begin_x) innerhalb des Window win. Jede Veraenderung, die auf einem der Windows auf der vom Subwindow eingenommenen Flaeche vorgenommen wird, wird auf beiden Windows vorgenommen. Die Koordinaten begin_y, begin_x sind spezifizierte relative zum Gesamtbildschirm, nicht die relative (0,0) des win. Wenn entweder lines oder col 0 (zero) ist, wird jene Dimension zu (LINES-begin_y) oder (COLS-begin_x) respektive gesetzt.

```
flushok(win, boolf) [*]
WINDOW *win;
bool   boolf;
```

Normalerweise wird durch refresh ein fflush stdout gemacht.

flushok gestattet dies zu steuern. Wenn boolf TRUE ist (non-zero), wird das fflush gemacht, sonst (FALSE) nicht.

```
idlok(win, boolf)
WINDOW *win;
bool   boolf;
```

Reserviert fuer spaetere Erweiterungen. Dies koennte zum Beispiel dazu benutzt werden, der refresh-Routine anzuzeigen, ob insert und delete line Sequenzen der termcap benutzt werden duerfen.

```
baudrate() [*]
```

gibt die Ausgabebaudrate des Terminals zurueck. Dies ist eine systemabhaenige Konstante.

```
erasechar() [*]
```

gibt das erase-Zeichen des Terminals zurueck, z.B. das Zeichen, das benutzt wird, damit der Nutzer ein Einzelzeichen des Inputs loeschen kann.

```
killchar() [*]
```

gibt analog das Zeilenloeschzeichen zurueck.

```
char *
getcap(str)
char *str;
```

gibt einen Zeiger auf die termcap capability zurueck. (siehe termcap(5)).

Anhang A: Die termcap-Variablen

Das folgende stellt nur eine Zusammenfassung der Faehigkeiten dar. Die Gesamtbeschreibung der Terminals betreffend, siehe termcap(5).

Die Faehigkeiten vom termcap sind von 3 Arten: string valued options, numeric valued option und Boolean options. Die string valued options sind die kompliziertesten, da sie ueberfluessige Informationen enthalten.

Intelligente Terminals erfordern oftmals das Erfuellen von intelligenten Operationen in einer hohen (und manchmal sogar in einer niedrigen) Geschwindigkeit. Das wird durch eine Zahl vor dem string in der Faehigkeit spezifiziert und besitzt Bedeutung fuer die Faehigkeiten, die ein P vor ihrer Erklaerung haben. Das ist normalerweise eine Zahl von Millisekunden, um die Operation zu erfuellen. Im gegenwaertigen System, das keine genau programmierbaren Verzoegerungen besitzt, wird das durch das Aussenden einer Sequenz von pad-Zeichen (normalerweise Nullen, die jedoch durch die Spezifizierung durch PC veraendert werden koennen) getan. In einigen Faellen wird pad besser berechnet, als gewisse Anzahl von Millisekunden fuer eine Operation pro Zeile. Das wird als "12*" vor der Faehigkeit spezifiziert, d.h. 12 Millisekunden pro Zeile. Faehigkeiten, bei denen das sinnvoll ist, werden mit "P*" gekennzeichnet.

A.1. Variablen gesetzt durch setterm()

Type	Name	Pad	Description
char *	AL	P*	Add new blank Line
bool	AM		Automatic Margins
char	BC		Back Cursor movement
bool	BS		BackSpace works
char *	BT	P*	Back Tab
bool	CA		Cursor Adressable
char *	CD	P*	Clear to and of Display
char *	CE	P	Clear to End of line
char *	CL	P*	Clear screen
char *	CM	P	Cursor Motion
char *	DC	P*	Delete Character
char *	DL	P*	Delete Line sequence
char *	DM		Delete Mode (enter)
char *	DO		Down line sequence
char *	ED		End Delete mode
bool *	EO		can Erase Overstrikes with
char *	EI		End Insert mode
char *	HO		HOme cursor
bool	HZ		HaZeltine ~ braindamage
char *	IC	P	Insert Character
bool	IN		Insert-Null blessing

char *	IM		enter Insert Mode (IC usually set, too)
char *	IP	P*	Pad after char Inserted using IM+IE
char *	LL		quick to Last Line, column 0
char *	MA		ctrl character MAp for cmmd mode
bool	Mi		can Move in Insert mode
bool	NC		No Cr: \r sends \r then eats 0
char *	ND		Non Destructive space
bool	OS		Over Strike works
char	PC		Pad Character
char *	SE		Standout End (may leave space)
char *	SF	P	Scroll Forwards
char *	SD		Stand Out begin (may leave space)
char *	SR	P	Scroll in Reserve
char *	TA	P	Tab (not I or with padding)
char *	TE		Terminal address enable Ending sequence
char *	TI		Terminal address enable Initialization
char *	UC		Underline a single Character
char *	UE		Underline Ending sequence
bool	UL		Underlining works even though !OS
char *	UP		UPline
char *	US		Underline Starting sequence
char *	VB		Visible Bell
char *	VE		Visual End sequence
bool	XN		a Newline gets eaten after wrap

A.2. Variablen gesetzt von gettmode()

Typ	Bezeichnung	Beschreibung
bool	NONL	Term can't hack linefeeds donig a CR
bool	GT	Gtty zeigt Tabs an
bool	UPPERCASE	Terminal erzeugt nur Grossbuchstaben

Sollten US und UE in der termcap-Liste nicht existieren, werden sie von SO und SE im setterm() kopiert. Bezeichnungen, die mit einem X beginnen, sind fuer ungewoehnliche Umstaende vorgemerkt.

Anhang B: Die WINDOW-Struktur

Die Window-Struktur ist wie folgt definiert:

```
struct _win_st {
            short               _cury,_curx;
            short               _maxy,_maxx;
            short               _begy,_begx;
            short               _flags;
            short               _ch_off;
            bool                _clear;
            bool                _leave;
            bool                _scroll;
            char                **_y;
            short               *_firstch;
            short               *_lastch;
            struct _win_st      *_nextp, *_orig;
);

#define WINDOW  struct _win_st
```

Alle Variablen, die mit dem _-Zeichen beginnen, sollten niemals direkt vom Nutzer beeinflusst werden !!!

_cury und _curx sind die gegenwaertigen (y, x) Koordinaten fuer das Window. Neue Zeichen, die dem Bildschirm hinzugefuegt werden, werden an diesem Punkt eingesetzt.

_maxy und _maxx sind die Maximalwerte, die fuer (_cury, _curx) zulaessig sind.

_begy und _begx sind die Anfangs-(y, x)-Koordinaten auf dem Terminal fuer das Window, d.h. das home des Windows.

Beachte, dass _cury, _curx, _maxy und _maxx durch (_begy, _begx) relativ begrenzt sind, nicht das Home des Terminals.

Flags koennen einen oder mehrere der folgenden Werte besitzen:

```
        #define _ENDLINE        001
        #define _FULLWIN        002
        #define _SCROLLWIN      004
        #define _FLUSH          010
        #define _FULLLINE       020
        #define _IDLINE         040
        #define _STANDOUT       0200
```

_ENDLINE sagt aus, dass das Ende der Zeile fuer dieses Window gleichzeitig das Ende eines Bildschirms ist.

_FULLWIN sagt aus, dass dieses Window ein Bildschirm ist.

_SCROLLWIN zeigt an, dass das letzte Zeichen dieses Bildschirms in der unteren rechten Ecke des Terminals ist;

d.h., wenn das Zeichen dorthin gesetzt wird, wird das Terminal rollen.

_FLUSH bewirkt ein fflush stdout am Ende jedes refresh(), wenn es TRUE ist.

_IDLINE ist reserviert fuer spaetere Benutzung und wird durch idlock() gesetzt.

_STANDOUT sagt aus, dass alle hinzugefuegten Zeichen im Standout mode erscheinen.

_ch_off ist fuer alle Subwindows der x-Offset in den _firstch und _lastch Arrays fuer dieses Window. Fuer Mainwindows ist dieser immer 0. Fuer Subwindows ist es die Differenz zwischen dem Startpunkt des Mainwindows und dem des Subwindows. Damit koennen die change_marks relativ zum Mainwindow sein. Dadurch sind diese markers im Bereich global.

_clear sagt aus, ob eine clear-screen-Sequenz nach dem naechsten refresh()-Aufruf entstehen soll. Das ist nur fuer den Bildschirm von Bedeutung. Der Anfangs-clear-screen fuer den ersten refresh Aufruf entsteht durch das anfaengliche Setzen von clear to be TRUE for curscr, der stets ein clear-screen, falls gesetzt, erzeugt, irrelevant fuer die Dimensionen der enthaltenen Windows.

_leave ist TRUE, wenn die gegenwaertigen (y, x) Koordinaten und der Cursor belassen werden sollen, nachdem sich das letzte Zeichen auf dem Terminal geaendert hat oder sich nicht bewegte, wenn es keine Aenderung gibt.

_scroll ist TRUE, wenn Rollen erlaubt ist.

_y ist ein Pointer zu einem Array von Zeilen, die beshreiben, wie das Window auf dem Terminal darzustellen ist. Folglich ist _y[i] ein Pointer zur i-ten Zeile.

_firstch und _lastch sind Zeiger auf durch malloc bereitgestellt Arrays. _firstch repraesentiert die Position des ersten Zeichens in der Zeile, das waehrend eines refresh() veraendert werden soll. Diese Position ist fuer die i-te Zeile in _firstch[i] gespeichert. _lastch repraesentiert die Position des letzten Zeichens in einer Zeile, das waehrend eines refresh() veraendert werden soll. Diese Position ist fuer die i-te Zeile in _lastch[i] gespeichert.

```
        #define _NOCHANGE       -1
```

_NOCHANGE ist gesetzt in _firstch fuer jede Zeile des Windows, in der seit dem letzten refresh() keine Aenderungen gemacht wurden.

_orig ist ein Zeiger, der fuer Windows, die kein Subwindow

sind, NULL enthaelt. Fuer Subwindows zeigt er auf das Mainwindow, in welchem es enthalten ist.

_nextp ist ein Zeiger in einer umlaufenden Verbindungsliste aller Windows, die Subwindow des gleichen Mainwindows sind, und dem Mainwindow selbst.

Notizen:

U U C P

Implementierung

Vorwort

Das Programmpaket UUCP ist ein Kommunikationssoftwarepaket, das die Dateiuebertragung und die entfernte Kommandoausfuehrung zwischen zwei Wega-Systemen bzw. anderen kompatiblen Systemen unterstuetzt. Die Kopplung erfolgt ueber die Terminalschnittstellen. Diese Unterlagen stellen vor allem eine Anleitung fuer die Implementierung dieses Programmpaketes dar. Darueberhinaus koennen aber auch einige Abschnitte fuer den Nutzer von Interesse sein. So sind in Abschnitt 2 die Kommandos beschrieben. Diese Beschreibungen gehen ueber die des Programmierhandbuchs hinaus und sind mit zahlreichen Beispielen versehen.

Inhaltsverzeichnis Seite

1. Allgemeines

Das Programmpaket UUCP unterstuetzt die Kopplung von Rechnern unter WEGA bzw. anderen kompatiblen Betriebssystemen. Es ist damit die Uebertragung von Dateien, die entfernte Kommandoausfuehrung und die Nachrichtenuebertragung moeglich. Dieses Programmpaket erfordert keine Veraenderungen des Betriebssystems. Es verhaelt sich aehnlich anderen Benutzern, die sich an das System ueber login-Ports anschliessen und alle lokalen Schutzregeln befolgen.

Der zugrundeliegende Betrieb des Netzes ist sehr einfach. Jedes teilnehmende System hat ein Spool-Directory, in der die auszufuehrende Arbeit gespeichert wird (Dateien sind zu kopieren oder Kommandos sind auf dem entfernten System auszufuehren).

Das Standardprogramm uucico bewerkstelligt den eigentlichen Transfer. Das Programm startet, indem es einen bestimmten Uebertragungskanal zu einer fernen Anlage auswaehlt, mit dem es einen Austausch ausfuehren will. Uucico sucht dann den entsprechenden Anschluss, stellt die Verbindung her, meldet sich am entfernten System an und startet dort das uucico-Programm, um einen Kommunikationspartner zu haben.

Sind zwei dieser Programme verbunden, vereinbaren sie zuerst ein Protokoll und beginnen dann Daten auszutauschen. Jedes Programm, beginnend mit dem aufrufenden, arbeitet alle Auftraege aus dem Spool-Directory ab. Wenn keine Arbeit mehr vorhanden ist, wird die Verbindung abgebrochen.

2. Kommandoarbeit

Die Benutzung des Programmkomlpexes UUCP erfolgt mittels der Kommandos uucp, uux, uusend, uulog, uuclean, uuname und uucico, deren Kurzbeschreibungen im "WEGA-Programmierhandbuch" enthalten sind. Nachfolgend erfolgt eine ausfuehrliche Beschreibung der Kommandos uucp, uux und uucico, die das Hauptinterface fuer den Nutzer bilden.

2.1. Kommando uucp

Mit dem Kommando uucp wird die Dateiuebertragung von/zu einem entfernten Rechner angewiesen. Im Ergebnis der Abarbeitung werden Kommando- bzw. Daten-Files in das Spool-Directory des jeweiligen Rechners eingetragen und - wenn nicht die -r Option angegeben ist - das Programm uucico zur Durchfuehrung der Datenuebertragung gestartet.

Kommandoformat:

uucp [-r] [-d] [-c] [-m] [-g] [-s] [-x] Quell-File Ziel-File

Ein File-Name kann ein Pfadname auf der lokalen Maschine sein oder die Form

Systemname!Pfadname

haben. Wenn als Kommandointerpreter die C-Shell benutzt wird, dann muss dem "!" ein "\" vorangestellt werden. Der angegebene Systemname muss in der Liste der Systemnamen (USERFILE,L.sys) verzeichnet sein. Pfadnamen koennen volle Pfadnamen oder eine beliebige, nicht mit "/" beginnende Bezeichnung sein. Im letzteren Fall dehnt UUCP diesen Pfadnamen aus, d.h. es wird der aktuelle Directory-Name vorangestellt. Im Normalfall ist dieser Directory-Name der Name des Arbeits-Directorys im lokalen System. Wird der Pfadname in der Form ~user angegeben, wird das Login-Directory des Nutzers user fuer die Pfadnamenausdehnung genutzt. Diese Ausdehnung erfolgt an dem System, an dem die entsprechende Datei gespeichert ist. Shell-Sonderzeichen, die im Pfadnamen fuer ein entferntes System enthalten sind, muessen in ' ' eingeschlossen sein bzw. es muss ein "\" vorangestellt sein, da die Substitution in diesem Fall erst am entfernten System erfolgt.

Optionen:

-r Verhinderung des automatischen Starts des Programms uucico. Es wird nur die Eintragung der Auftraege in das Spool-Directory vorgenommen.

-d Errichtet evtl. notwendige Directories fuer den Dateitransfer

-c Statt des im Standardfall ueblichen Kopierens der Quelldatei in das Spool-Directory wird fuer die Uebertragung die Quelldatei direkt benutzt.

-m Die erfolgreiche Abarbeitung des Auftrages wird durch eine mail-Mitteilung verzeichnet (in Ablehnungsfaellen wird immer eine mail-Eintragung vorgenommen).

-gletter In den Namen des C-Files wird eine Mitteilung der Form "letter" eingefuegt. Die Laenge von "letter" darf 1 Zeichen betragen. Diese Mitteilung kann vom Nutzer bei der Abarbeitung des Auftrages ausgewertet werden.

-xnum Angabe der gewuenschten Debugging-Ebene (num=4 Kurzprotokoll, num=8 ausfuehrliches Protokoll). Ausgegeben werden nur Informationen, die die Analyse der Kommandozeile und das Errichten der Eintragungen im Spool-Directory betreffen, nicht jedoch das Protokoll der Uebertragung.

-sdir UUCP benutzt statt des standardmaessigen /usr/spool/uucp das Directory "dir" als Spool-Directory, um dort die Uebertragungsauftraege einzutragen. Das LOGFILE wird dabei trotzdem in /usr/spool/uucp errichtet. Hierbei ist zu beachten, dass das mit "dir" angegebene Directory auf dem gleichen physischen Geraet liegen muss wie das urspruengliche Spool-Directory. Eine automatische Abarbeitung der somit im Directory "dir" eingetragenen Auftraege ist nicht moeglich, da das automatisch von uucp gestartete uucico nur das urspruengliche Spool-Directory nach Arbeit absucht. Um diese Auftraege auszufuehren, muss uucico getrennt und mit der "-d" Option (unter Angabe des gewuenschten Spool- Directory) gestartet werden (siehe Beispiel).

-esys Sendet das uucp Kommando zum System "sys", wo es ausgefuehrt wird (Beachte - die Ausfuehrung ist nur dann erfolgreich, wenn die abgesetzte Maschine es erlaubt, das uucp Kommando durch /usr/lib/uucp/uuxqt auszufuehren. Das ist in /usr/lib/uucp/L-cmd festgelegt.)

Fehlerbehandlung:

Fehlerhafte Eingaben (wie z.B. fehlerhafte Parameterangaben, nicht existierende Systemnamen bzw. nicht vorhandene Dateien im lokkalen Rechner) werden durch Fehlerausschriften auf dem Bildschirm abgewiesen. Bei nicht zulaessigen Systemnamen, nicht vorhandenen Ausgangsdateien und einer syntaktisch falschen Kommandoangabe erfolgt keine Eintragung in das Spool-Directory. Bei Angabe einer falschen Option erfolgt zwar eine entsprechende Ausschrift auf

dem Terminal, das Kommando wird aber (bei Ignorierung der falschen Option) trotzdem ausgefuehrt. Die ordnungsgemaesse Eintragung bzw. Abarbeitung des Auftrages wird im LOGFILE verzeichnet. Treten waehrend der Uebertragung nicht behebbare Fehler auf, so wird ein entsprechender Eintrag im LOGFILE vorgenommen (s.Abschn.4 Diagnoseunterstuetzung).

Beispiele zur Angabe der Pfadnamen:

- uucp /z/meier/dat WEGA_A!/z/otto

  Die Datei dat aus dem Directory /z/meier des lokalen Rechners wird zum Rechner WEGA_A gesendet und in das Directory /z/otto geschrieben.

- uucp /z/meier/* WEGA_A!/z/otto

  Alle Dateien aus dem Directory /z/meier des lokalen Rechners werden in das Directory /z/otto des Systems WEGA_A geschrieben.

- uucp WEGA_B!/z/src/'*' /z/src

  Alle Dateien aus dem Directory /z/src des Rechners WEGA_B werden in das Directory /z/src des lokalen Rechners kopiert. Hier ist die Angabe der '' notwendig!

- cd /z/doc
  uucp text.15 WEGA_C!/z/otto

  UUCP stellt dem Namen text.15 den Pfadnamen des aktuellen Directory (/z/dok) voran und uebertraegt die Datei in das Directory /z/otto des Systems WEGA_C.

- uucp dat WEGA_A!~otto/text

  Die Datei dat aus dem aktuellen Directory wird am Rechner WEGA_A in das Directory text im Logindirectory (entsprechend Passwordeintrag) des Nutzers otto geschrieben. Die Ausdehnung des Pfadnamens fuer ~otto/text erfolgt erst am System WEGA_A.

Beispiele zur Interpretation des letzten Teils des Pfadnamens (Sinnfaelligkeit der d- Option)

Voraussetzung fuer die nachfolgenden Beispiele: Am Rechner WEGA_A existiert ein Directory /z/otto ,das leer ist.

- uucp /z/meier/data WEGA_A!/z/otto

  Die Datei data wird in das Directory /z/otto des Rechners WEGA_A uebertragen, unter dem urspruenglichen Namen data.

- uucp /z/meier/data WEGA_A!/z/otto/sub

  Wenn der letzte Teil des angegebenen Pfadnamens des Zielortes physisch nicht vorhanden ist, nimmt UUCP stets an, dass das der gewuenschte Name der Ziel-Datei ist und nicht eines Directorys, unter dem die Datei stehen soll.

  Die Datei data wird in das Directory /z/otto des Rechners WEGA_A uertragen, wo es den Namen sub erhaelt.

- uucp /z/meier/data WEGA_A!/z/otto/sub/data

  Vom Rechner WEGA_A kommt eine Ablehnung (SN2), da das Directory /z/otto/sub nicht existiert

- uucp -d z/meier/data WEGA_A!/z/otto/sub/data

  Die Datei data vom lokalen Rechner wird in das Directory /z/otto/sub des Rechners WEGA_A geschrieben (unter dem Namen data), wobei das vorher nicht vorhandene Directory /z/otto/sub errichtet wurde.

Beispiel fuer die Benutzung der "-s" Option

- uucp -r -s/z/new /z/meier/data remote!/z/otto
  uucico -r1 -d/z/new

Beispiel fuer die Benutzung der "-e" Option

Da nur eine begrenzte Anzahl von Schnittstellen zur Verfuegung steht, kann es bei der Vernetzung mehrerer Rechner notwendig werden, eine Netzarchitektur aufzubauen, bei der nicht jeder Rechner mit jedem anderen verbunden ist. Mit der Option "-e" ist es moeglich, Dateien ueber zwei Rechner zu uebertragen. Dabei ist jedoch zu beruecksichtigen, dass mehrfach Verbindungen aufgebaut werden muessen und es deshalb sehr zeitaufwendig ist.

- uucp -eWEGA_A WEGA_B!/z/otto/dat /z/meier

  Vorraussetzung: Es muss eine Verbindung vom lokalen Rechner zum System WEGA_A und von System WEGA_A zum System WEGA_B bestehen. Das lokale System und das System WEGA_B muessen nicht miteinander verbunden sein. Die Datei /z/otto/dat vom System WEGA_B wird zum lokalen System in das Directory /z/meier kopiert. Der Datenaustausch geht ueber den Rechner WEGA_A. Dazu wird auf dem Rechner ein uucp-Kommando ausgefuehrt, dass die Daten von System WEGA_B nach WEGA_A kopiert und ein weiteres, dass die Daten zum lokalen System sendet.

Eine weitere Moeglichkeit zur Dateiuebertragung ueber mehrere Rechner hinweg bietet das Programm uusend (siehe uusend.1).

2.2. Kommando uux

Das Kommando uux wird fuer die Anweisung einer Kommandoausfuehrung angewendet, wenn eine oder mehrere Komponenten der Kommandokette (Kommandos oder Dateien) sich auf einen entfernten Rechner beziehen. Bei Abarbeitung von uux werden die fuer die Kommandoausfuehrung der angewiesenen Kommandos benoetigten Dateien auf dem entsprechenden Rechner gesammelt und anschliessend das Programm UUXQT gestartet, das die Kommandoausfuehrung uebernimmt.

uux [-] [-r] [-xnum] Kommandokette

Die Kommandokette besteht aus einem oder mehreren Argumenten, so dass sie wie eine Shell-Kommandokette aussieht. Bei der Angabe der "Kommandokette" ist folgendes zu beachten:

- Am Beginn der Kommandokette steht eine der auszufuehrenden Komponenten, der ein Praefix der Form "Systemname!" (Achtung! bei Verwendung der C-Shell muss dem "!" ein "\" vorangestellt sein) vorangestellt wird. Dieser Systemname gibt das Ausfuehrungssystem an, in dem die gesamte Kommandokette ausgefuehrt wird.

- Allen weiteren auszufuehrenden Komponenten sowie moeglichen Options zu den Komponenten der Kommandokette darf kein "!" vorangestellt werden (sonst Ausdehnung des Pfadnamens). Das Ausfuehrungssystem fuer diese Kommandos wird durch den System-Praefix des ersten Kommandos in der Kette bestimmt.

- Alle Komponenten der "Kommandokette", die ein "!" enthalten, werden als Dateien interpretiert und entsprechend ausgedehnt.

Falls sich eine Komponente auf dem lokalen System befindet, kann der Systemname leer sein ("!" muss aber vorhanden sein). Uux stellt dann automatisch den Namen des eigenen Systems voran. Fuer die Angabe von Pfadnamen gilt das unter Abschn. 2.1. gesagte. Stehen in der uux-Kommandozeile (rechts von uux) Shell- Sonderzeichen, wie "<",">" und "|", so muessen diese in Anfuehrungsstriche eingeschlossen oder durch einen vorangestellten Backslash ("\") markiert werden. Bei der Benutzung von Pipes ist darauf zu achten, dass es keine Pipes ueber mehrere Rechner hinweg gibt, d.h. alle Komponenten einer Pipe muessen auf dem gleichen Rechner ausgefuehrt werden.

Options:

- Der Standardeingang fuer die Kommandokette wird vom Standardeingang des uux-Kommandos genommen

-r Verhinderung des automatischen Starts des Programms uucico

-xnum Angabe der gewuenschten Debugging-Ebene (num=4 Kurzprotokoll, num=8 ausfuehrliches Protokoll). Ein Protokoll zum Ablauf der Uebertragung mittels uucico bzw. Kommandoausfuehrung uuxqt kann nur durch direkten Start dieser Programme mit der entsprechenden Debugging-Option erzielt werden. Die Option fuehrt zur Ausgabe von Informationen, die die Analyse der Kommandozeile sowie das Errichten der Eintragungen im Spool-Directory betreffen

Beispiele:

- Mit der Kommandozeile

  uux WEGA_A!cc !programm

  wird die Compilierung der Datei programm (des aktuellen Directorys) des lokalen Rechners auf dem Rechner WEGA_A angewiesen. Die Datei a.out verbleibt im Directory /usr/lib/uucp/.XQTDIR des Systems WEGA_A.

- Beispiel fuer die Benutzung einer entfernten Datei:

  uux !diff !/z/f1 WEGA_B!/z/otto/f2 ">" !/z/f3

  Am lokalen Rechner wird ein diff Kommando ausgefuehrt, das die Unterschiede zwischen der lokalen Datei /z/f1 und der sich auf WEGA_B befindlichen Datei /z/otto/f2 fixiert. Das Ergebnis wird in der Datei /z/f3 des lokalen Rechners geschrieben.

- Beispiel fuer die Benutzung eines entfernten Programms (Compiler, ...)

  uux WEGA_C!nroff !/z/otto/text1 ">" !/z/otto/text1.d

  Die lokale Datei text1 wird am Rechner WEGA_C mit der dort vorhandenen Komponente nroff uebersetzt. Das Resultat wird wieder an den lokalen Rechner zurueckgesendet und in die Datei text1.d geschrieben.

- Beispiel fuer die Benutzung eines entfernten Geraetes:

  uux !pr !/z/otto/prog.c ">" WEGA_B!/dev/lp1

  Die lokale Datei prog.c wird auf dem Drucker des Rechners WEGA_B ausgegeben.

- Beispiel fuer die Benutzung der Option "-":

  pr abc | uux - WEGA_C!lpr

  Damit wird angewiesen, dass der Standardeingang fuer das lpr Kommando auf dem entfernten Rechner WEGA_C von der Standardausgabe von "pr abc" genommen wird.

- Beispiel fuer Options in der Kommandokette

  uux WEGA_A!cc -c -O !prog.c -o prog.o

  Die lokale Datei prog.c wird am Rechner WEGA_A uebersetzt. Der Objektmodul erhaelt den Namen prog.o (er verbleibt im Directory /usr/lib/uucp/.XQTDIR des Rechners WEGA_A).

- Beispiel fuer eine Pipe innerhalb der Kommandokette

  uux "WEGA_C!deroff !/z/doc/d1|pr > !/dev/lp"

  Die Datei d1 vom lokalen Rechner wird am Rechner WEGA_C mittels deroff bearbeitet und anschliessend ebenfalls auf WEGA_C durch pr aufbereitet. Das Resultat wird am lokalen Rechner auf lp ausgedruckt.

- Senden einer Mitteilung auf einen Bildschirm eines entfernten Rechners

  uux !echo Das ist eine Mitteilung ">" WEGA_A!/dev/console

- Auflisten eines Directory-Inhaltes eines entfernten Rechners

  uux WEGA_A!ls WEGA_A!/usr ">" !/z/otto/contents

- Beispiel der Kommandoausfuehrung ueber zwei Rechner hinweg

  uux WEGA_B!uux \(WEGA_C!cc WEGA_C!/z/otto/test.c\)

  Auf dem System WEGA_C soll ein C-Compiler aufgerufen werden. Es bestehen aber nur Verbindungen zwischen dem lokalen System und WEGA_B und zwischen WEGA_B und WEGA_C. Der in Klammern eingeschlossene Teil der Kommandozeile (vor die Klammern muss ein "\") wird vom ersten uux unveraendert weitergeleitet und erst vom zweiten uux auf dem System WEGA_B ausgewertet. Die Ausgabedatei verbleibt auf dem System WEGA_C (/usr/lib/uucp/.XQTDIR) und kann mit einem uucp-Kommando zum lokalen Rechner geholt werden.

Fehlerbehandlung:

Ueber eine fehlerhafte Kommandoausfuehrung auf dem entfernten Rechner wird der Nutzer ueber eine mail-Mitteilung informiert. Wenn Auftragseintrag und Uebertragung erfolgreich, die Kommandoausfuehrung jedoch nicht erfolgreich gewesen sind, kann durch Start des Programmes uuxqt versucht werden, die Kommandoausfuehrung nochmals anzustossen. Fehler in der Phase der Auftragsaufsetzung durch uux bzw. Uebertragung durch uucico werden im LOGFILE verzeichnet. Bei Fehlern waehrend der Datenuebertragung kann uucico nochmals gestartet werden.

2.3. Kommando uucico

Das Programm uucico fuehrt Uebertragungsauftraege aus, die mit Hilfe der Kommandos uucp bzw. uux in das Spool-Directory eingetragen wurden . Uucico realisiert den Aufbau einer Verbindung zu einem entfernten Rechner sowie den Datenaustausch. Es wird im Allgemeinen durch die Programme uucp, uux oder uuxqt automatisch gestartet. Dem Bediener dient es fuer Testzwecke, um eventuell bei einer vorangegangenen Datenuebertragung aufgetretene Fehler analysieren zu koennen. Darueberhinaus kann es benutzt werden, um ein nachtraegliches Abarbeiten von in dem Spool-Directory des lokalen oder entfernten Rechners verbliebene Auftraegen durchzufuehren.

Kommandoformat

```
uucico -r1 [-xnum] [-ssys] [-ddir]
```

Options :

-r1 Start des Programmes im Master-Modus . Diese Option muss beim Aufruf des Programms uucico durch den Bediener stets angegeben werden .

-ssys Pruefen des Spool-Directorys auf Arbeit nur bezueglich des Systems sys. Bei vorhandenen Auftraegen wird die Uebertragung durchgefuehrt. Wenn kein Auftrag fuer dieses System vorliegt, wird trotzdem ein Ruf an dieses aufgesetzt, um das dortige Spool-Directory nach Auftraegen abzufragen (Polling).

-ddir Nutzung des Directorys "dir" als Spool-Directory. Dabei ist zu beachten, dass sich das Directory "dir" auf dem gleichen physischen Geraet befinden muss wie das Spool-Directory. Anderenfalls beendet uucico mit Fehlerstatus seine Arbeit .

xnum Angabe der gewuenschten Debugging-Ebene (num=4 Kurzprotokoll, num=8 ausfuehrliches Protokoll)

Beispiel :

Mit dem Kommando

```
uucico -r1 -susg
```

wird das Spool-Directory des lokalen Rechners auf Auftraege fuer das System "usg" geprueft und gegebenenfalls die Auftraege abgearbeitet. Ist keine Eintragung fuer das System usg vorhanden, wird trotzdem eine Verbindung mit diesem System hergestellt. Falls dort Auftraege fuer das lokale System existieren, uebernimmt das System usg die MASTER-Rolle (Rollentausch) und fuehrt die Auftraege aus.

Fehlerbehandlung

Die Informationen ueber die Durchfuehrung der Datenuebertragung sind im LOGFILE niedergelegt. Falls detailliertere Angaben ueber den Verlauf der Uebertragung und moegliche Fehlerursachen gewuenscht werden, muss das Debugging- Protokoll eingeschaltet werden. Dies geschieht mittels der -x Option. In diesem Falle wird auf dem Bildschirm des Bedieners das Protokoll der Uebertragung angezeigt. Mit der Angabe der -x Option auf dem MASTER-Rechner wird automatisch auch das Protokoll auf dem SLAVE eingeschaltet. Das Protokoll des SLAVE wird in die Datei AUDIT im Spool-Directory geschrieben, wo sie spaeter vom Bediener ausgewertet werden kann. Wenn nach einer fehlerhaften Uebertragung der Ruf wiederholt werden soll (C.-File ist im Spool-Directory verblieben), muessen eventuell im Spool-Directory vorhandene LCK-Dateien (verbleiben nur nach anormalem Abbruch von uucico, nicht wenn uucico nach einem Uebertragungsfehler normal beendet wird), vorher geloescht werden.

Treten in der Phase des Verbindungsaufbaus Fehler auf, die auf eine Nichtuebereinstimmung der Sequenz-Nummer der Uebertragung hinweisen, sind diese Zaehler in den Dateien SQFILE der beiden Systeme in Uebereinstimmung zu bringen.

3. Struktur und Arbeitsweise des Programmpaketes

3.1. Bestandteile

Das Programmpaket zur Unterstuetzung der Rechnerkopplung UUCP besteht aus vier primaeren und vier sekundaeren Programmen. Als primaere werden hier diejenigen Programme bezeichnet, die direkt am Datenaustausch bzw. der entfernten Kommandoausfuehrung beteiligt sind. Die uebrigen sekundaeren Programme werden benutzt, um fuer den Nutzer einen gewissen Service bereitzustellen.

Primaere Programme:

uucp — Erstellt ausgehend von einer Analyse der gegebenen Kommandos Arbeitsdateien (Daten-Files und Kommando-Files) im Spool-Directory, die die geforderte Datentransferoperation genauer charakterisieren und spaeter von uucico zur Ausfuehrung der eigentlichen Uebertragung benutzt werden.

uux — Analysiert eine Kommandozeile zur entfernten Ausfuehrung von WEGA-Kommandos und erstellt im Spool-Directory entsprechende Arbeits- und Ausfuehrungsdateien.

uusend — Analysiert eine Kommandozeile zur Dateiuebertragung ueber mehrere Rechner und erstellt im Spool-Directory entsprechende Arbeits- und Ausfuehrungsdateien.

uucico — Fuehrt die im Spool-Directory befindlichen Arbeitsdateien aus und realisiert die eigentliche Uebertragung.

uuxqt — Fuehrt die von uux erstellten Ausfuehrungsdateien zur entfernten Ausfuehrung von WEGA-Kommandos aus.

Als Kommandointerface fuer den Nutzer sind nur uucp und uux sinnvoll. Uucico und uuxqt werden von uucp bzw. uux automatisch gestartet. Fuer Testzwecke ist jedoch auch ein getrennter Start von uucico und uuxqt moeglich.

Sekundaere Programme:

uuname — Zeigt die Namen aller im lokalen System bekannten entfernten Rechner an.

uulog — Vereinigt mehrere LOGFILES in ein LOGFILE und zeigt die Ausfuehrung von UUCP-Kommandos betreffenden Statusinformationen (Eintragungen im LOGFILE) an.

uuclean Loescht alte Dateien im Spool-Directory.

uusetty Fuehrt fuer uucico Hilfsoperationen in den Phasen des Verbindungsaufbaus und -abbruchs durch (nicht fuer den Bediener vorgesehen).

Ein weiteres Programm, das zwar nicht zum Programmpaket UUCP gehoert, aber trotzdem das Programm uucico zur Datenuebertragung nutzt, ist das Programm mail (siehe auch mail(1)). Dieses Programm kann auch zur Uebertragung von Nachrichten an einen Nutzer eines entfernten Systems genutzt werden. Dazu ist in der Kommandozeile vor dem Nutzernamen, durch ein ! getrennt der Systemname zu setzten.

3.2. Grundstruktur

3.2.1. Dateien und Directories

Folgende Dateien muessen mit den entsprechenden Eigenschaften existieren (ab WEGA Version 3.0.) bzw. bei einer nachtraeglichen Implementierung eingefuegt werden:

/etc/UGETTY	0700	bin	system
/etc/UGETT1	0700	bin	system
/bin/uucp	4111	uucp	system
/bin/uux	4111	uucp	system
/bin/uuname	4111	uucp	system
/bin/uulog	0111	uucp	system
/bin/uusend	4111	uucp	system
/usr/lib/uucp/uucico	6111	uucp	system
/usr/lib/uucp/uuxqt	4111	uucp	system
/usr/lib/uucp/uuclean	0100	bin	system
/usr/lib/uucp/uusetty	4110	wega	system

Folgende Dateien muessen bei der Implementierung eingerichtet werden (siehe Abschn. 3.2.2.). Bereits vorhandene Dateien sind nur als Beispiele zu betrachten.

/usr/lib/uucp/L.sys	0400	uucp	system
/usr/lib/uucp/L-devices	0400	uucp	system
/usr/lib/uucp/L-cmd	0400	uucp	system
/usr/lib/uucp/USERFILE	0400	uucp	system
/usr/lib/uucp/SQFILE	0440	uucp	system

Folgende Directories muessen, falls sie nicht vorhanden sind, erzeugt werden:

/usr/spool/uucp
/usr/spool/uucp/uucppublic
/usr/lib/uucp/.XQTDIR

3.2.2. Permanente Dateien

Diese Dateien, die bestimmte Systemdaten enthalten, befinden sich in /usr/lib/uucp. Sie muessen bei der Imlementierung des Programmpaketes UUCP vom Nutzer errichtet werden.

L.sys

Die Datei L.sys enthaelt fuer jedes entfernte System, mit dem vom lokalen System aus eine Verbindung aufgenommen werden kann, eine Zeile.

sys time dev speed phone login

Die Felder sind durch Leerzeichen voneinander getrennt.

sys — Name des entfernten Systems

time — enthaelt Wochentage und Zeiten, zu denen das entfernte System gerufen werden kann. "Any" bedeutet jeder beliebige Tag. "WK" bedeutet jeder beliebige Arbeitstag. Ein Fehlen der Stundenangabe bedeutet jede beliebige Zeit.

dev — Name des fuer die Verbindung benutzten Geraetes. Anzugeben ist nur der letzte Teil des Namens(z.B.tty5)

speed — Uebertragungsgeschwindigkeit - moeglich sind 300, 1200, 2400, 9600 und 19200

phone — Der Inhalt dieses Feldes muss mit dem Feld dev uebereinstimmen.

login — Dieses Feld, das Login - Informationen enthaelt, ist in eine Reihe von Unterfeldern gegliedert, die folgende Form haben:

expect send [expect send] ...

Bedeutung:

expect — Der Inhalt dieses Feldes wird beim Loginprozess erwartet

send — Der Inhalt dieses Feldes wird an das entfernte System gesendet, falls expect erhalten wurde. Das Feld kann wiederum als eine Folge von Unterfeldern realisiert werden:

expect[-send-expect] ...

Hierbei wird send gesendet, wenn das vorhergehende expect nach 20 Sekunden

nicht erhalten wurde. Folgende Zeichenfolgen haben eine besondere Bedeutung:

- wenn in expect "" steht, wird nichts erwartet und sofort send gesendet

- fuer send koennen folgende Sonderzeichen stehen:

  NL - \n wird gesendet
  EOT - \004 wird gesendet
  BREAK - break wird gesendet
  CTRL/Z- \032 wird gesendet

Beispiel einer vollstaendigen Zeile:

```
WEGA_A Any tty5 9600 tty5 "" NL login:-NL-login: uucp ssword: lan
```

Das entfernte System WEGA_A kann zu jeder beliebigen Zeit ueber den Terminalkanal /dev/tty5 mit einer Uebertragungsgeschwindigkeit von 9600 Baud gerufen werden. Der login-Eintrag hat folgende Bedeutung: Ohne etwas zu erwarten("") wird ein \n gesendet und anschliessend nochmals gesendet und wieder 'login:' erwartet. Wird 'login:' beim ersten oder zweiten Mal empfangen, so wird 'uucp' gesendet. Als naechstes wird 'Password:' erwartet. Es genuegt, wenn die letzten Zeichen zum Vergleich herangezogen werden. Wird diese Zeichenkette erkannt wird das naechste Feld gesendet. Diese Reihenfolge des login-Eintrages, wie sie im Beispiel angegeben ist, wird fuer die Kopplung von WEGA-Systemen empfohlen (mit Aenderung des Passwords).

L-devices

Diese Datei enthaelt fuer jedes Geraet, ueber das eine UUCP-Kopplung vorgesehen ist, eine Zeile folgenden Formates:

type device call-unit speed

Bedeutung der Felder:

type	Typ der Verbindung ("DIR" fuer Direktverbindungen)
device	Name des Geraetes, das zur Kopplung dient(z.B.tty5)
call-unit	Fuer Direktverbindungen muss dieses Feld mit dem Feld device uebereinstimmen
speed	Uebertragungsgeschwindigkeit

Beispiel:

```
DIR tty5 tty5 9600
```

L-cmd

In dieser Datei ist eine Liste der Kommandos enthalten, die entfernte Rechner am lokalen System ausfuehren duerfen. Jedes Kommando muss auf einer neuen Zeile stehen. Existiert diese Datei nicht, so koennen alle Kommandos ausgefuehrt werden. Die Kommandos rmail, mail, uucp und uux sollten auf jeden Fall in der Liste enthalten sein, um eine Nachrichtenuebertragung und eine Dateiuebertragung ueber mehrere Rechner zu ermoeglichen. Alle weiteren Kommandos haengen vom konkreten Anwendungsfall ab. Wenn z.B. der Drucker von anderen Systemen mit genutzt werden soll, so sind die Druckkommandos zu erlauben.

SQFILE

UUCP fuehrt in der Datei SQFILE Zaehler, die die Anzahl der erfolgreichen Verbindungen zu jedem System enthaelt. Beim Verbindungsaufbau werden die Zaehler beider Systeme auf Uebereinstimmung ueberprueft. Stimmen diese nicht ueberein, wird die Verbindung abgebrochen. Fuer jedes entfernte System exstiert eine Zeile des Formates:

```
sys count time
```

Bedeutung der Felder:

sys — Name des entfernten Systems

count — Anzahl der erfolgreichen Verbindungen zu diesem System

time — Zeitpunkt der letzten Uebertragung

Diese Datei muss vom Nutzer bei der Implementierung von UUCP errichtet werden. Dabei wird auf jede Zeile jedoch nur der Name des entfernten Systems angegeben. Die beiden anderen Felder brauchen nicht eingerichtet werden. Sie werden vom Programmpaket UUCP belegt und staendig aktualisiert.

Wenn die Eintraege auf beiden Systemen fehlen bzw. die Dateien auf beiden Systemen fehlen, erfolgt der Verbindungsaufbau ohne Pruefung dieser Zaehler.

USERFILE

Diese Datei enthaelt Informationen, die die Zugriffsrechte der einzelnen dem UUCP-System bekannten Nutzer und der entfernten Systeme beinhalten:

1. zu welchen Dateien ein Nutzer des lokalen Systems Zugriff hat

2. zu welchen Dateien jeder konkrete entfernte Rechner Zugriff hat

3. welchen login-Namen jeder entfernte Rechner am lokalen System benutzen darf

4. ob der konkrete entfernte Rechner vom lokalen System zurueckgerufen werden soll, um seine Identitaet zu ueberpruefen

Die Zeilen haben folgendes Format:

login,sys [c] Pfadname [Pfadname]...

login — login-Name fuer den Nutzer oder fuer das entfernte System

sys — Systemname fuer den entfernten Rechner (ein leeres Feld fuer 'login' oder 'sys' bedeutet, dass alle anderen Systeme bzw. Nutzer, die in den daruebersteheden Zeilen nicht aufgefuehrt sind, gemeint sind)

c — wahlweise angebbare Option fuer einen Rueckruf

Pfadname — Praefix des Pfadnamens zu dem der Nutzer Zugriffserlaubnis hat

Realisierung der Kontrolle der Zugriffsrechte nach den oben genannten vier Punkten:

1. Wenn das Programmpaket UUCP ein Kommando ausfuehrt, das auf dem lokalen System gegeben wurde, d.h. UUCP arbeitet im Master-Mode, dann werden als erlaubte Pfadnamen angenommen, die in der ersten Zeile im USERFILE angegeben sind, deren login-Name mit dem login-Namen des Nutzers uebereinstimmt, der das Kommando gab. Falls eine solche Zeile nicht gefunden wurde, wird die erste Zeile benutzt, deren Feld fuer den login-Namen leer ist.

2. Wenn das Programm auf ein Kommando reagiert, das von einem entfernten Rechner gegeben wurde, d.h. UUCP arbeitet im Slave-Mode, dann werden als erlaubte Pfadnamen die angenommen, die in der ersten Zeile angegeben sind, deren Systemname mit dem Systemnamen des entfernten Rechners uebereinstimmt. Falls eine solche Zeile nicht gefunden wurde, wird die erste Zeile benutzt, deren Feld fuer den Systemnamen leer ist.

3. Wenn sich ein entfernter Rechner am lokalen System anmeldet, muss der dabei benutzte login-Name auch im USERFILE enthalten sein. Es koennen mehrere Zeilen des gleichen login-Namen enthalten sein und in einer dieser Zeilen muss der Systemname mit dem Namen des konkreten entfernten Systems uebereinstimmen oder das Feld fuer den Systemnamen muss leer sein.

4. Wenn die Zeile im USERFILE, die mit (3.) ausgesucht wurde, die c-Option enthaelt, erfolgt ein Rueckruf an den entfernten Rechner, bevor irgendwelche Operationen augefuehrt werden.

Beispiele:

Die Zeile

```
uucp,WEGA_A /z /usr
```

erlaubt dem entfernten System WEGA_A sich unter dem Namen uucp anzumelden und auf alle Dateien, die sich in /z oder /usr befinden, zuzugreifen.

Eine Datei USERFILE kann folgendermassen aufgebaut sein:

```
uucp,WEGA_A /z /usr
uucp,       /z
meier,      /z/meier
,           /z /usr /bin
```

Diese Zeilen erlauben dem System WEGA_A, sich mit dem Namen uucp anzumelden und auf die Datein der /z und der /usr zuzugreifen und weiterhin erlaubt es allen anderen Systemen sich unter dem Namen uucp anzumelden und auf die Dateien des Directory /z zuzugreifen. Der Nutzer mit dem login-Namen meier darf bei Verwendung der uucp-Kommandos nur auf die Dateien der Directory /z/meier zugreifen. Alle anderen Nutzer haben Zugriff auf alle Dateien, die sich in den Directories /z , /usr und /bin befinden.

3.2.3. Temporaere Dateien

Ausser SEQF, das sich in /usr/lib/uucp befindet, werden saemtlich hier beschriebenen temporaeren Dateien im Spool-Directory (/usr/spool/uucp) errichtet.

Temporaere Daten-Files

Waehrend des Empfangs von Daten von einem entfernten Rechner werden die empfangenen Datensaetze im Spool-Directory in temporaere Dateien zwischengespeichert. Die Namen dieser Dateien haben folgende Form:

```
TM.pid.nnn
```

Hierbei sind

TM ein fester Praefix

pid eine Prozessidentifikationsnummer

nnn eine dreistellige Nummer, die bei jedem Start von uucico bei Null beginnt und fuer jede empfangene Datei um eins erhoeht wird.

Nachdem die entfernte Datei vollstaengig empfangen wurde, wird die TM-Datei an den gewuenschten Bestimmungsort kopiert, wobei es auch den gewuenschten Namen erhaelt. Dabei wird die TM-Datei geloescht. Nur wenn uucico anormal beendet wird oder waehrend des Kopierens an den Zielort ein Abbruch erfolgt, bleibt die TM-Datei im Spool- Directory stehen. Ein regelmaessiges Loeschen aller aelteren TM-Dateien (mittels uuclean -pTM) ist deshalb ratsam.

Log-Dateien (LOGFILE, SYSLOG)

Waehrend der Arbeit von uucp, uucico, uux und uuxqt werden Informationen ueber eingereihte Nutzeranforderungen (Kommandos), ueber erfolgreich oder nicht erfolgreich durchgefuehrte Rufe an ein entferntes System, ueber die Ausfuehrung von uux- Kommandos sowie ueber den Status von Kopieroperationen in der Datei LOGFILE im Spool-Directory eingetragen. Die moeglichen Eintragungen sind im Abschnitt 4. erklaert. Unter bestimmten Umstaenden (parallele Arbeit mehrerer UUCP-Komponenten und gleichzeitiger gemeinsamer Zugriff zum LOGFILE) werden mehrere individuelle Log-Dateien gebildet, deren Namen die Form LOG.nnn haben. Diese individuellen Log-Dateien sollten mittels uulog in eine Datei LOGFILE vereint werden. Es ist ratsam, das LOGFILE in regelmaessigen Zeitabstaenden zu loeschen, da neue Eintragungen einfach angefuegt werden und die schon vorhandenen bestehen bleiben, wodurch das LOGFILE stetig waechst und auch schwer ueberschaubar wird.

In der Datei SYSLOG wird fuer jede erfolgreiche Uebertragung eine Zeile eingetragen, die die Uhrzeit der Uebertragung, Namen des Nutzers und des entfernten Systems, Angaben zur ausgefuehrten Operation (Senden, Empfangen), die Anzahl der uebertragenen Bytes und die dafuer benoetigte Zeit enthaelt.

System Status Dateien

System Status Dateien werden von uucico im Spool-Directory errichtet. Ihr Namen hat die Form

STST.sys

wobei sys der Name des entfernten Systems ist, mit dem eine

Kommunikation im Gange ist. Diese Dateien enthalten Informationen ueber bestimmte Fehlerzustaende, wie z.B. Fehler beim Login oder generell beim Verbindungsaufbau usw. Waehrend des Datenaustausches zwischen zwei Rechnern enthaelt die Datei den Status "TALKING". Bei fehlerfreiem Verlauf der Konversation wird die Datei bei Abbruch der Verbindung wieder geloescht. Bei anormaler Beendigung von uucico bleibt die Datei im Spool- Directory erhalten. Wenn der Grund des vorzeitigen Abbruchs der Verbindung eine Nichtuebereinstimmung der Sequenzzaehler (in der Datei SQFILE, siehe Abschn. 3.2.2.) war, so muss die STST-Datei generell vor Start einer erneuten Uebertragung zu diesem Rechner geloescht werden, sonst erfolgt kein Ruf. Bei einer anormalen Beendigung des Programms uucico (z.B. durch ein kill oder einen Systemabsturz) kann es sein, dass die STST-Datei mit dem Status "TALKING" bestehen bleibt. In einem solchen Fall muss die Datei ebenfalls vor Beginn einer neuen Uebertragung geloescht werden.

Lock - Dateien

Lock-Dateien werden fuer jedes aktive Geraet und fuer jedes entfernte System, mit dem eine Konversation laeuft, gebildet. Die Namen der Lock-Dateien haben folgende Form:

LCK.name

wobei name der Name des Geraetes oder des entfernten Systems ist. Lock-Dateien werden gebildet, um einen parallelen Zugriff (z.B. seitens mehrerer zu der gleichen Zeit aktiver uucico-Programme) auf ein und dasselbe Geraet bzw. entfernte System zu verhindern. Um zu sichern, dass der Zugriff mehrerer Komponenten zum gemeinsam genutzten LOGFILE streng sequentiell verlaeuft, wird von der jeweiligen Komponente vor dem Zugriff zum LOGFILE ebenfalls eine entsprechende Lock-Datei errichtet (LCK.LOG) und sofort nach dem Zugriff wieder geloescht. Gleiches gilt fuer das SQFILE (LCK.SQ). Um das gleichzeitige Uminitialisieren von Terminalkanaelen zu verhindern, wird ebenfalls eine Lock-Datei (INIT.LCK) angelegt. Falls das uucico-Programm anormal beendet wird, koennen Lock-Dateien im Spool-Directory verbleiben. Nach einer Zeit von 24 Stunden werden diese ignoriert (neu genutzt). Soll jedoch ein neuer Ruf erfolgen (zu dem gleichen System oder ueber das gleiche Geraet), muessen diese Lock-Dateien vorher geloescht werden, sonst findet kein Ruf statt.

Waehrend des Verbindungsaufbaus wird beim Deaktivieren der Terminalschnittstelle kurzzeitig eine Datei GET.name gebildet. Wenn nach einer anormalen Verbindung eine solche Datei im Spool-Directory verbleiben sollte, so ist diese zu loeschen und ausserdem ist der Eintrag fuer die entsprechende Terminalschnittstelle in der Datei /etc/inittab zu ueberpruefen und gegebenenfalls zu aendern (siehe Abschn. 5.1.).

C.,D. und X.-Dateien

Die Programme uucp und uux analysieren das gegebene Kommando und bereiten die Uebertragung vor. Als Resultat dieser Vorbereitung errichtet uucp bzw. uux im Spool-Directory C.-,D.- und X.-Dateien, die die auszufuehrende Operation naeher beschreiben (C.*), die zu uebertragenden Daten enthalten (D.*) und die auszufuehrende Kommandos beschreiben (X.*).

AUDIT

Diese Datei wird im Spool-Directory des SLAVE-Systems errichtet, wenn der MASTER fuer die Uebertragung das Debugging-Protokoll eigeschaltet hatte ("-x" Option bei uucico). Beim MASTER wird dieses Protokoll standardgemaess auf dem Bildschirm des Bedienterminals ausgegeben. Das Protokoll, das die Aktivitaeten des SLAVE- Systems dokumentiert (wird beim Einschalten des Protokolls beim MASTER automatisch mit eingeschaltet), wird in der Datei AUDIT geschrieben und bleibt nach Beendigung der Uebertragung erhalten.

SEQF

Diese Datei, die sich im Directory /usr/lib/uucp befindet, beinhaltet nur eine Zeile. Hier wird ein Zaehler gefuehrt, der beim Errichten jeder temporaeren C.-, D.- oder X.- Datei im Spool-Directory um 1 erhoeht wird. Dieser Zaehler wird benutzt, um die Einmaligkeit jeder dieser Dateinamen zu garantieren. Das Vorhandensein dieser Datei ist nicht Bedingung. Bei nicht vorhandener Datei SEQF wird es neu durch uucp, uucico und uuxqt errichtet.

3.3. Arbeitsweise

3.3.1. Allgemeines

Jedes UUCP-System hat ein Spool-Directory, in welchem Anweisungen fuer auszufuehrende Kopplungen mit anderen Rechnern abgespeichert werden (Kommando-Dateien, zu uebertragende Dateien, auszufuehrende WEGA-Kommandos). Die Programme uucp und uux, die das Hauptinterface fuer den Bediener bilden, analysieren das gegebene Kommando und schreiben alle fuer die Datenuebertragung bzw. entfernte Kommandoausfuehrung benoetigten Informationen in das Spool-Directory.

Das Programm uucico realisiert die eigentliche Uebertragung, die im Hintergrund stattfindet. Auf der Grundlage der aus dem Spool-Directory entnommenen Informationen waehlt uucico die Uebertragungsleitung aus und stellt die Verbindung zum gewuenschten Rechner her. Dabei meldet sich

das lokale uucico am entfernten System an (login), woraufhin dort ebenfalls uucico gestartet wird. Nachdem auf diese Weise der Kontakt zwischen beiden uucico-Programmen hergestellt wurde, stimmen diese sich ueber das zu benutzende Uebertragungsprotokoll ab und fuehren die Datenuebertragung aus. Zuerst uebertraegt das Programm, welches die Uebertragung initialisierte, alle angewiesenen Daten (entsprechend den Auftraegen im eigenen Spool-Directory). Dann fragt es das andere Programm, ob bei diesem System ebenfalls Uebertragungsauftraege vorliegen. Ist das der Fall, werden diese ausgefuehrt. Wenn auf keiner der beiden Seiten mehr Auftraege vorliegen, beenden beide uucico-Programme ihre Arbeit.

Wird eine entfernte Kommandoausfuehrung angewiesen (uux), startet das uucico- Programm des Zielrechners nach erfolgreicher Uebertragung das Programm uuxqt, welches das gewuenschte WEGA-Kommando ausfuehrt. Nach Beendigung der entfernten Kommandoausfuehrung erfolgt eine mail-Mitteilung an den Nutzer, der den Auftrag ausgeloest hatte.

3.3.2. Beginn einer uucp-Uebertragung

Nach dem Aufruf eines uucp-Kommandos werden der Quelldateiname, der Zieldateiname und der Systemname ueberprueft, um das Kommando nach einem der folgenden Typen zu klassifizieren:

typ 0 Kopieren der Quelle in ein Ziel der lokalen Anlage

typ 1 Erhalten von Dateien von anderen Systemen

typ 2 Senden von Dateien zu entfernten Systemen

typ 3 Senden von Dateien von einem entfernten System zu einem anderen

typ 4 Erhalten von Dateien eines entfernten Systems, wenn der Quelldateiname Shell-Sonderzeichen enthaelt, wie ? * [].

Nachdem die Aufgaben im Spool-Directory hinterlegt wurden, wird das uucico Programm gestartet (ausser die Option -r wurde gegeben). Dieses Programm versucht eine Verbindung aufzubauen, um die hinterlegten Aufgaben auszufuehren.

Im Einzelnen wird bei den oben genannten Typen wie folgt verfahren:

Typ 0 Ein cp-Kommando wird benutzt, um die Aufgaben auszufuehren. Die Options -d und -m werden in diesem Fall nicht unterstuetzt.

Typ 1 Eine einzeilige Kommandodatei (der Dateiname beginnt mit C.*) vom Typ "S" (Sende-Kommando) wird fuer jede Datei erstellt und in das Spool-Directory (/usr/spool/uucp) geschrieben. Die Datei besteht aus mehreren Feldern, wobei jedes Feld durch ein Leerzeichen getrennt ist. (Alle Kommandodateien und Befehlsdateien benutzen das Leerzeichen als Trennzeichen.)

Typ 2 Fuer jede Quelldatei wird eine Kommandodatei vom Typ "R" (Empfangskommando) erzeugt und die Quelldatei wird in eine Textdatei der Spool-Directory kopiert. (Eine Option "-c" verhindert die Kopie. In diesem Fall wird die Datei aus der angegebenen Quelle uebertragen.)

typ 3
und
typ 4 uucp erzeugt ein uucp-Kommando (Kommandodatei vom Typ "X") und sendet es zum entfernten System. Das entfernte uucico-Programm fuehrt dann das uucp-Kommando aus.

Jede Kommandodatei besteht aus einer oder mehreren Zeilen, deren Inhalt aus folgenden Feldern besteht:

T File1 File2 User Options [D-file] [Mode]

Hierbei sind:

T Typ der auszufuehrenden Operation (S-Senden, R-Empfangen von Dateien, X-Ausfuehren eines entfernten uucp)

File1 voller Pfadname der Quelldatei oder ein ~loginname/pfadname, wobei ~loginname am entfernten System ausgedehnt wird.

File2 voller Pfadname der Zieldatei oder ein ~loginname/pfadname

User Name des Nutzers, der die Uebertragung angefordert hat

Options die Uebertragung betreffende Optionen

D-file Name der Datei, die die zu uebertragenden Daten enthaelt (nur bei Typ "S")

Mode Modus des D-files

3.3.3. Beginn einer uux-Uebertragung

Das uux-Kommando wird benutzt, um die Ausfuehrung eines WEGA-Kommandos zu verlagern, wobei das ausfuehrende System ein entferntes System ist und einige Dateien nicht vom lokalen System sein muessen.

Uux erzeugt eine Befehlsdatei (der Dateiname beginnt mit X.*), in der die Namen der fuer die Ausfuhrung benoetigten Dateien (einschliesslich der Standardeingabe), der Login-Name des Nutzers, das Ziel der Standardausgabe und das auszufuehrende Kommando aufgefuehrt sind. Diese befindet sich entweder im Spool-Directory zur lokalen Ausfuehrung oder wird zu einem entfernten System mittels eines erzeugten "Sende"-Kommandos (siehe Typ 2 uucp Beschreibung) gesandt.

Fuer benoetigte Dateien, die nicht auf dem ausfuehrendem System sind, erzeugt uux "Empfangs"-Kommandos (siehe Typ 1 uucp Beschreibung). Diese Befehlsdateien werden auf das ausfuehrende System uebertragen und durch das uucico-Programm abgearbeitet. Dies gelingt nur dann, wenn das lokale System die Erlaubnis hat, Dateien in das entfernte Spool-Directory zu senden. Die Kontrolle erfolgt durch die Datei USERFILE (/usr/lib/uucp/USERFILE) des entfernten Systems.

Die Befehlsdatei wird durch das uuxqt-Programm auf dem ausfuehrenden System abgearbeitet. Diese Datei besteht aus mehreren Zeilen, jede davon enthaelt ein Identifikationszeichen und eines oder mehrere Argumente. Die Reihenfolge der Zeilen ist nicht relevant. Die Zeilen sehen wie folgt aus:

- U Nutzer System

  Nutzer — Nutzer, der die Kommandoausfuehrung angewiesen hat

  System — Systemname des Nutzers

- F File-Name Ursprungsname

  File-Name — generierter Name der Datei, unter dem es beim Ausfuehrungssystem gespeichert wird

  Ursprungsname — Letzter Teil des Pfadnamens der Ursprungsdatei. Es koennen keine oder mehrere Zeilen mit dem Praefix F enthalten sein (fuer jede zur Kommandoausfuehrung benoetigte Datei eine Zeile).

- I File-Name

  File-Name — Standard-Eingabe fuer das auszufuehrende Kommando. Die Standard-Eingabe wird entweder durch "<" in der uux-Kommandozeile spezifiziert oder ueber eine Pipe von der Standard-Eingabe des uux- Kommandos uebernommen. (die uux-Option "-" wurde angegeben). Falls eine Standard-Eingabe nicht angegeben wurde (mit den oben beschriebenen 2 Moeglichkeiten), wird "/dev/null" benutzt.

- O File-Name Systemname

  File-Name — Standard-Ausgabe fuer das auszufuehrende Kommando

  Systemname — System auf dem die Standard-Ausgabe erfolgen soll. Die Standard-Ausgabe wird durch ">" in der uux-Kommandozeile spezifiziert. Wurde keine Standard-Ausgabe angegeben, wird "/dev/null" benutzt.

- C Kommando [Argumente] ...

  Kommando — auszufuehrendes Kommando

  Argumente — in der uux-Kommandozeile spezifizierte Argumente. Die Standard-Eingabe und Standard-Ausgabe erscheinen in dieser Zeile nicht mit.

3.3.4. Ablauf eines uucico-Aufrufs

Das uucico-Programm erfuellt die folgenden Hauptfunktionen:

- Durchsuchen der Spool-Directory nach Kommandodateien (C.*)
- Einen Ruf an das entfernte System absetzen
- Umsetzen des benutzten Protokolls
- Ausfuehren der Anforderungen beider Systeme
- Protokollanforderungen und Arbeitsabschluss

Uucico kan auf verschiedene Weise gestartet werden:

- von einem System Daemon

- von einem der Programme uucp, uux, uuxqt oder einem uucico-Anruf eines entfernten Systems

- direkt von einem Benutzer (gewoehnlich zu Testzwecken)

- von einem entfernten System (das Programm uucico muss dabei als Standard-Shell im uucp-Logineintrag angegeben werden)

Wenn uucico auf eine der ersten drei Arten aufgerufen wird, nimmt das Programm den Master-Modus an. In diesem Modus beginnt es die Verbindung zu einem entfernten System aufzubauen. Falls das Programm von einem entfernten System gestartet wird, arbeitet es im Slave-Modus.

Der Master-Mode operiert auf eine der zwei Arten:

- Das Programm ueberprueft die Spool-Directory nach Systemen, die gerufen werden sollen.

- Wurde beim Start von uucico (im Master-Mode) die Option "-s" angegeben, fuehrt uucico nur Auftraege fuer das spezifizierte System aus. Dabei erfolgt auf jeden Fall ein Ruf an das System, auch wenn keine Auftraege vorhanden sind. Auf diese Art kann ein Polling eines entfernten Systems realisiert werden.

Ueberpruefen der Spool-Directory

Die Namen der Arbeitsdateien in der Spool-Directory (/usr/spool/uucp/...) haben folgendes Format:

Typ.sysnxxxx

Typ C - fuer Kommandodatei
D - fuer Textdatei
X - fuer Befehlsdatei

sys Name des entfernten Systems, mit dem kommuniziert werden soll

n ein mittels der uucp-Option "-g" eingefuegter Buchstabe (Standard "n"), der zur besonderen Kennzeichnung benutzt werden kann.

xxxx eine vierstellige fortlaufende Zahl (entnommen der Datei /usr/lib/uucp/SEQF)

Die Ueberpruefung erfolgt in der Weise, dass die Spool-Directory nach Dateien mit dem Praefix C durchsucht wird. Wurde eine Datei gefunden, startet uucico einen Ruf an das darin spezifizierte System. Ist keine Datei mehr vorhanden, dann beendet uucico die Arbeit.

Ruf des Entfernten Systems

Anhand des Systemnamens wird in der Datei L.sys die diesem System zugeordnete Zeile gesucht. Der Inhalt des "time"-Feldes dieser Zeile wird nun mit der aktuellen Zeit verglichen um festzustellen, ob zu dieser Zeit ein Ruf erlaubt ist. Mit Hilfe der ebenfalls aus dieser Datei entnommenen Daten zu Geraetenamen und Uebertragungsgeschwindigkeit sucht uucico in der Datei L-devices ein fuer diesen Ruf geeignetes Geraet. Wenn ein Geraet erfolgreich "eroeffnet" wurde (open), schreibt Uucico wiederrum eine entsprechende Lock-Datei in das Spool-Directory, um zu verhindern, dass ein parallel arbeitendes uucico zum gleichen Geraet zugreift.

Nachdem die Auswahl und das Eroeffnen des Geraetes erfolgreich abgeschlossen wurden, nutzt uucico die in L.sys enthaltenen Login-Informationen, um sich am entfernten System anzumelden (siehe Abschn. 3.2.2.). Verlaeuft das Anmelden erfolgreich, wird am entfernten System ebenfalls uucico gestartet (als Standard-Shell im fuer uucp angelegten Eintrag in der Password-Datei des entfernten Systems) und zwar im SLAVE-Modus.

Die Konversation zwischen den beiden uucico-Programmen (siehe auch Anlage) beginnt mit einem Handshake, der von dem gerufenen (dem SLAVE) System gestartet wird. Das SLAVE-System sendet dem MASTER (dem in MASTER-Modus gestarten uucico) eine Nachricht, um ihm mitzuteilen, dass es bereit ist, die Anfangsnachricht zu empfangen. Daraufhin sendet der MASTER seinen Systemnamen und die Sequenznummer der Verbindung. Diese Daten werden beim SLAVE auf Zulaessigkeit geprueft. Wenn diese Pruefung positiv ausfaellt, beginnt die Auswahl des Uebertragungsprotokolls. Das SLAVE-System kann aber auch mit einer Nachricht "call-back ist gefordert" antworten (das heisst, im USERFILE des SLAVE war fuer dieses System die call-back Option angegeben worden). Daraufhin wird die laufende Verbindung beendet und der Verbindungsaufbau beginnt von neuem, nun jedoch von Seiten des ehemaligen SLAVE-Systems, das jetzt die MASTER-Rolle uebernimmt. Damit erfolgt eine zusaetzliche Verifikation des Partnersystems.

Auswahl des Uebertragungsprotokolls

Das entfernte System (SLAVE) sendet eine Liste der dort vorhandenen Uebertragungsprotokolle (Im Standardfall gibt es nur ein Protokoll: den Paketdriver). Diese Nachricht hat folgende Form:

Plist wobei list eine Zeichenfolge ist, in der fuer jedes verfuegbare Protokollein Zeichen steht

Der andere Partner vergleicht diese Liste mit den im eigenen System vorhandenen Protokollen und waehlt ein bei

beiden vorhandenes aus. Wird kein gemeinsames Protokoll gefunden, erfolgt eine entsprechende Mitteilung und die Verbindung wird beendet. Es wird eine Nachricht folgender Form gesendet:

Ucode wobei code entweder das Zeichen fuer das ausgewaehlte Protokoll ist oder ein N, wenn kein gemeinsames Protokoll gefunden wurde.

Alle Nachrichten, die waehrend der Phase des Verbindungsaufbaus ausgetauscht werden, sind zur Sicherung der Synchronisation in ein beginnendes DLE (Code 020) und in ein abschliessendes NULL- Zeichen (Code 0) eingeschlossen. Nach erfolgter Auswahl des Protokolles wird dieses gestartet und der Austausch aller folgenden Steuernachrichten und Daten wird unter Benutzung dieses Protokolls durchgefuehrt).

Datenaustausch

Die Rollen zu Beginn der Datenuebertragung (MASTER oder SLAVE) entsprechen dem Modus mit dem jedes Programm gestartet wurde. Der Datenaustausch wird durch 2 Protokollebenen realisiert. Das ist erstens die Ebene, in der die Mitteilungen zur Steuerung des gesamten Prozesses des Austausches von Daten realisiert werden. Diese Ebene ist fest im Programm uucico verankert und wird durch die Protokollauswahl beim Verbindungsaufbau nicht beeinflusst. Die zweite darunterliegende Ebene ist die des Uebertragungsprotokolls, das die fehlerfreie Uebertragung der Steuermitteilungen der Ebene 1 und der Daten selbst absichert. Im Standardfall (nur ein Uebertragungsprotokoll zur Auswahl) ist das der Paketdriver, der alle zu uebertragenden Nachrichten in Pakete verpackt.

Ebene 1:

Zur Steuerung des Datenaustausches werden 5 Nachrichtentypen benutzt:

S der MASTER sendet eine Datei

R der MASTER empfaengt eine Datei

C Kopieren der Datei an den Zielort wurde beendet

X der SLAVE fuehrt ein uucp - Kommando aus

H Beendigung der Uebertragung.

Der MASTER sendet R, S oder X - Nachrichten (und die dazugehoerigen Daten), bis alle Auftraege aus seinem Spool-Directory abgearbeitet sind. Danach wird eine H - Nachricht gesendet. Der SLAVE antwortet entsprechend mit

SY, SN, RY, wobei Y und N Yes oder No bedeuten. Die Antworten des SLAVE auf S und R (Y oder N) haengen von den Zugriffsrechten der gewuenschten Dateien (Directories) ab. Die Zugriffsrechte werden ausgehend vom USERFILE und vom Dateimodus (Lese/Schreiberlaubnis) der gewuenschten Dateien (Directories) geprueft.

Beim Empfang von Dateien werden die eingehenden Daten in eine temporaere Datei (TM...) im Spool-Directory geschrieben. Nach dem vollstaendigen Empfang einer Datei wird sie an den gewuenschten Zielort kopiert. Danach sendet das empfangende System eine C - Nachricht an das Sendende. Wurde die temporaere Datei erfolgreich an den Zielort kopiert, wird CY gesendet. Andernfalls (Fehler beim Kopieren, fehlende Zugriffsrechte, usw.) lautet die Nachricht CN, was bedeutet, dass die temporaere Datei in das von uucico errichtete Directory /usr/spool/uucp/uucppublic/user geschrieben wurde, wobei user der Name des Nutzers ist, der die Uebertragung angewiesen hatte. In diesem Fall wird an dem System, das die Uebertragung angefordert hatte, eine entsprechende mail - Nachricht an den Nutzer geschickt, unabhaengig davon, ob die uucp - Option "-m" angegeben wurde oder nicht. (Eine mail - Nachricht wird auch bei allen anderen Ablehnungsfaellen (SN, RN, XN) generiert.)

Wenn das SLAVE-System eine H-Nachricht erhaelt, durchsucht es das Spool- Directory nach vorhandenen Auftraegen. Liegen Uebertragungsauftraege fuer das MASTER-System vor, antwortet der SLAVE mit HN . Daraufhin tauschen die beiden uucico-Programme die Rollen (MASTER <---> SLAVE) und die vorhandenen Auftraege werden abgearbeitet. Findet der SLAVE in seinem Spool-Directory keine Arbeitseintraege antwortet er mit HY, was zum Abbruch der Verbindung fuehrt.

Ebene 2 - Paketdriver:

Der Paketdriver ist ein Teil des Programms uucico, der aus den zu uebertragenden Daten und Steuernachrichten (Ebene1) Pakete formiert und diese mit einer Prueffolge versieht. Die Datenuebertragung mit Hilfe des Paketdrivers ist vollkommen transparent, d.h. es gibt keine Einschraenkungen hinsichtlich der Codes der zu uebertragenden Daten.

Der Paketdriver zerlegt die zu uebertragenden Daten in Bloecke zu je 64 Byte (die Steuermitteilungen werden entsprechend aufgefuellt). Jedem zu uebertragenden Paket wird ein Header von 6 Byte Laenge vorangestellt. Vom Header wird eine getrennte Pruefsumme gebildet.

Die Bytes des Header haben folgende Bedeutung:

1 Synchronisationszeichen (DLE)

2 kodierte Paketlaenge

3,4 Pruefsumme des Datenblockes

5 Kontrollbyte (enthaelt Informationen ueber das Paket)

6 Pruefsumme des Headers (Exclusive - ODER der Bytes 2-5)

Der Paketdriver wird durch eine spezielle Routine aktiviert. Die Paketdriver der beiden kommunizierenden Systeme stellen dann eine logische Verbindung her, wobei ein Austausch von Headern vollzogen wird. Verlaeuft das erfolgreich, sind die Paketdriver beider Systeme bereit, alle ihnen uebergebenen Informationen zu uebertragen. Alle Pakete werden vom Sender fortlaufend nummeriert. Der empfangende Paketdriver sendet dem sendenden fuer jedes empfangene Paket eine Quittung. Die Zuordnung der Quittungen zu den gesendeten Paketen erfolgt ueber die Paketnummer. Der Sender kann bis zu 3 Pakete im voraus senden, d.h. ohne auf die entsprechenden Quittungen zu warten. Erst wenn auf 3 gesendete Pakete keine Quittung vorliegt, unterbricht die sendende Seite die Uebertragung und wartet auf Quittungen. Bei Fehlersituationen (falsche Pruefsumme, falsche Headerpruefsumme, keine Antwort in einer festgelegten Zeit (Timeout - 20s), ...) erfolgt eine Wiederholung der Uebertragung. Die Wiederholung beginnt beim ersten nicht quittierten Paket. Nach 5 aufeinanderfolgenden Fehlversuchen wird die Verbindung geloest.

Verbindungsabbruch

Der Verbindungsabbruch erfolgt getrennt auf beiden Ebenen des Uebertragungstokolls. Zuerst wird die Verbindung der beiden Paketdriver beendet. Dabei tauschen die beiden Seiten spezielle Endenachrichten aus. Daraufhin werden alle belegten Pruefsegmente freigegeben. Der anschliessende Verbindungsabbruch auf der 1., hoeheren Ebene erfolgt ohne Paketdriver. (Die Nachrichten werden wieder in DLE und NULL-Zeichen eingeschlossen und ohne Paketdriver uebertragen.) Beide uucico-Programme senden sich Endenachrichten. Sobald beide Seiten die Endenachricht des Partners erhalten haben, gilt die Verbindung als geloest.

Nach dem Loesen der Verbindung bereinigt das sich zu diesem Zeitpunkt im SLAVE- Modus befindliche Programm das Spool-Directory und beendet seine Arbeit. Das MASTER-Programm prueft erst sein Spool-Directory, ob dort Auftraege fuer ein anderes System vorliegen (falls uucico nicht mit der "-s" -Option gestartet wurde). Ist das der Fall, beginnt der Verbindungsaufbau mit diesem System, sonst beendet das MASTER-uucico ebenfalls seine Arbeit.

4. Diagnoseunterstuetzung

4.1. Moeglichkeiten der Fehlererkennung

Die wichtigsten Informationen ueber den Ablauf der Datenuebertragung sind im LOGFILE niedergelegt. Anhand dieser Eintragungen kann ermittelt werden, in welcher Phase die Fehler aufgetreten sind und es koennen entsprechende Schritte unternommen werden. Die wichtigsten Mitteilungen des LOGFILE sind im Abschnitt 4.2. dieser Schrift erklaert. Bei eindeutig erkannten Fehlern kann die Wiederholung der Datenubertragung und damit Abarbeitung des Auftrages mittels des Kommmandos uucico angewiesen werden. (Wenn waehrend der Uebertragung ein nichtbehebbarer Fehler auftritt, wird die Verbindung abgebrochen. In diesem Fall verbleibt die die Uebertragung anweisende Kommando-Datei (C.-File) im Spool-Directory. Eine Wiederholung des Rufes kann mittels uucico erreicht werden, womit alle im Spool-Directory befindlichen C.-Files ausgefuehrt werden). Wenn bei sich haeufig wiederholenden Fehlern die Informationen des LOGFILE's nicht ausreichen, um die Fehlerursache festzustellen, kann das Testprotokoll (Debugging-Protokoll) eingeschaltet werden. Zum Einschalten des Testprotokolls, das alle Aktivitaeten waehrend des Verbindungsaufbaus, sowie waehrend der Datenuebertragung genauestens dokumentiert, muss uucico vom Bediener gestartet werden (nicht automatisch durch uucp oder uux), d.h. bei uucp bzw. uux wurde die "-r" Option angegeben. Zum Start des Testprotokolls muss uucico mit der "-x" Option gestartet werden. Bei "-x4" (Debugging-Ebene 4) werden die Hauptetappen protokolliert. Vom Paketdriver sind dabei keine Informationen enthalten. Bei Angabe von "-x8" wird das volle Testprotokoll ausgegeben, einschliesslich der umfangreichen Informationen des Uebertragungsprotokolls. Das Testprotokoll wird standardgemaess auf den Bildschirm des Bedieners ausgegeben, kann aber auch zwecks spaeterer Analyse in eine Datei geschrieben werden, z.B. mit dem Kommando

```
uucico -r1 -x8 2>&1 | tee /usr/spool/uucp/proto
```

Bei Start des Testprotokolls beim MASTER-System wird das Protokoll automatisch auch beim SLAVE mit eingeschaltet. Das Protokoll des SLAVE-Rechners wird in der Datei AUDIT im Spool- Directory geschrieben. Zur Auswertung des Testprotokolls ist eine detailierte Kenntnis des Uebertragungsablaufes und des Uebertragungsprotokolls erforderlich.

Eine weitere Moeglichkeit, sich ueber den erfolgreichen Ablauf der Uebertragung zu informieren, steht mit dem mail-Kommando zur Verfuegung, da alle fehlerhaften Uebertragungen eine mail-Eintragung erzeugen. Die mit der -m Option versehenen uucp-Kommandos erzeugen auch bei ordnungsgemaesser Uebertragung eine mail-Eintragung.

4.2. Mitteilungen im LOGFILE

Die Mitteilungen im LOGFILE haben folgendes Format:

Nutzer System (Zeit) Text

Nachfolgend werden die wichtigsten Texte der Mitteilungen erklaert. Fuer die meisten angegebenen Bedieneraktionen ist es empfehlenswert, die entsprechenden Abschnitte (Beschreibung der entsprechenden Dateien, ...) mit zu Hilfe zu nehmen. Fuer die Bestimmungen der Bedienerhandlungen ist es notwendig, alle die konkrete Uebertragung betreffenden Eintragungen im LOGFILE im Zusammenhang zu betrachten, um daraus den Fehlerhergang und die moegliche Ursache genauer abzuleiten. Der Bediener muss von Fall zu Fall selbst entscheiden, ob nach Beseitigung der moeglichen Fehlerursache eine Wiederholung des Rufes sinnvoll ist und ab welcher Etappe die Wiederholung beginnen soll (Eintragung des Auftrages uucp bzw. uux , Uebertragung uucicco, Kommandoausfuehrung uuxqt oder uucico).

- ...QUE'D

  UuCP hat den Auftrag in das Spool-Directory eingetragen.

- DONE WORK HERE

  Sowohl Quell- als auch Zielort lagen auf dem lokalen System. Uucp hat die Kopieroperation am eigenen Rechner ausgefuehrt (ohne uucico).

- SUCCEEDED (CALL TO SYS)

  Der Ruf an das System "SYS" war erfolgreich. (login, Start des entfernten uucico).

- REQUIRED CALLBACK

  erscheint am Slave-System. Fuer das MASTER-System wurde die Callback-Option (L.sys) angegeben. Die laufende Verbindung wird beendet. Das ehemalige Slave-System baut die Verbindung, nun als MASTER, neu auf.

- PREVIOUS BADSEQ

  Eine vorhergehende Verbindung war aus Gruenden der Nichtuebereinstimmung der Sequenznummern der Verbindung (im SQFILE) abgebrochen worden. Deshalb wird

auch die laufende Verbindung abgebrochen.

Bediener:

Vor einem weiteren Versuch muss erst die Datei STST... im Spool-Directory geloescht werden (natuerlich auch die beiden Sequenzzaehler in Uebereinstimmung gebracht werden).

- HANDSHAKE FAILED (BADSEQ)

Die Sequenznummern der Verbindung (im SQFILE) der beiden Rechner stimmen nicht ueberein. Abbruch der Verbindung.

Bediener:

Kontrolle und evtl. Korrektur der jeweiligen Sequenzzaehler (SQFILE's).

- CAN NOT CALL (SYSTEM STATUS)

Eine System-Status-Datei (STST...) ist im Spool-Directory verblieben und der dort eingetragene Status bedeutet, dass zu diesem System noch eine Verbindung aktiv ist.

Bediener:

Falls es keine aktive Verbindung mehr gibt, auf deren Beendigung gewartet werden muesste, muss die STST... -Datei im Spool-Directory geloescht werden.

- NO WORK

Uucico wurde ohne die "-s"-Option gestartet und im Spool- Directory sind keine Auftraege verzeichnet.

- LOCKED (call to name)

Es existiert eine Lock-Datei (LCK...) fuer das System "name", die einen gleichzeitigen Ruf zu diesem System verbietet.

Bediener:

Wenn eine Konversation mit diesem System aktiv ist - Beendigung abwarten. Wenn keine Konversation auf diesem System aktiv ist, - Loeschen der entsprechenden LCK-Datei.

- FAILED (Call to name)

  Ein Ruf an das System "name" wurde mit Fehler beendet.

  Bediener:

  Analyse der Fehlerursache anhand der anderen Eintragungen.

- SUCCEEDED (call to name)

  Der Verbindungsaufbau zum System "name" ist erfolgreich verlaufen.

- HANDSHAKE FAILED (Text)

  Das Anfangs-Handshake zwischen den beiden Rechnern wurde mit Fehlerstatus beendet. "Text" gibt die Fehlerursache an. Abbruch der Verbindung.

- FAILED (startup)

  Der Start des Uebertragungsprotokolls (Paketdriver) wurde mit Fehlerstatus beendet. Abbruch der Verbindung.

- OK (startup)

  Das Uebertragungsprotokoll wurde ordnungsgemaess gestartet. Die Verbindung zwischen den beiden Systemen ist aufgebaut und bereit zur Uebertragung der Daten.

- OK (conversation complete)

  Die Konversation mit dem entfernten Rechner (Verbindungsaufbau, Datenaustausch, Verbindungsabbruch) wurde fehlerfrei beendet.

- FAILED (conversation complete)

  Auf irgend einer Etappe der Verbindung mit dem entfernten Rechner trat ein Fehler auf. Die Verbindung wurde beendet.

  Bediener:

  Feststellen der Fehlerursache an Hand der anderen Eintragungen im LOGFILE.

- REQUEST (.Text...)

  Angefordert wurde die Ausfuehrung der Anweisung "Text". Diese Eintragung erfolgt nach ordnungsgemaess aufgebauter Verbindungen, vor der Ausfuehrung. "Text" kann auch eine Begruendung der Ablehnung der Ausfuehrung beinhalten.

- ACCESS (DENIED)

  Die Zugriffsrechte zu einer am lokalen System befindlichen Datei sind nicht vorhanden. (Datei nicht vorhanden, Directory nicht vorhanden und fehlende "make"-Directory-Option, ...)

  Bediener:

  Pruefen der Zugriffsrechte zu den entsprechenden Dateien.

- FAILED (CAN'T READ DATA)

  Gefordert war eine Datei an ein entferntes System zu senden. Die zu sendende Datei liess sich am lokalen Rechner nicht oeffnen.

  Bediener:

  Kontrolle der entsprechenden Datei.

- FAILED (CAN'T CREATE TM)

  Angewiesen war, Daten von einem entfernten Rechner zu empfangen. Die temporaere Datei TM... , in welcher die eingehenden Daten zwischengespeichert werden, laesst sich nicht errichten.

  Bediener:

  Kontrolle des Spool-Directorys (Vorhandensein, Zugriffsrechte...)

- Name (XUUCP DENIED)

  Bei einer Datenuebertragung (angefordert von einem entfernten System, das heisst das System, an dem diese Eintragung erscheint, arbeitet im Slave-Modus) machte sich der Start eines entfernten uucp erforderlich. Das ist zum Beispiel der Fall, wenn der Pfadname der entfernten Datei Shell- Sonderzeichen enthaelt. Dabei wurde festgestellt, dass die Zugriffsrechte fuer eine mit dem Kommando

angeforderte Datei "Name" nicht vorhanden sind.

Bediener:

Pruefung der Zugriffsrechte.

- PERMISSION (DENIED)

1. Der Slave-Rechner soll eine Datei empfangen und die Zugriffsrechte fuer den vom MASTER gewuenschten Zielort sind nicht in Ordnung.

   Bediener:

   Pruefen des Zielortes

2. Der SLAVE-Rechner soll eine Datei senden und die Zugriffsrechte zu der zu sendenden Datei sind nicht vorhanden.

- DENIED (CAN'T OPEN)

1. Der SLAVE-Rechner soll eine Datei empfangen und kann die temporaere Datei TM..., wo die eingehenden Daten zwischengespeichert werden, nicht errichten.

   Bediener:

   Pruefen des Spool-Directorys (Zugriffsrechte,...)

2. Der SLAVE-Rechner soll eine Datei senden und kann die zu sendende Datei nicht zum Lesen eroeffnen.

   Bediener:

   Pruefen der betreffenden Datei (Zugriffsrechte)

- COPY (FAILED)

Beim Kopieren der empfangenen Datei vom temporaeren Datenfile an den gewuenschten Zielort trat ein Fehler auf (Zugriffsrechte). Die empfangene Datei wird dabei meist in das Directory /usr/spool/uucp/uucppublic/user kopiert, wobei "user" der Name des Nutzers ist, der die Uebertragung angewiesen hatte.

Bediener:

Pruefen des gewuenschten Zielortes (Zugriffsrechte),

evtl. kopieren der Datei aus dem Directory /usr/spool/uucp/uucppublic/user an den Zielort.

- COPY (SUCCEEDED)

  Die empfangene Datei wurde ornungsgemaess an den geforderten Zielort kopiert, das temporaere Datenfile geloescht.

- BAD READ (expected 'C' got FAIL)

  Beim Lesen einer Steuermitteilung (erwartet wurde laut Protokoll die Mitteilung "C") wurde nichts empfangen. Es trat eine Unterbrechung der Verbindung auf.

- BAD READ (expected 'C' got 'S')

  Laut Protokoll wurde die Steuermitteilung "C" erwartet. Anstatt von "C" wurde "S" empfangen (Verletzung des Protokolls).

  Ursache: Stoerung der Uebertragung (logisch, physisch).

- NO (AVAILABLE DEVICE) od. NOT (AVAILABLE DEVICE)

  Fuer die gewuenschte Verbindung wurde kein geeignetes Geraet gefunden.

  Bediener:

  Pruefen der Dateien L.sys und L-devices.

- WRONG TIME TO CALL (sysname)

  Die aktuelle Zeit faellt nicht in die fuer eine Verbindung zum System sysname erlaubte Zeitspanne (festgelegt in der Datei L.sys)

  Bediener:

  Kontrolle des Zeitfeldes der entsprechenden Eintragung in L.sys.

- FAILED (LOGIN)

  Der Versuch des MASTER-Systems, sich am SLAVE-System anzumelden (login) ist fehlgeschlagen (Beginn des Verbindungsaufbaus, erste Kontaktaufnahme mit dem

entfernten System).

Bediener:

Pruefung der Leitung (Steckverbindung). evtl. Pruefung der Geraetezuordnung (L.sys, /dev, /etc/inittab).

Pruefen, ob fuer beide Rechner die gleiche Geschwindigkeit vorgesehen ist.

- LOST LINE (LOGIN)

Beim Lesen der Login-Information vom Slave-Rechner trat ein Fehler auf (Phase des Anmeldens des Master- Rechners am SLAVE-Rechner)

Bediener:

Pruefen der Leitung.

- SQLOCK (CAN'T LOCK)

Beim Errichten des Lockfiles LCK.SQ trat ein Fehler auf.

Bediener:

Pruefen der Spool-Directory. Falls dort ein Datei LCK.SQ einer frueheren abnormal beendeten Verbindung zurueckgeblieben ist, muss sie geloescht werden

- NO CALL (MAXRECAUS)

Es wurden mehrmals (10) erfolglose Versuche hintereinander unternommen, eine Verbindung zu einem entfernten System aufzubauen.

Bediener:

Pruefung der Fehlerursache. Loeschen der Datei STST...

- XQT DENIED (cmd)

Das Kommando cmd , dessen Ausfuehrung mittels uux angewiesen wurde, ist am Ausfuehrungssystem fuer uux nicht erlaubt.

Bediener:

Pruefung der Liste der erlaubten Kommandos

(/usr/lib/uucp/L-cmd)

- XQT (PATH, cmd)

  Uuxqt hat die Abarbeitung des mittels uux angewiesenen Kommmandos cmd vorgesehen (dies ist keine Meldung ueber eine erfolgreiche Kommandoausfuehrung.

- INIT FAILED

  Das Deaktivieren der Terminalschnittstelle verlief fehlerhaft.

  Bediener:

  Pruefen des Eintrags fuer die Terminalschnittstelle in der Datei /etc/inittab.

5. Implementierung

5.1. Notwendige Zusaetze in der Betiebssystemumgebung

Systemgenerierung

WEGA ist mit einem geraeteeigenem NETWORK-NODE-NAME (max. 7 Zeichen) zu generieren (siehe WEGA-Software/Systemhandbuch Abschn. 5.4.).

Passwordeintrag fuer uucp

Der Passwordeintrag in der Datei /etc/passwd fuer uucp ist zu ueberpruefen, und gegebenenfalls zu aendern.

```
home directory      - /usr/spool/uucp/uucppublic
login shell     - /usr/lib/uucp/uucico
group ID        - system
password
```

Initialisierung des Terminalkanals

Fuer die Kanaele, die fuer die Rechnerkopplung mit UUCP vorgesehen sind (muessen in L-devices vereinbart sein), muss in der Datei /etc/inittab der entsprechende Eintrag geaendert werden. Fuer die entsprechenden Terminalleitungen muss als auszufuehrendes Kommando "/etc/UGETTY" eingetragen werden (anstelle /etc/GETTY). Gegebenenfalls muss die Geschwindigkeit fuer den Kanal noch geaendert werden (siehe getty(M)). Anschliessend sind die Kommandos

```
INIT 2
kill pid(Prozessnummer des alten Prozesses auf dem
         Terminalkanal)
```

auszufuehren.

5.2. Administrative Arbeit waehrend der Nutzung

Waehrend der Nutzung des Programmpaketes UUCP ist es notwendig, regelmaessig bestimmte Taetigkeiten auszufuehren, die das Saeubern der Directories sowie das Aktualisieren der einzelnen Dateien betreffen.

Einbringen neuer Systeme

Falls eine Kopplung mit einem neuen, dem lokalen UUCP-System noch nicht bekannten entfernten Rechner gewuenscht wird, muessen in den entsprechenden Dateien (die die Konfiguration bestimmen) folgende Eintragungen vorgenommen werden (siehe dazu auch Abschn. 3.2.2.).

- In die Datei USERFILE muss fuer das neue System eine Zeile eingetragen werden, die die Zugriffsrechte festlegt.

- Im File L.sys muss eine Zeile fuer das System angefuegt werden.

- Falls eine neue Leitung benutzt wird, ist diese in die Datei L.devices einzutragen. Weiterhin ist fuer diese Leitung auch der Eintrag in der Datei /etc/inittab zu aendern (siehe Abschn. 5.1.).

- Die Datei SQFILE (siehe Abschn. 3.2.2.) muss ebenfalls eine Zeile fuer das neue System enthalten.

Aenderung der Uebertragungsgeschwindigkeit

Eine Aenderung der Uebertragungsgeschwindigkeit auf einer bestimmten Leitung erfolgt durch eine Aenderung der entsprechenden Werte in den Dateien L.sys und L-devices. Weiterhin muss die Geschwindigkeit in der Datei /etc/inittab geaendert werden (siehe getty(M)). Die Einstellung ueber das Kommando stty ist fuer UUCP nicht ausreichend. Weiterhin muessen natuerlich die Geschwindigkeiten von 2 miteinander kommunizierenden Rechnern uebereinstimmen. Fuer die Kopplung von Wega-Systemen sind die Geschwindigkeiten 9600 baud oder 19200 Baud zu waehlen.

Bereinigung der Directories

Das Spool-Directory ist in regelmaessigen Zeitabstaenden auf alte, nicht mehr benoetigte Dateien zu pruefen, um eine Uebersichtlichkeit dieses Directorys zu wahren. Das betrifft auch die Datei LOGFILE. Neue Eintragungen werden an den bestehenden Inhalt hinten angefuegt. Deshalb ist es angebracht, die Datei LOGFILE von Zeit zu Zeit zu loeschen. Sind partielle LOGFILES entstanden (durch parallele Arbeit mehrerer UUCP) koennen diese durch uulog (siehe uulog(1)) in ein LOGFILE vereinigt werden. Das Loeschen nicht mehr benoetigter Dateien kann bequem mittels uuclean (Angabe des Alters der zu loeschenden Dateien moeglich), durch eine selbst erstellte Shell-Prozedur oder einfach durch Benutzung der entsprechenden Kommandos des Betriebssystems WEGA erfolgen.

Hinweis:

- LCK. -Dateien, die wegen einer abnormalen Beendigung von UUCICO im Spool-Directory verblieben sind, verhindern eine erneute Uebertragung zum gleichen System (ueber die gleiche Leitung). Sie muessen vor Anweisung einer neuen Uebertragung geloescht werden.

- Die in der Datei SQFILE (siehe Abschn. 3.2.2.) gespeicherten Sequenzzaehler duerfen nicht willkuerlich veraendert werden (sonst kein erfolgreicher Verbindungsaufbau moeglich - BADSEQUENCE). Diese Sequenzzaehler muessen stets mit den analogen Zaehlern in den entsprechenden Partnersystemen uebereinstimmen.

Da nach Uebertragungen, die mit Fehlerstatus beendet wurden, die Auftraege im Spool-Directory verbleiben, ist es ratsam, periodisch UUCICO zu starten, wodurch diese Auftraege abgearbeitet werden.

5.3. Kopplung mit anderen Systemen

Mit dem Programmpaket UUCP ist auch die Kopplung mit anderen kompatiblen Systemen moeglich. Bei der Kopplung mit den Systemen MUTOS 1700 bzw. MUTOS 1630 ist darauf zu achten, dass die Ubertragungsgeschwindigkeit nur 2400 Baud betragen darf. Weiterhin muss in der Datei L.sys zum Initialisieren eines login-Prozesses ein 'CTRL/Z' (anstelle von 'NL' fuer WEGA-Systeme; siehe auch Abschn. 3.2.2.) gesendet werden. Beispiele sind in der Datei L.sys enthalten.

5.4. Hinweise

1. Fuer die Kopplung sind nur die Terminalkanaele der 16-Bit Rechnerkarte tty4-7 zu benutzen.

2. Das Kommando uucp kopiert vor der Verbindungsaufnahme alle zu uebertragenden Dateien in die Spool-Directory. Bei der Uebertragung sehr grosser Datenmengen ist es zu empfehlen, das Programm uucp mit der Option -c zu starten. Dadurch wird ein Kopieren unterdrueckt und ein Anwachsen der Spool-Directory ueber die Grenzen des /usr Filesystems hinaus wird verhindert.

3. Alle Punkte der Implementierung sind an beiden zu koppelnden P8000 durchzufuehren.

4. Verbindungsleitungen:
   V.24 RxD ist mit TxD des jeweils anderen Rechners zu verbinden. Der Schirm ist einseitig an Masse anzuschliessen.(P8000: RxD-Pin 2, TxD-Pin 3, SG-Pin 7; P8000/compact: RxD-Pin 3, TxD-Pin 2, SG-Pin 7).
   IFSS ED+ ist mit SD- und ED- mit SD+ des jeweils anderen Rechners zu verbinden.
   Steckerbelegungen:
   P8000: ED+ 13, ED- 14, SD+ 10, SD- 19, Bruecke 7-9 innerhalb des Steckers
   P8000/compact: ED+ 14, ED- 13, SD+ 19, SD- 7, Bruecken 9-7 und 10-12 innerhalb des Steckers.

Anlage

Nachrichtenaustausch

Die folgende Darstellung soll einen zusammfassenden Ueberblick ueber den Nachrichtenaustausch waehrend der Verbindung zweier Systeme geben. Die Darstellung ist der Uebersichtlichkeit halber sehr vereinfacht und verfolgt den Nachrichtenaustausch in Ablehnungs- bzw. Fehlerfaellen nicht weiter.

Master		Slave
EOT od. NL (Zeichen, um das login Proramm zu aktivieren)	--->	
	<---	login:
	<---	password:
		(uucico wird gestartet)
	<---	Shere
S'sysname'-Q'seq' (sysname - Systemname seq - Verbindungs-zaehler aus /usr/lib/uucp/SQFILE)	--->	
	<---	ROK (RCB -call back) (RLCK-sysname nicht i.O.)
	<---	P'list' (list -Liste von moeglichen Protokollen)
U'proto' (proto - ausgewaehltes Protokoll)	--->	

```
                Pakettreiber ein

R,S oder X-Nachricht        --->

                            <---  RY, SY bzw. XY
                                  (RN, SN bzw. XN im
                                  Ablehnungsfall)

                          nach RY bzw. SY
                          DATENAUSTAUSCH
                          (Bestaetigung
                          mit CY bzw. CN)

(wenn noch weitere
Auftraege vorhanden
sind, dann wieder
Senden von S,R oder
X-Nachrichten)

(wenn keine Auftraege
mehr vorhanden sind)
H                           --->

                                  (wenn noch Auftraege
                                  sind - senden von
                                     HN
                                  anschliessend Rollen-
                                  tausch)

                                  (wenn keine Auftraege)
                            <---  HY

HY                          --->

                Pakettreiber aus

OOOOO                       <-->        OOOOO
```

C - B e s o n d e r h e i t e n

Vorwort

In dieser Unterlage werden spezielle Hinweise zu Besonderheiten der Sprache C unter dem System WEGA gegeben. Der Leser sollte mit den Basiskonzepten von C und den U8000-Aufrufvereinbarungen (siehe dazu WEGA-Dienstprogramme Band A) vertraut sein.

Jede Installation enthaelt Rechnerabhaengigkeiten, die die Sprache C beeinflussen. Dieses Papier beschreibt die Maschinenabhaengigkeiten und die C-Spracherweiterungen.

Im ersten Abschnitt wird beschrieben was bei der Portierung von C-Programmen zu WEGA zu beachten ist. Dabei werden im einzelnen die Maschinen- und Objektformatabhaengigkeiten, die Routinen 'setret' und 'longret' und die moeglichen Probleme durch die Uebergabe von Parametern in Registern diskutiert.

Die unter WEGA verfuegbaren Erweiterungen der Sprache C werden im zweiten Abschnitt erlaeutert.

Inhaltsverzeichnis Seite

1. Portierung von Programmen zu WEGA

1.1. Einfuehrung

Beim Portieren von C-Programmen muss der Anwender einige Kenntnisse ueber die speziellen Eigenheiten der Maschine, die sein Programm beeinflussen koennen, beruecksichtigen. Dieser Abschnitt beschreibt einige Aspekte, die bei der Uebertragung von Programmen beachtet werden muessen.

1.2. Routinen 'setret' und 'longret'

Wird auf dem P8000 ein C-Programm abgearbeitet, treten bei der Vereinbarung von Registern Probleme auf, wenn 'setjmp' und 'longjmp' verwendet werden. Ein Ersetzen von 'setjmp' und 'longjmp' durch 'setret' und 'longret' und ein Streichen des Registerattributs der Variablenvereinbarungen fuehrt dazu, dass das Programm unter WEGA ausfuehrbar wird.

Der Stackrahmen des P8000 C-Compilers unterscheidet sich vom SM-4 MUTOS. Das P8000 benutzt nur ein Register, das als Frame-Pointer und Stack-Pointer benutzt wird. Es ist nicht moeglich, (wie beim SM-4 MUTOS) die Kette der Unterprogramm-Aufrufe zurueckzugehen, um die Registervariablen zurueckzuerhalten.

1.3. Registerbenutzung fuer Parameteruebergabe

Der U8000 hat einen groesseren Registersatz als der SM-4-Prozessor. Um diese Register effektiv zu nutzen, werden die Parameter beim P8000 in Register geladen. Programme, bei denen die Parameter im Stack abgelegt sind (wie es beim SM-4 der Fall ist) und die von dort herausgeholt werden, arbeiten auf dem P8000 nicht. Bei den meisten Programmen reicht ein neuer Uebersetzungslauf aus, um die Probleme zu beseitigen. Haben jedoch solche Programme eine variable Anzahl von Parametern, muessen zuvor spezielle Zwischenschritte erfolgen. Diese "Spezialbehandlungen" werden in den nachfolgenden Abschnitten beschrieben.

Im Bild 1-1 ist ein Programm mit einer variablen Anzahl von Parametern dargestellt, das auf dem SM-4 laeuft und wo die Argumente aus dem Stack gelesen werden. Dieses Programm kann bis zu zwei Zeigerargumenten haben. Das gleiche Programm ist dann noch einmal im Bild 1-2 aufgefuehrt, allerdings erfolgt hier das Laden der Paramemter in verschiedene Register.

```
/*
**   Dieses Programm teilt bis zu zwei Zeichenketten-
**   argumenten Speicherraum zu und kopiert diese dann
```

```
**   in den zugewiesenen Raum. Das erste Argument (na)
**   entspricht der Anzahl der Argumente. Das zweite (ap)
**   und das optionale dritte Argument sind die Zeiger
**   auf die zu kopierenden Zeichenketten. Das Programm
**   gibt einen Zeiger zurueck, der auf den Platz weist,
**   auf den die Zeichenketten kopiert wurden.
**/

char *
copy (na, ap)
char *ap;
{
        register char   *p, *np;
        char            *onp;
        register int    n;
        p = ap;
        n = 0;
        if (*p == 0)
                return 0;
        do
        {
                n++;
        } while (*p++);
        if (na > 1)
        {
                p = (&ap)[1];
                while (*p++)
                        n++;
        }
        onp = np = alloc(n);
        p = ap;
        while (*np++ = *p++)
                continue;
        if (na > 1)
        {
                p = (&ap)[1];
                np--;
                while (*np++ = *p++)
                        continue;
        }
        return onp;
}
```

Bild 1-1 Beispiel eines SM-4 Programmes

```
char *
copy(na, ap1, ap2)
char    *ap1, *ap2;
{
        reg char   *p, *np;
        char       *onp;
        reg int    n;
        p = ap1;
        n = 0;
        if (*p == 0)
```

```
                return 0;
        do
        {
                n++;
        } while (*p++);
        if (na > 1)
        {
                p = ap2;
                while (*p++)
                        n++;
        }
        onp = np = alloc(n);
        p = ap1;
        while (*np++ = *p++)
                continue;
        if (na > 1)
        {
                p = ap2;
                np--;
                while (*np++ = *p++)
                        continue;
        }
        return onp;
}
```

Bild 1-2 P8000 Version des im Bild 1-1 gezeigten SM-4 Programmes

Das Modifizieren von Programmen mit einer variablen Anzahl von Parametern unterschiedlicher Typen ist schwierig. Das Bild 1-3 zeigt ein solches Programm. Es handelt sich hierbei um eine Version der C-Bibliotheksroutine 'printf', die zur Illustration der Uebergabe von Parametern in Registern abgeaendert wurde.

```
#define R7     0 /* prcnt == 0 implies r7 already seen */
#define R5     0 /* prcnt == 0 implies r5 already seen */
#define R3     0 /* prcnt == 0 implies r3 already seen */
#define prmax 5  /* max. Anzahl von Registerparametern */
#define true 1

/*
**   Routine fuer das Ausrichten des Parameterzeigers
**   entsprechend den U8000 Aufrufereinbarungen.
**   Nicht benutzte Register werden uebersprungen.
**   Dies erfolgt in C nur fuer 'long' Parameter,
**   die in Register geladen werden.
*/

zalign(prcnt, ip, stk)
int *prcnt;    /* Parameterzaehler */
int **ip;      /* Zeiger auf das niederwertige Wort
                  eines 'long' Wortes */
int *stk;      /* Adresse des ersten Parameter im Stack */
```

```
{
     int t;
          /* 'long' kann nicht in r6 oder r4 beginnen */
     if (*prcnt == R7 || *prcnt == R5)
{
     (*prcnt)++; /* ueberspringe unbenutzte Register */
     (*ip)++;
}
else if(*prcnt == R3) /* 'long' kann nicht in r2 starten */
{
     *prcnt += 2;    /* ueberspringe r2 */
     *ip = &(*stk);  /* Parameter kommt aus dem Stack */
     return;
}

/*   Austauschen der Reihenfolge der Worte in einem
     'long' Wort, die beim Einschreiben in den
     Lokalspeicher invertiert abgelegt wurden. */

t = **ip;
**ip = *(*ip + 1);
*(*ip + 1) = t;
}


/*
**   Im folgenden Beispiel wird eine Routine angegeben,
**   die eine variable Anzahl von Parametern verschiedener
**   Groesse benutzt. Es handelt sich sich dabei um ein
**   Muster einer formatierten E/A-Routine.
*/

     printz (fmt, r6, r5, r4, r3, r2, stack)
     register unsigned char  *fmt;
                    /*Zeiger auf Formatzeichenkette */
     int   r6, r5, r4, r3, r2;
                     /* in Reg. uebergebene Parameter */
     int  stack;    /* erster Parameter im Stack */
     {
       int pr6; /* Speicher fuer Parameterregister 6 */
       int pr5; /* die Reihenfolge der Speicher- */
       int pr4; /* vereinbarung fuer die Parameter- */
       int pr3; /* register zeigt zwei Effekte:  */
       int pr2; /* erstens, 'long' - Worte lassen ihre */
                /* Worte austauschen; zweitens, der Zeiger */
                /* auf den Parameterspeicher kann fuer */
                /* Parameter in den Registern und im Stack */
                /* inkrementiert werden */
       int prcnt; /* Anzahl der vorhandenen Parameter */

     int i;
     union { int  *ip;  long  *lp; } x;

     /* rette die Registerparameter in den Speicher */
     pr6 = r6;
     pr5 = r5;
```

```
     pr4 = r4;
     pr3 = r3;
     pr2 = r2;
     x.ip = &pr6;
     prcnt = 0;
     while (true)
     {    /* einmal fuer jedes Formatzeichen */
       i = *fmt++;
       switch(i)
       {
          case ' ': return; /* end of format */
          case '%': i = *fmt++;
                    switch(i)
                    {
                    case 'd': putint(*x.ip++);
                              break;
                    case 'D': if (prcnt < prmax)
                               zalign(&prcnt,&x.ip,&stack);
                              putlong(*x.lp++);
                                /*second word done below*/
                              prcnt++;
                              break;
                    case 'c': putchar(*x.ip++);
                              break;
                    default:  putchar('%');
                              putchar(i);
                              break;
                    }
                    prcnt++;
                    if (prcnt == prmax)
                       /* start using stack parameters */
                       x.ip = (int *)&stack;
                    break;
          default: putchar(i);
                   break;
       }
    }
}

main ()
{
printz("%c",'z');
printz("double: %D",1L);
printz("decimal: %d",69);
printz("%c%c%c%c%c%c%c","a','b','c','d','e','f','g');
printz("%D %D %D %D",100L,123456L,1L,98765432L);
printz("%D %d %c %d",32L,10,'x',52);
}
```

Bild 1-3 P8000-Programm mit einer variablen Anzahl von Argumenten unterschiedlichen Typs

1.4. Objektformatabhaengigkeiten

Programme, die aus der Objektdatei heraus Informationen aus dem Kopf auswerten, muessen modifiziert werden. Typische MUTOS-Dienstprogramme, die die Objektdateien durchmustern und auch dem P8000-Nutzer zur Verfuegung stehen, sind z.B. 'make' und 'nlist'. Eine vollstaendige Beschreibung des P8000-Objektcodeformats liegt in a.out(5) vor.

1.5. Byteanordnung im Wort

Die Anordnung der Bytes ist im P8000 eine andere als beim SM-4. Im P8000 befindet sich das hoeherwertige Byte eines Wortes auf einer geraden Adresse, wohingegen sich das niederwertige Byte auf der naechsthoeheren ungeraden Adresse befindet. Beim SM-4 ist diese Reihenfolge genau vertauscht. Das bedeutet, dass SM-4 Programme, die Bytes innerhalb eines Wortes manipulieren oder 'long'-Groessen mittels Zeiger manipulieren, auf dem P8000 nicht ordnungsgemaess arbeiten. Das gleiche gilt auch beim Transfer von Dateien zwischen einem P8000 und einem SM-4, wo dann die einzelnen Worte der Datei byteweise getauscht werden muessen.

Beispiel: Im Speicher befindet sich eine 8-Byte lange Zeichenkette, beginnend auf der Adresse 100:

```
00, 01, 02, 03, 04, 05, 06, 07
```

Auf beiden Rechnern belegt die Zeichenkette die Adressen 100 ... 107. Betrachtet man jedoch den Wort-Wert der Adresse 102, so befindet sich beim P8000 auf der Adresse 102 das hoeherwertige Byte 02, so dass man in diesem Fall den Wert 0203 erhaelt. Beim SM-4 ist 03 das hoeherwertige Byte, so dass man bei gleicher Adressierung das Wort 0302 erhaelt. Durch:

```
char *p;
int  i;
i = (*p++*256) + *p++;
```

erhaelt man verschiedene Resultate auf beiden Maschinen. Um das Problem des Datentransfers zwischen beiden Maschinen besser zu veranschaulichen, sei die angegebene Zeichenkette im SM-4 als Struktur, die aus vier einzelnen Bytes, gefolgt von zwei Worten besteht, betrachtet

```
100: 00
101: 01
102: 02
103: 03
104: 0504
105: 0706
```

Wird nun die Zeichenkette zum U8000 transferiert, so entsteht:

```
100 = 00
101 = 01
102 = 02
103 = 03
104 = 0405
105 = 0607
```

Daran ist zu erkennen, dass vor der Verarbeitung die Bytes der Worte auf den Adressen 104 und 106 vertauscht werden muessen, waehrend die Bytes der Adressen 100 bis 103 unveraendert bleiben koennen.

1.6. Rechnerarchitekturabhaengigkeiten

Eine weitere zu beachtende Besonderheit betrifft die Benutzung des /dev/mem Device.Beim SM-4 beginnt der Speicher der System-Daten mit der Adresse 0 von /dev/mem. Beim P8000 beginnt der Speicher der System-Befehle mit der Adresse 0. So dass ein Programm wie z.B. 'ps', das die Adressen im System-Daten-Speicher untersucht, anstelle von /dev/mem (mem(4)) jetzt das Device /dev/kmem benutzen muss.

Die -n Option, die sich auf die 8K-'Pages' des SM-4 bezieht, wird nicht unterstuetzt. Das P8000 hat 64K-'Pages', so dass dafuer die -i Option (getrennte I&D) benutzt werden kann. Beide Optionen verbinden ein Programm so, dass mehrere Kopien desselben Programms sich die ersten 'Pages' teilen koennen.

1.7. Merkmale des C-Compilers

Der WEGA-C-Compiler gestattet Registervariable des Typs short, int, pointer(Zeiger),long und double. Diese Registervariablen koennen auch vorzeichenlos sein. Die Vereinbarungen register char werden ignoriert. Im nichtsegmentierten Mode stehen fuer die Registervariablen sieben Register zur Verfuegung. Im segmentierten Mode reduziert sich die Anzahl der Register auf sechs.

Die Groesse der verschiedenen Variablentypen ist wie folgt:

Type	Groesse (in Bits)
character	8
unsigned character	8
short	16
unsigned short	16
int	16
unsigned int	16
pointer (nonsegmented)	16
pointer (segmented)	32
long	32
unsigned long	32
float	32
double	64
register double	80 (IEEE format)

Obwohl intern 80 Bits fuer die Darstellung von 'double'-Variablen in Registern benutzt werden, bedeutet das nicht, dass auch das Ergebnis aus 80 Bits bestehen muss. So werden z.B. in der Anweisung

```
register double  d=1.1;
```

nur 64 Bits fuer die Initialisierung von d mit 1.1 benutzt.

Beim Umwandeln von SM-4 C-Programmen in P8000 C-Programme, ist es wichtig zu wissen, dass der SM-4 C-Compiler (CC) keine Vorzeichenerweiterung ausfuehrt, wenn Zeichen (char) durch einen 'cast'-Operator als unsigned gekennzeichnet werden. SM-4 Programme, die Ausdruecke der Form

```
(unsignded) C
```

enthalten, wobei C ein Zeichen ist, muessen in

```
(unsigned character) C
```

geaendert werden, um die Vorzeichenerweiterung beim P8000 zu unterdruecken.

2. C-Erweiterungen

2.1. Allgemeines

Gegenueber dem ueblichen C-Sprachumfang wurden einige Erweiterungen aufgenommen. Diese sind nachfolgend aufgefuehrt.

2.2. Zuweisung einer Struktur

Strukturen koennen zugewiesen, als Argumente an Funktionen uebergeben werden und auch von diesen zurueckgegeben werden. Die dabei verwendeten Operanden muessen vom gleichen Typ sein.

Anmerkung:
Bei Funktionen, die Strukturen als Ergebnis zurueckliefern, gibt es allerdings bei der WEGA-Implementation von C eine Einschraenkung. Tritt innerhalb einer Returnbehandlung ein Interrupt auf und wird dabei dieselbe Funktion noch einmal aufgerufen, kann der Wert, der durch den ersten Aufruf zurueckgegeben wird, verfaelscht sein. Dies ist nur fuer Programme von Bedeutung, bei denen echte Interrupts auftreten koennen (z.B. im Betriebssystem oder in Nutzerprogrammen, die von Signalen gebrauch machen). Gewoehnliche rekursive Aufrufe werden davon nicht beruehrt.

2.3. Elementnamen von Strukturen und Unions

Elementnamen von Strukturen und Unions werden werden jetzt eindeutig identifiziert, d.h. ein und derselbe Name kann fuer Elemente unterschiedlicher Strukturen und Unions verwendet werden. Nachfolgend ist ein einfaches Beispiel angegeben:

```
/*
**      Das folgende Beispiel zeigt die Benutzung von
**      benannten strukturierten Feldern innerhalb von
**      Strukturen oder Unions.
**      Der Wert des "all"-Feldes soll 00010203 hex sein
*/

main ()
{
     union
     {
          struct
          {
            int j;
            int k;
          } sl;

          struct
          {
```

```
            int p;
            char j;
            char k;
          } s2;

          long all;

     } u,*p;

p = &u;
p->s1.j = 1;
p->s2.j + 2;
p->s2.k + 3;
}
```

Bild 2-1 Einfaches Beispiel der Benennung von strukturierten Feldern

2.4. Aufzaehlungstyp

Dieser Typ entspricht dem Skalartyp in PASCAL. Die nachstehend aufgefuehrte Syntax definiert diesen Typ.

enum-specifier:

```
enum { enum-list }
enum identifier { enum-list }
enum identifier
```

enum-list:

```
enumerator
enum-list, enumerator
```

enumerator:

```
identifier
identifier = constant-expression
```

Die Bedeutung des unter "enum-specifier" angegebenen Bezeichners ist aehnlich der Struktur ("structure tag") in einem Strukturbezeicher. Er benennt eine besondere Aufzaehlung.

Beispiel:

```
enum color { gruen, rot, gelb, blau }
...
enum color *cp, col;
```

Im angegebenen Beispiel ist "color" eine Aufzaehlung von verschiedenen Farben; cp ein Zeiger und col ein Objekt.

Die Bezeichner in der enum-Liste werden als Konstanten vereinbart und koennen ueberall wo Konstanten benoetigt

werden, auftreten. Falls keine Aufzaehlungen mit dem Zeichen "=" auftreten, beginnen die Werte der Konstanten mit dem Wert 0 und werden um 1 inkrementiert, in der Reihenfolge des Auftretens (von links nach rechts). Eine Aufzaehlung mit dem "="-Zeichen weist dem entsprechenden Bezeichner den angezeigten Wert zu. Nachfolgende Bezeichner setzen die Progression des zugewiesenen Wertes fort.

In der WEGA-Implementation werden alle Aufzaehlungsvariablen als ganze Zahlen ("Integer") behandelt.

2.5. Datentyp void

Der neue "void"-Datentyp gestattet einer Routine ohne Ergebnis zurueckzukehren. Dieser Datentyp macht es ueberfluessig, solche Routinen so zu vereinbaren, dass sie eine ganze Zahl ("Integer") als Ergebnis liefern. Sollte versucht werden, das Ergebnis solcher Routinen zu benutzen oder so eine Routine doch mit einem Rueckgabewert zu beenden, wird eine Fehlermeldung ausgegeben.

KOMBINAT VEB
ELEKTRO-APPARATE-WERKE
BERLIN-TREPTOW
»FRIEDRICH EBERT«

HEIM-ELECTRIC
EXPORT-IMPORT
Volkseigener Außenhandelsbetrieb
der Deutschen Demokratischen Republik

EAW-Automatisierungstechnik Export-Import

Storkower Straße 97
Berlin, DDR - 1055
Telefon 432010 · Telex 114158 heel dd

VEB ELEKTRO-APPARATE-WERKE BERLIN-TREPTOW
»FRIEDRICH EBERT«
Stammbetrieb des Kombinats EAW
DDR - 1193 Berlin, Hoffmannstraße 15-26
Fernruf: 2760
Fernschreiber: 0112263 eapparate bln
Drahtwort: eapparate bln