UDOS-Software
Mikroprozessorsoftware

EAW *electronic*

P8000

transcribed in ~12h by O. Lehmann during 2006-06-18 and 2006-06-30
Version 1.7 (2015-04-24)

Diese Dokumentation wurde von einem Kollektiv des Kombinates

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

erarbeitet.

 Im Interesse einer staendigen Weiterentwicklung werden die Nutzer gebeten, dem Herausgeber Hinweise zur Verbesserung mitzuteilen.

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

WAE/03-0102-01

Ausgabe: 12/86

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 enthaelt folgende Unterlagen:

- U880-ASM/PLINK Benutzerhandbuch
  (Bearbeiter: R.Haupt)

- U8000/881-PLZ/ASM Benutzerhandbuch
  (Bearbeiter: R.Kuehle)

- ZLINK Benutzerhandbuch
  (Bearbeiter: L.Mielenz)

- IMAGER Benutzerhandbuch
  (Bearbeiter: L.Mielenz)

U 8 8 0 - A S M / P L I N K

Benutzerhandbuch

Inhaltsverzeichnis Seite 1

1. U880-Assembler

Fuer das 8-Bit-Mikrorechnersystem steht als Entwicklungssoftware der U880-Assembler zur Verfuegung. Der U880-Assembler laeuft unter dem Betriebssystem UDOS.
Aus einem mittels Editor erfassten Assembler-Quellprogramm wird mit dem U880-Assembler standardmaessig eine verschiebliche Objektdatei erstellt. Neben der Objektdatei wird gleichzeitig eine Listendatei erstellt, die den Quellkode und den dazugehoerigen Objektkode ausweisen.

1.1. Assemblerbedienung

Der Aufruf des U880-Assemblers erfolgt durch folgende Kommandozeile:

```
%ASM Dateiname.S (Optionen) bzw.
%ASM Dateiname_1.S ... Dateiname_n.S (Optionen)
```

Bei der Angabe von mehreren Quelldateien wird nur eine Objektdatei angelegt und zwar mit dem Dateinamen der Quelldatei, die als erste in der Kommandozeile eingegeben wurde. Die Quelldateien muessen mit der Bezeichnung ".S" enden. Die Bezeichnung ".S" muss aber nicht beim Programmaufruf mit eingegeben werden. Optionen sind wahlfrei und werden in Klammern gesetzt. Es koennen eine oder mehrere Optionen, getrennt durch ein Leerzeichen, angegeben werden. Die Reihenfolge ist dabei unbedeutend. Folgende Optionen sind moeglich:

Option	Erlaeuterung
M	Erlaubt Makrobearbeitung, ansonsten werden Makrodefinitionen und -aufrufe als Fehler angezeigt.
S	Es wird eine Symboltabelle erzeugt, die an den Objektmodul angefuegt wird.
X	Veranlasst die Erzeugung einer alphabetisch sortierten Crossreferenzliste am Ende der Listendatei.
A	Veranlasst die Uebersetzung des Moduls mit absoluten Adressen
NOL	Unterdrueckt die Erstellung einer Listendatei.
NOM	Unterdrueckt die Ausgabe von Makroexpansionen.
NOO	Unterdrueckt die Erstellung einer Objektdatei.
NOW	Unterdrueckt die Ausgabe von Fehlerwarnmeldungen.
D-Zeichenkette	Die angegebene Zeichenkette (maximal 18 Zeichen) wird im Listenkopf jeder Seite untergebracht.
P	Veranlasst die Ausgabe der Listendatei ueber die logische Ein-/Ausgabeeinheit Nr.3 des Betriebssystems.

Eine Aenderung der Namens- bzw. logischen Geraetezuweisung kann mit folgenden Optionen erfolgen:

Option	Erlaeuterung
I-Dateiname	Zwischendatei
L-Dateiname	Listendatei
M-Dateiname	Makrodatei
O-Dateiname	Objektdatei
T-Dateiname	Symboluebertaufdatei
X-Dateiname	Crossreferenzdatei

Die Erstellung der Objektdatei und Listendatei erfolgt in zwei Durchlaeufen (Pass 1 und Pass 2). Wurde die Option S oder X eingegeben, so erfolgt fuer die Generierung der Symbol- bzw. Crossreferenzdatei noch ein dritter Durchlauf (Pass 3). Die Assemblerlaeufe haben im einzelnen folgende Wirkung:

Pass 1:
- Identifikation der Anweisungen
- Pruefung der Anweisungen auf formale Richtigkeit
- Ausgabe einer Zwischendatei auf Systemdiskette
- Abschlussausschrift "Pass1 COMPLETE"

Pass 2:
- Erstellung der Objektdatei
- Erstellung der Listendatei

Pass 3:
- Erstellung einer sortierten Symboltabelle

Am Ende des Assemblerlaufes erscheint die Ausschrift

"n ASSEMBLY ERRORS"
"ASSEMBLY COMPLETE"

Beispiele fuer den Programmaufruf ASM:

%ASM TEST

- erstellt die Dateien TEST.OBJ und TEST.L auf der Anwenderdiskette

%ASM TEST (O=0/TEST S P NOL)

- erstellt die Objektdatei TEST1.OBJ mit anschliessender Symboltabelle auf der Diskette im Laufwerk 0. Der Aufbau der Listendatei wird unterdrueckt. Die Ausgabe der Liste erfolgt auf dem Bildschirm

% ASM TEST1 TEST2 (X)

- erstellt von beiden Dateien eine gemeinsame Objektdatei TEST1.OBJ und eine Listendatei TEST1.L mit einer Crossreferenzliste

1.2. Sprachbeschreibung

Die Struktur des U880-Assemblerbefehles hat folgenden Aufbau:

Namensfeld, Operationsfeld, Operandenfeld und Kommentarfeld. Die Abgrenzung des Namensfeldes, des Operationsfeldes und des Operandenfeldes kann durch ein oder mehrere Leerzeichen oder ein oder mehrere Tabs erfolgen.

Namensfeld:

Das Namensfeld besteht aus einem oder mehreren alphanumerischen Zeichen, wobei max. 6 alphanumerische Zeichen beruecksichtigt werden. Das erste Zeichen des Namensfeldes muss ein Buchstabe sein. Das Namensfeld kann mit einem ":" abgeschlossen werden.

Operationsfeld:

Das Operationsfeld besteht aus dem Befehlsmnemonik oder einem Pseudobefehl.

Operandenfeld:

Das Operandenfeld besteht aus einem oder mehreren durch Komma getrennte Operanden, z.B.:

- Registeroperanden A, B, C, D, E, H, L, I, R, IX, IY, BC, DE, HL, SP, AF, AF'
- Adressoperanden
- Direktwerte (binaere, oktale, dezimale, hexadezimale Darstellung)

Eine komplette Tabelle ueber alle Operationsfeld/Operandenfeld-Kombinationen ist in dem Buch "Wissensspeicher Mikrorechnerprogrammierung" von Classen, Oefler zusammengestellt.

Kommentafeld:

Das Kommentarfeld beginnt mit einem Semikolon und kann bis zur 80. Spalte fuer die Erlaeuterung des Assemblerbefehles genutzt werden. Die Fortsetzung der Erlaeuterung auf einer naechsten Zeile muss mit einem Semikolon beginnen.
Die Notation eines Assemblerbefehles kann sowohl in Grossbuchstaben als auch in Kleinbuchstaben erfolgen.

1.2.1. Ausdruecke

Der U880- Assembler uebersetzt logische und arithmetische Ausdruecke in einer Reihenfolge von links nach rechts. In der nachfolgenden Tabelle sind saemtliche Operatoren entsprechend ihres Vorranges aufgelistet:

Operator	Funktion	Vorrang
+	unaeres Plus	1
-	unaeres Minus	1
.NOT. oder \	logische Negation	1
.RES.	Result	1
**	Potenzierung	2
*	Multiplikation	3
/	Division	3
.MOD.	Modulo	3
.SHR.	logische Rechtsverschiebung	3
.SHL.	logische Linksverschiebung	3
+	Addition	4
-	Subtraktion	4
.AND. oder &	logisches Und	5
.OR. oder ^	logisches Oder	6
.XOR.	exklusives Oder	6
.EQ. oder =	gleich	7
.GT. oder >	groesser als	7
.LT. oder <	kleiner als	7
.UGT.	vorzeichenloses groesser als	7
.ULT.	vorzeichenloses kleiner als	7

Anhand von Beispielen soll die Wirkung einiger Operatoren dargestellt werden.

Beispiel Vorzeichenoperator:

```
LD A,-2 ist identisch mit LD A,0FEH
```

Der Operand -2 als 8-Bit-Groesse ergibt in der Zweierkomplementendarstellung den Wer 0FEH.

Beispiel Vergleichsoperator:

Die Vergleichsoperatoren .EQ., .GT., .LT., .UGT. und .ULT. liefern als Ergebnis ein logisches TRUE (-1) oder ein logisches FALSE (0), wenn der Vergleich wahr oder falsch ist.

```
1.EQ.2 liefert den wert (0) FALSE
(2+2).LT.5 liefert den Wert (-1) TRUE
```

Beispiel Resultatsoperator:

Die Anwendung dieses Operators verhindert einen fehlerhaften Ueberlauf bei der Berechnung eines Argumentes.

```
LD BC, 7FFFH+1 wuerde einen Ueberlauffehler verursachen,
LD BC,.RES.(7FFFH+1) dagegen nicht.
```

Beispiel Modulooperator:

Der Modulooperator wird definiert als

```
X.MOD.Y = X -Y*(X/Y)
(X/Y) stellt eine Integerdivision dar.
8.MOD.3 ergibt als Ergebnis den Wert 2
```

1.2.2. Pseudobefehle

Pseudobefehle haben das gleiche Basisformat wie die Assemblerbefehle. Als Operationskode fuer die Pseudobefehle koennen auftreten:

(Marke:) DEFB Ausdruck
Generiert ein Byte

(Marke:) DEFW Ausdruck
Generiert ein Wort (2 Byte)

(Marke:) DEFM 'Zeichenkette'
Generiert eine Folge von Bytes, die den ASCII-Wert eines jeden Zeichens der Zeichenkette widerspiegelt.

(Marke:) DEFT 'Zeichenkette'
Aehnlich DEFM, nur dass zusaetzlich am Anfang ein Byte eingefuegt wird, das die Laenge der Zeichenkette angibt.

(Marke:) DEFS Ausdruck
Reserviert die im Ausdruck angegebene Anzahl von Bytes

(Marke:) END
Ende des Quellprogrammes (kann auch weggelassen werden)

Name: EQU Ausdruck
Weist dem Namen den Wert des Ausdruckes zu. Der Name darf im Programm nicht neu definiert werden.

Name: DEFL Ausdruck
Analog EQU, dem gleichen Namen kann aber durch eine weitere DEFL-Anweisung ein neuer Wert zugewiesen werden.

GLOBAL Name1, Name2, ...
Kennzeichnet die angegebenen Operanden als globale Groessen, die in anderen Quelltextmodulen als externe Groessen verwendet werden koennen.

EXTERNAL Name1, Name2, ...
Kennzeichnet die angegebenen Operanden als externe Groessen, die in anderen Quelltextmodulen als globale Groessen definiert worden sind.

(Marke:) ORG Ausdruck
Setzt den Befehlskodezaehler auf den Wert des Ausdrucks.

(Marke:) COND Ausdruck
ENDC
Die zwischen COND und ENDC eingeschlossenen Quelltextzeilen werden nur uebersetzt, wenn der Wert des Ausdrucks wahr, d.h. ungleich Null, ist. (bedingte Uebersetzung).

Makroname MACRO #P0,#P1,...,#Pn
(Marke:) ENDM
Die zwischen MACRO und ENDM eingeschlossenen Anweisungen bilden den zum Makronamen gehoerenden Makrokoerper (siehe Abschn. 1.2.3.).

Beispiele:

```
          ZAEHL:    DEFB 31H
```

Der Inhalt der Adresse ZAEHL erhaelt den Wert 31H.

```
          E21:      DEFM 'ERR21'
```

Das Feld mit dem Namen "E21" wird mit den ASCII-Werten 45,52,52,32,21 gefuellt.

```
                    DEFS 50
```

Der momentane Adresszaehler wird um 50 Bytes erhoeht und somit ein Speicherbereich von 50 Bytes reserviert.

```
          E22:      DEFT 'ERR22'
```

Das Feld mit dem Namen E22 wird mit dem Wert 05 (Anzahl der ASCII-Zeichen) und den ASCII-Zeichen 45,52,52,32,32 gefuellt.

```
          WERT:     EQU  0
```

Die Konstante "WERT" ist mit Null identisch.

1.2.3. Makros

Fuer die Programmerarbeitung immer wiederkehrender Programmabschnitte gibt es zwei Moeglichkeiten:

- Makros und
- Unterprogramme

Sowohl Makros als auch Unterprogramme werden nur einmal geschrieben, koennen jedoch ueber Aufrufe beliebig oft abgearbeitet werden. Der Unterschied zwischen beiden Methoden besteht darin, dass im Assemblerlauf jedes Unterprogramm als separater Bestandteil uebersetzt wird, Makros dageben in ihrer ganzen Laenge bei jedem Aufruf ins Programm eingefuegt werden.

Makrodefinition:

```
     Name:     MACRO #P0,#P1,...,#Pn
     (Name:)   ENDM
```

In der Makro-Pseudoanweisung wird das Symbol "Name" ein Makro mit den aktuellen Werten P0,...,Pn zugeordnet. Die Gesamtheit der Anweisungen, die zwischen "MACRO" und "ENDM" stehen, wird als Modellanweisung bezeichnet.

```
Beispiel  INIT:     MACRO #ADR #ZAEHL #WERT
                    LD   B, #ZAEHL
                    LD   HL, #ADR
          M1        LD   (HL), #WERT
                    INC  HL
                    DJNZ M1
```

ENDM

Hinweise, die zu beachten sind:

- beliebig viele Parameter sind erlaubt
- "#" muss vor jedem Parameter stehen und darf in Makros sonst nicht verwendet werden
- "Name" darf die Zeichen BLANK, KOMMA, SEMIKOLON und NUMMERNZEICHEN nicht enthalten
- Makrodefinition muss vor Makroaufruf erfolgen
- Makroaufrufe innerhalb eines Makros sind erlaubt
- Makrodefinitionen innerhalb einer Makrodefinition sind nicht zulaessig
- Parameternamen duerfen an jeder beliebigen Stelle der Modellanweisung auftreten

Bei der Assemblierung werden Makros zunaechst in einer Makrozwischendatei ohne Uebersetzung abgelegt. Erst mit dem Makroaufruf erfolgt eine Zuordnung der aktuellen Werte und die Uebersetzung des angesprochenen Makros in den Objektkode.

Makroaufruf:

Name 'S0', 'S1', ..., 'Sn'

'S0' bis 'Sn' stellen die aktuellen Werte fuer die Formalparameter "P0 bis Pn" dar, "Name" ist der Makroname

```
Beispiel  INIT BER3 100 0     ;Aufruf des Makros "INIT"
          LD   B, 100
          LD   HL, BER3
N1:       LD   (HL),  0       ;Einfuegen des Makros in
                              ;das Objektprogramm
          INC  HL
          DJNZ N1
          ENDM
```

Die aktuellen Parameter koennen beliebige Zeichenfolgen sein, getrennt durch Kommata oder Leerzeichen. Die Zuweisung der aktuellen Parameter beim Aufruf entspricht der durch die Formalparameter "P0 bis Pn" vorgegebenen Reihenfolge. Entspricht die Anzahl der aktuellen Parameter nicht der Anzahl der Formalparameter, so erhalten die fehlenden Parameter den Wert 0.

Beispiel

```
TRANS:    MACRO #ADR1 #ADR2 #LAEN #REGHL
          COND REGHL.GT.0
          PUSH HL
          ENDC
          LD   DE, #ADR1
          LD   HL, #ADR2
          LD   BC, #LAEN
          LDIR
          COND #REGHL.GT.0
          POP  HL
          ENDC
```

```
          ENDM
```

Makroaufruf a)

```
          TRANS BER1 BER2 100 1
          COND 1.GT.0
x)        PUSH HL
          ENDC
x)        LD   DE, BER1
x)        LD   HL, BER2
x)        LD   BC, 100
x)        LDIR
          COND 1.GT.0
x)        POP  HL
          ENDC
          ENDM
```

Makroaufruf b)

```
          TRANS BER1 BER2 100      ,#REGHL nicht angefuehrt
          COND 0.GT.0
x)        PUSH HL
          ENDC
x)        LD   DE, BER1
x)        LD   HL, BER2
x)        LD   BC, 100
x)        LDIR
          COND 0.GT.0
x)        POP  HL
          ENDC
          ENDM
```

x) angekreuzte Befehle werden uebersetzt

Symbolgenerator:
Der Formalparameter "#SYM" kann in einer Modellanweisung an jeder beliebigen Stelle implizit angewandt werden. Bei der Uebersetzung wird der Parameter durch eine 4-stellige Hexadezimalkonstante ersetzt, die bei jedem neuen Makroaufruf um 1 erhoeht wird. auf diese Weise koennen verschiedene Marken in unterschiedlichen Aufrufen eines gleichen Makros eingesetzt werden. (Mehrfachdefinition in einer Marke fuehrt zur Fehlermeldung)

Beispiel

```
          LOESCH:   MACRO #ADR #ZAEHL #WERT
A1#SYM:   LD   (HL), #WERT
          DJNZ A1#SYM
          ENDM
```

1. Aufruf:

```
          LOESCH BER3 100 0
A10000:   LD   (HL), 0
          DJNZ A10000
          ENDM
```

2. Aufruf

```
          LOESCH BER3 100 0
```

```
A10001:   LD   (HL), 0
          DJNZ A10001
          ENDM
```

Rekursion:

Innerhalb eines Makros koennen Aufrufe fuer andere Makros erfolgen. Jeder rekursive Makroaufruf erzeugt eine weitere Makroexpansion

z.B.:

```
MACR1:    MACRO #A #B
SYMB:     DEFL #B
          MACR2     '#A'      ;Rekursiv-Makro "MACR2"
          ENDM

MACR2:    MACRO #A
          COND SYMB.GT.0
SYMB:     DEFL SYMB-1
          MACR2     '#A-'
          #A
          ENDC
          ENDM
```

Makroaufruf:

```
MACR1     RLA  3
SYMB:     DEFL 3
          MACR2     'RLA'
          COND SYMB.GT.0
SYMB:     DEFL SYMB-1
          MACR2     'RLA'
          COND SYMB.GT.0
SYMB:     DEFL SYMB-1
          MACR2     'RLA'
          COND SYMB.GT.0
SYMB:     DEFL SYMB-1
          MACR2     'RLA'
          COND SYMB.GT.0
SYMB:     DEFL SYMB-1
          MACR2     'RLA'
          RLA
          ENDC
          RLA
          ENDC
          RLA
          ENDC
          RLA
          ENDC
```

1.3. Assembler-Kommandos

Die Assembler-Kommandos stehen im Quellprogramm und dienen im wesentlichen zur Steuerung des Listing-Formats. Sie muessen mit einem Stern "*" und ab Spalte 1 beginnen. Notwendige Argumente sind durch ein oder mehrere Leerzeichen, Tabs oder Kommata getrennt aufzufuehren. Fuer den U880-Assembler gelten folgende Kommandos:

- *Elect	Fortsetzung auf einer neuen Seite
- *Heading s	die Zeichenkette s (max. 28 Zeichen) wird als Ueberschrift ueber jede neue Seite gedruckt
- *Include Dateiname	Einfuegen der Datei "Dateiname" in das aktuelle Quellprogramm
- *List OFF	die Listenausgabe wird unterdrueckt
- *List ON	die Listenausgabe wird fortgesetzt
- *Maclist OFF	die Listenausgabe eines Makroblockes und eines Makroaufrufes wird unterdrueckt
- *Maclist ON	Listenausgabe aller Makros
- *Page n	n gibt die Anzahl der Zeilen pro Seite an (Standard sind 58 Zeilen pro Seite)

1.4. U880-Assembler-Errors

Beim Assembler-aufruf koennen folgende Fehler auftreten:

INVALID OPTION: s

Der unter s angegebene Text wurde nicht als gueltige Optionsangabe anerkannt. Ursache koennte eine fehlende Optionsspezifikation oder eine falsche Abgrenzung der Optionen sein.

MEMORY TO SMALL

Der fuer den Assemblerlauf benoetigte Speicherbereich ist nicht ausreichend. Der benoetigte Speicherbereich ist abhaengig von den vereinbarten Optionen.

LINE TO LONG: Dateiname

Eine oder mehrere Quelltextzeilen in der Datei "Dateiname" sind laenger als 128 Zeichen.

FILE NOT FOUND: Dateiname

Die angegebene Quelltextdatei mit dem Namen "Dateiname" ist in dem Direktory nicht gefunden worden.

INVALID ATTRIBUTES: Dateiname

Die in der Quelltextdatei "Dateiname" verwendeten Attribute sind nicht vom Typ ASCII, die Satzlaenge betraegt 128 Byte.

INVALID FILE: Dateiname

Die Zeichenkette "Dateiname" ist kein korrekter Dateiname.

I/O ERROR e ON UNIT u

Ein- bzw. Ausgabefehler der Fehlerklasse "e" auf dem logischen Geraet "u". Ausfuehrliche Erlaeuterungen sind in der UDOS-Dokumentation beschrieben.

2. U880-PLINK

Der U880-Plinker verarbeitet einen oder mehrere Objektmodule, die durch einen Assembler oder einen PLZ/SYS-Compiler erstellt wurden und gibt ein einzelnes Programm in Form einer ausfuehrbaren Prozedurdatei aus. Der Plinker besorgt die Umwandlung der Module und loest die Beziehungen zwischen den separat assemblierten Modulen auf. Zusaetzlich koennen eine Ladelistendatei und eine binaere Symboltabelle erstellt werden.
Der U880-Plinker laeuft unter dem Betriebssystem UDOS und ermoeglicht damit dem Anwender die Steuerung aller Ein-/Ausgabemoeglichkeiten.

2.1. Anwendung des U880-PLINK

Der Plinker wird vom Betriebssystem UDOS durch das Kommando PLINK, gefolgt von einer Liste der Objektdateinamen und Optionen, aufgerufen.

```
%PLINK Dateiname * (Optionen)
```

Dabei bedeutet:
- * es koennen ein oder mehrere Objektdateinamen angegeben werden
- () die Angabe in Klammern ist wahlweise

Der Linkerlauf erfolgt in zwei Phasen.
In der ersten Phase baut der Linker das Bindeverzeichnis auf, das die Zuweisung von absoluten Adressen und globalen Namen ermoeglicht. Externe Namen muessen teilweise geloest werden (d. h. Zuweisung der absoluten Adresse zu den entsprechenden globalen Namen).
In der zweiten Phase wird der Objektkode jedes Moduls verarbeitet, lokale Bezuege werden umgeordnet, externe Bezuege werden aufgeloest, und eine Phasendatei und eine Datei der Ladeliste werden erstellt. Eine wahlweise dritte Phase wird ausgefuehrt, wenn eine Symboldatei zu erstellen ist.
Am Ende des Linklaufes wird der Fehlerzaehler auf dem Bildschirm angezeigt:

```
n ERRORS
```

"n" ist die Zahl der mehrfach definierten globalen Symbole und der ungeloesten externen Bezuege. Die letzte Meldung ist:

```
LINK COMPLETE
```

Der Linklauf kann durch Eingabe eines "?" unterbrochen werden. Durch erneute Eingabe eines "?" wird die Abarbeitung fortgesetzt.
Zwei Fehlermeldungen koennen beim Linklauf auf dem Bildschirm angezeigt und in die Listendatei geschrieben werden:

```
MULTIPLY - DEFINED GLOBAL IN MODULE:
Symbolname          Modulname
```

Diese Meldung zeigt, dass der Symbolname als GLOBAL in zwei verschiedenen Modulen vereinbart worden ist.

```
UNRESOLVED EXTERNAL IN MODULE:
Symbolname          Modulname
```

Diese Meldung zeigt, dass ein Mnemonik, der als EXTERNAL vereinbart worden ist, keinen entsprechenden GLOBAL-Namen hat.
Mit $ = X bzw. $ = $+X, wobei X eine hexadezimale Zahl darstellt, werden den Modulen absolute Speicherplaetze zugewiesen. Jedem Modul kann separat eine feste Adresse zugeordnet werden. Dies hat beim Testen von Programmabschnitten einen Vorteil, denn die zu testenden Module koennen auf gerade Adressen gelinkt werden. Es ist nur darauf zu achten, dass keine Speicherueberlagerungen der einzelnen Module auftreten, denn sonst kommt folgende Ausschrift auf den Bildschirm:

```
POSSIBLE CODE OBERLAY AT n IN Modulename
```

"n" ist die Adresse, auf die der Modul gelinkt wurde.
Ist die Testphase beendet, so ist es ratsam, den verschieblichen Modulen aufeinanderfolgende Speicherbereiche zuzuordnen, da sonst zu viel Speicherplatz benoetigt wird.

Beispiel
Drei Module sollen zwecks Test in drei verschiedenen geraden Adressbereichen gelinkt werden. Der Datenbereich soll ebenfalls ab einer geraden Adresse beginnen.

```
PLINK $=2000 M1 $=3000 M2 $=4000 M3 $=8000 DATEN
```

Die Module M1, M2, M3 werden jeweils auf die Adressen 2000H, 3000H, 4000H und die Daten auf die Adresse 8000H gelinkt.
Im naechsten Beispiel sollen die einzelnen Module ab der Adresse 2000H aufeinanderfolgend und die Daten ab Adresse 8000H gelinkt werden.

```
PLINK $=2000 M1 M2 M3 $=8000 DATEN
```

Diese Linkanweisung gilt nur fuer verschiebliche Module. Bei Programmen, die bereits mit festen Adressen assembliert worden sind, entfaellt diese Linkanweisung.
Der Linker kann auf Wunsch bis zu 16 nicht unmittelbar aufeinanderfolgende Segmente im Speicher generieren, deren Groesse stets ein ganzzahliges Vielfaches der Aufzeichnungslaenger der Prozedurdatei ist.
Fuer die Realisierung von Ueberlagerungsstrukturen (s. Abschn. 2.4.) ermoeglicht der Linker, sogenannte Linkmodule zu generieren. Hierbei werden den Modulen nur Startadressen zugewiesen und EXTERNAL- und GLOBAL- Symbole aufgeloest. Die Generierung eines Maschinenkodes in der Prozedurdatei

entfaellt. Module, die nur gelinkt werden sollen, erhalten ein Minus vor ihren Namen.

Beispiel
PLINK $=2000 RESIDENT - OVERLAY.2

Der Modul RESIDENT erhaelt eine Prozedurdarei mit dem Maschinenkode. Der Module OVERLAY.2 wird nicht in den Maschinenkode umgesetzt. Mit Hilfe der Segmentierung und der Generierung von Linkmodulen lassen sich mittels Ueberlagerungsstruktur Programme implementieren, die ein gegebenes Speichervolumen ueberschreiten.
Mit Hilfe der Modulidentifikation, kann der Dateiname komplett oder nur teilweise mit dem Laufwerksnamen oder der Laufwerkspezifikation angegeben werden.

Beispiele
1) PLINK MODA MODB.OBJ
2) PLINK $SYDOS: 2/MULTIPLY

Das erste Beispiel zeigt die Verwendung der unvollstaendigen Dateinamen MODA und MODB.OBJ, die auf dem Master-Laufwerk gefunden werden koennen.
Das zweite Beispiel zeigt die Verwendung eines vollstaendigen Dateinamens (MULTIPLY), der im Laufwerk 2 des Geraetes SYDOS gefunden wird.
Jeder Objektkode muss vom Typ "binaer" sein und eine Satzlaenge von 128 Bytes haben. Zusaetzlich muss ein Objektmodul Informationen beinhalten, die ihn identifizieren. Der Linklauf wird abgebrochen, wenn ein Modul nicht die korrekte Identifikation enthaelt. Der Dateiname jedes Objektmoduls muss mit dem Anhang ".OBJ" in Gross- oder Kleinbuchstaben enden. Ist dieser Anhang in der Kommandozeile nicht mit angegeben, so ergaenzt der Linker automatisch den Anhang, bevor die Prozedurdatei eroeffnet wird. Die Ausgabedatei des Linklaufs enthaellt den Namen der ersten Objektdatei in der Kommandozeile, ohne den Anhang ".OBJ". Die Listendatei und die Symboldatei haben den gleichen Namen wie die Prozedurdatei, jedoch mit dem Anhang ".MAP" bzw. ".SYM".

Beispiel
PLINK PROG (SY)

In diesem Beispiel sucht der Linker den Objektmodul PROG.OBJ, erzeugt die Dateien PROG, PROG.MAP und PROG.SYM.

PLINK mod1.obj mod2 TTY.INT

In diesem Beispiel sucht der Linker den Objektmodul mod1.obj, mod2.obj und TTY.INT.OBJ, erzeugt die Dateien mod1 und mod1.map.

2.2. PLINK-Optionen

Wenn Optionen verwendet werden, dann sind sie in Klammern zu setzen. Sie werden in der Kommandozeile nach dem Dateinamen gesetzt. Das Format der Option ist ein Schluessel-

wort, das von einem Gleichheitszeichen (=) und einem entsprechenden Wert oder Namen gefolgt werden kann. Die signifikanten ersten Buchstaben des Schluesselwortes werden in der nachfolgenden Beschreibung gross geschrieben.

Entry = n
Spezifiziert die Startadresse fuer die Abarbeitung der Prozedurdatei. "n" kann ein hexadezimaler Wert oder ein GLOBAL-Name sein. Wenn die Option nicht angegeben ist, wird die Anfangsadresse des ersten Objektmoduls als Startadresse genommen. Falls ein GLOBAL-Name spezifiziert worden ist, aber nicht gefunden wird, erfolgt eine Fehlermeldung und die Option wird ignoriert.

LET

Die vollstaendige Ausfuehrung des Linklaufes wird auch bei einem Fehler in den Symboldefinitionen erzwungen. Wird die Option nicht angegeben, fuehren Fehler nicht zur Erstellung der Phasendatei.

Map = Dateiname

Der Linker erzeugt eine Liste mit den Startadressen und Laengen der einzelnen Module, sowie alle alphabetisch geordneten GLOBAL's mit ihren zugeordneten Adressen. Der Dateiname spezifiziert eine Datei, in der die Ladeliste und die Fehlermeldungen (keine Abbruchfehler) ausgegeben werden. Ist kein Dateiname angegeben, so wird die Ladeliste mit dem Anhang ".MAP" und dem gleichen Dateinamen wie die Phasendatei erzeugt.

Name = Dateiname

Der Linker weist der Prozedurdatei den angegebenen Namen zu. Sonst entspricht dieser dem ersten Modulnamen ohne Anhang ".OBJ".

NoMap

Vom Linker wird keine Datei mit der Ladeliste erstellt.

NOWarning

Der Linker unterdrueckt die Ausgabe von Meldungen ueber mehrfach definierte GLOBAL's oder ungeloeste EXTERNAL's.

Print

Diese Option bewirkt die Ausgabe der Ladeliste und Fehlermeldungen auf dem logischen Geraet SYSLST. Standardausgabe ist die Ausgabe auf dem Bildschirm.

RLength = n

Diese Option spezifiziert die Aufzeichnungslaenge einer Prozedurdatei. "n" kann die hexadezimalen Werte 100, 200 und 400 annehmen. Als Standardwert wird fuer "n" 100H

gesetzt.

STacksize = n

Mit dieser Angabe wird die Groesse des Stackbereiches festgelegt. "n" ist die Anzahl in Bytes fuer den Stackbereich. Standardmaessig wird fuer den Stack ein Bereich von 80H Byte festgelegt.

SYmbol = Dateiname

Diese Angabe erzeugt eine binaere Symboldatei fuer die absolten GLOBAL's und internen Symbole jedes Objektmoduls fuer die Anwendung der symbolischen Fehlersuchprogrammme. Fehlt die Angabe des Dateinamens, so erhaelt die Datei den gleichen Namen wie die Prozedurdatei mit dem Anhang ".SYM".
Hinweis:
Die Option NOM geht vor MAP = Dateiname, die Option NOW schliesst LET ein.
Falls eine Prozedurdatei nicht erfolgreich generiert wurde, wird keien Symboltabelle erstellt und die Listendatei enthaelt nur die Fehlermeldungen.
Wird vom Anwender eine Ausgabedatei mit der Laenge 0 spezifiziert (z.B. bei einer Arbeitsdatei), so wird diese Datei nach dem Linklauf geloescht.

Beispiel

```
PLINK $=1000 PROG1
```

Dieses Kommando bewirkt die Erstellung einer Prozedurdatei mit dem Namen PROG1 (Startadresse =1000H) und eine Listendatei PROG1.MAP.

```
PLINK PROG1 (N=$MYDOS:2/PROG.RUN SY NOM P E=MAIN)
```

Dieses Kommando erstellt eine Procedurdatei mit dem Namen PROG.RUN (Startadresse ist gleich der zugewiesenen Adresse des GLOBAL's MAIN). Ferner wird eine Symboldatei mit dem Namen PROG.RUN.SYM auf dem Laufwerk 2 des Geraetes MYDOS angelegt, sowie die Ladeliste und die Fehlermeldungen auf SYSLST geschrieben.

Definition der logischen Geraete:
Der Linker verwendet folgende logischen E/A-Geraete:

2	CONOUT	Fehlermeldungen
3	SYSLST	Ausgabe der Print-Option
4		Objektdateien
5		Phasendatei (Prozedurdatei)
6		Listdatei
7		Symboldatei

2.3. Listenformat

Die zugewiesenen Anfangsadressen und die Laenge jedes Modols sowie eine alphabetische Liste aller GLOBAL's mit den zugehoerigen Adressen und den Modulen, die sie enthalten, werden in einer vom Linker erzeugten Ladeliste abgelegt.

Zusaetzlich enthaelt die Liste den Programmnamen, seine Laenge und die Startadresse.

z.B.:
```
     PLINK  $=1000  MODA  MODB  $=2000  TAN.MATH  COT.MATH
            (N=VERARBEITUNG B=BEGINN)
```

Der Linker wuerde bei diesem Kommando folgende Ladeliste erstellen:

```
PLINK 1.0

LOAD MAP
MODULE              ORIGIN          LENGTH

MODA                1000            0A73
MODB                1A73            0319
TAN.MATH            2000            00A3
CON.MATH            20A3            00C5

GLOBAL              ADDRESS         MODULE

TAN                 2000            TAN.MATH
BEGINN              1085            MODA
COT                 20A3            COT.MATH
SQRT                1A73            MODB
VEKTOR              150D            MODA

PROGRAMM VERARBEITUNG -- 1168 BYTES
```

2.4. Ueberlagerungen

Dieser Abschnitt gibt einige Hinweise fuer den Anwender, der sein Programm als Ueberlagerungsstruktur implementieren muss. Die Verwaltung der Ueberlagerungsstruktur muss durch das Anwenderprogramm realisiert werden. Dabei ist zu beachten, dass das Laden eines Ueberlagerungssegments erfolgen muss, bevor zu diesem Segment Beziehungen hergestellt werden.
Eine Ueberlagerungsstruktur besitzt ein residentes Segment, das die Daten und Routinen enthaelt, die gemeinsam von den verschiedenen Segmenten benoetigt werden, d.h., die Routine, durch die ein anderes Segment geladen wird, befindet sich normalerweise in dem residenten Segment.
Das Problem einer Ueberlagerungsstruktur laesst sich an einem Beispiel darstellen.

Ein Programm (BEISPIEL) liest eine grosse Anzahl von Werten von einer Quelle (LESEN), reorganisiert und berechnet neue Daten (VERARBEITEN) und gibt diese Daten wieder aus (SCHREIBEN). Es wird angenommen, dass der verfuegbare Speicherbereich nur fuer ein Segment und die gemeinsamen Routinen ausreicht. Wenn Daten und Routinen als GLOBAL und EXTERNAL in jedem der vier Module, die das Programm enthalten, deklariert sind, kann der Anwender diese in einzelnen Linklaeufen wie folgt kombinieren:

Das Segment LESEN passt zusammen mit dem residenten Segment

BEISPIEL in den Speicher. Die beiden werden zusammen zu einer Phasendatei gebunden. Es ist anzumerken, dass der residente Modul Bezuege zu den Segmenten VERARBEITEN und SCHREIBEN enthaelt. Diese Module werden als Linkmodule verarbeitet, um die Aufloesung der Bezuege zu ermoeglichen. Fuer die Realisierung dieser Ueberlagerungsstruktur sind folgende Linkkommandozeilen zu notieren:

```
1)    PLINK BEISPIEL $=5000 LESEN $=5000 -VERARBEITEN
                    $=5000 -SCHREIBEN

2)    PLINK -BEISPIEL $=5000 SCHREIBEN (N=SCHREIBEN)

3)    PLINK -BEISPIEL $=5000 VERARBEITEN (N=VERARBEITEN)
```

Die beiden Ueberlagerungssegmente werden erstellt durch jeweiliges Linken mit dem residenten Modul, wobei der residente Modul BEISPIEL als Linkmodul verarbeitet wird.
Die Phasendateien VERARBEITEN und SCHREIBEN werden durch das Programm selbst waehrend der Abarbeitung geladen.

2.5. PLINK-Errors

Es gibt verschiedene Ursachen, die die weitere Abarbeitung des Linklaufes abbrechen. Wird der Linklauf abgebrochen, dann werden alle Dateien geschlossen, um zu vermeiden, dass die Diskette inkonsistent wird. Bei Abbruch des Linklaufes koennen folgende Meldungen auf dem Bildschirm erscheinen:

```
     INVALID OPTION: s
```

Die Zeichenkette s in der Kommandozeile ist keine gueltige Linkoption.

```
     LINK DIRECTORY OVERFLOW
```

Diese Fehlernachricht zeigt an, dass fuer die Ausfuehrung des Linklaufes der verfuegbare Speicherbereich nicht ausreichend ist. Das Linkverzeichnis ist grundsaetzlich aus GLOBAL's und EXTERNAL's fuer jeden Objektmodul zusammengesetzt. Dies benutzt der Linker zum Verschieben udn Aufloesen von Objektkodebeziehungen. Zusaetzlich sind Bereiche fuer die Arbeit mit Dateien erforderlich. Der Bereich fuer das Bindeverzeichnis kann klein gehalten werden, wenn die Zahl und die Laenge der globalen Symbole reduziert und die Zahl der EXTERNAL's minimiert werden. Diese Fehlernachricht erscheint auch, wenn mehr als 127 Module gelinkt werden sollen.

```
     PROGRAM TOO BIG
```

Diese Fehlernachricht hat zum Inhalt, dass der aktuelle Adresszeiger ueber hexadezimal FFFF erhoeht wurde. Der Programmbereich oder die Anfangsadresse muesste reduziert werden.

```
     TO MANY SEGMENTS
```

Mehr als 16 Segmente sollen erzeugt werden. Dieser Fehler kann eintreten, wenn der aktuelle Adresszeiger oft verringert wird, um ein neues Segment zu erzeigen.

FILE NOT FOUND: dateiname

Das Objekt "Dateiname" wurde nicht in dem Direktory gefunden.

INVALID FILE: Dateiname

Die Zeichenkette "Dateiname" ist kein korrekter Dateiname.

INVALID FORMAT: Dateiname

Das Objekt "Dateiname" hat Attribute, die nicht vom Typ "binaer" sind.

INVALID DATA: Dateiname

Das Objekt "Dateiname" enthaelt ungueltige Daten, wie CRC-Zeichenfehler oder einen Symbolnamen, dessen Laenge entweder null oder groesser als der maximal zulaessige Wert von 127 Zeichen ist. Durch erneute Assemblierung sollte versucht werden, den Fehler zu beheben.

I/O ERROR e ON UNIT u

Ausgabe der Fehlerklasee e auf dem logischen Geraet u. Naehere Erlaeuterungen sind aus der Dokumentation des Betriebssystems UDOS zu entnehmen.

U 8 0 0 0 / U 8 8 1 - P L Z / A S M

PLZ/ASM - Assembler

Inhaltsverzeichnis Seite

1. Einleitung

Der groesste Teil der Mikroprozessorsoftware wird derzeit in Assemblersprache erstellt. Daneben gewinnen aber auch hoehere Programmiersprachen fuer Mikroprozessoranwendungen immer mehr an Beduetung. Die Programmierpsrache PLZ/ASM stellt eine Erweiterung der bisher bekannten "reinen" Assemblersprache mit Elementen von hoeheren Programmiersprachen dar.
Diese Elemente unterstuetzen den Programmierer in den Fragen der Datenstrukturierung, der Programmablaufsteuerung und im modularen Programmaufbau. Das Konzept von PLZ/ASM fuer die Prozessoren U8000 und U881/U882 ist nahezu identisch. Entsprechende Uebersetzerprogramme unter dem Betriebssystem UDOS sind verfuegbar.
Die Programmiersprache PLZ/ASM ist Bestandteil der PLZ-Sprachfamilie. Innerhalb dieser Familie ist die hoehere Programmiersprache PLZ/SYS hervorzuheben, die zur Entwicklung von Systemprogrammen (Compiler, Dateihandler, Betriebssystemkomponenten u.a.m.) entworfen wurde.

2. Sprachkonzept

Innerhalb der nachfolgenden Abschnitte wird zur syntaktischen Darstellung eine modifizierte Backus-Naur-Darstellung verwendet. Syntaktische Konstruktionen werden durch Texte, die in "<","<" eingeschlossen sind, gekennzeichnet. Moegliche Wiederholungen einer Konstruktion werden durch Anhaengen von

+ (ein- oder mehrmalige Wiederholung) oder
* (null- oder mehrmalige Wiederholung) angezeigt.

Basissymbole der Sprache werden entweder gross geschrieben (Schluesselwoerter) oder in Apostrophe (fuer Symbole) gesetzt.
Runde Klammern sind Metasymbole und werden benutzt, um eine Gruppe von Konstruktionen zusammenzufassen, auf die ein Wiederholungssymbol folgt. Die Moeglichkeit des Weglassens einer Konstruktion wird durch Einschliessen in die Metasymbole [und] angezeigt. Die Sprachelemente werden anhand von Beispielen erlaeutert.

2.1. Strukturierung von PLZ/ASM-Programmen

PLZ/ASM-Programme bestehen aus einem oder mehreren getrennt uebersetzbaren Modulen. Diese werden unter Benutzung des Verbinderprogramms (LINKER) zu einem ausfuehrbaren Programm verbunden. Dabei enthaelt ein Modul das "Hauptprogramm". Dies ist eine als global gekennzeichnete Prozedur, die beim Verbinden als Eintrittspunkt angegeben wird.
Module bestehen aus Assembler- und hoeherer Sprachanweisungen, die entweder Daten vereinbaren, definieren oder Aktio-

nen beschreiben. Ausfuehrbare Anweisungen muessen innerhalb von Prozeduren erscheinen. Prozeduren erhalten ebenso wie Module Namen, so dass sie ueber CALL-Anweisungen aufrufbar sind. Eine Prozedur kann lokale Variablenvereinbarungen enthalten. Neben den Assembleranweisungen koennen innerhalb einer Prozedur zur Steuerung des Programmablaufs auch hoehere Sprachelemente, wie: DO-Schleifen, REPEAT-, EXIT-, IF-, CASE-Anweisungen verwendet werden.
Auf Datenvereinbarungen, Marken und Prozeduren kann von einem anderen Modul aus zugegriffen werden. Sie muessen in dem Modul, in dem sie definiert sind, als GLOBAL gekennzeichnet sein und in dem Modul, in dem auf sie zugegriffen wird, als EXTERNAL. Wenn sie als INTERNAL vereinbart sind, so ist ihr Gueltigkeitsbereich nur der Modul selbst. Der Gueltigkeitsbereich lokaler Groessen (LOCAL) ist die Prozedur, in der sie definiert sind.
Der groesste Teil eines PLZ/ASM-Programmes werden natuerlich Assembleranweisungen sein. die hoeheren Sprachelemente dienen zur besseren Strukturierung und Selbstdokumentation der Quellprogramme. Hoehere Sprachelemente realisieren zwei Grundfunktionen:

Vereinbaruzng und Definition von Daten
Definition der Programmstruktur (Module, Prozeduren)

Assemblersteueranweisungen steuern die Art der Uebersetzung (absolut oder verschieblich, bedingte Uebersetzung u.a.m.). Sie werden durch Angabe eines Zeichens "$" als erstes Zeichen gekennzeichnet und koennen an beliebiger Stelle innerhalb eines Moduls stehen. Beispiele, die die Struktur von PLZ/ASM-Programmen verdeutlichen werden im Abschn. 2.4. angegeben.

2.2. Datenvereinbarungen

Daten (Konstanten und Variablen) muessen vor ihrer Benutzung definiert oder vereinbart werden. Im allgemeinen vorvindet eine Datendefinition einen namen mit einem festen Wert oder Typ. Datenvereinbarungen fuehren einen Bezeichner als Namen einer Variablen ein und legen deren Typ und Gueltigkeitsbereich fest. Folgende drei Anweisungen werden benutzt, um Daten zu definieren und zu vereinbaren:

Konstantendefinition (CONSTANT)
Typdefinition (TYPE)
Variablenvereinbarung

2.2.1. Konstantendefinition

Durch eine Konstantendefinition erhaelt ein Name den Wert eines Konstantenausdrucks. Alle zur Definition verwendeten Namen muessen vorher definiert sein.
Der Gueltigkeitsbereich einer Konstanten ist der Modul, in dem sie definiert wurde (INTERNAL).

Eine Konstantendefinition hat folgende Syntax:

```
CONSTANT
     (<Konstantenname> ':=' <konstanter_Ausdruck>)+
```

Beispiel:

```
CONSTANT
     Zeilen := 24
     Spalten := 80
     ANZAHL := Zeilen*Spalten
     Zeichen_x := 'x'
```

2.2.2. Typedefinition

Datentypen werden mit einer Variablen verbunden, um deren Groesse zu kennzeichnen, oder um einen Namen als Marke zu definieren. Datentypen koennen entweder direkt in einer Variablendeklaration verwendet werden, oder sie koennen ueber einen vom Anwender ausgewaehlten Typnamen, der in einer Typanweisung definiert wurde, zur Kennzeichnung von Variablen herangezogen werden.
Eine Typdefinition hat folgende Syntax:

```
TYPE
     (<Typname> <Typ)*
```

Es werden einfache und strukturierte Datentypen unterschieden. Einfache Datentype sind:

SHORT_INTEGER oder BYTE	ein Byte (8 Bit)
INTEGER oder WORD	ein Wort (16 Bit)

Beim U8000 kommt noch dazu:

LONG_INTEGER oder LONG	zwei Worte (32 Bit)

Beispiele fuer einfache Datentypdefinitionen:

```
TYPE
     CHAR  BYTE
     DIGIT CHAR
     pointer WORD
```

An strukturierten Datentypen werden unterschieden:

ARRAY	Zusammenfassung von Komponenten eines Typs. Adressierung erfolgt ueber einen Index (0 bis N-1, bei N Elementen)
RECORD	Zusammenfassung von Komponenten (genannt Felder) unterschiedlichen Typs. Der Zugriff erfolgt ueber Feldnamen.

Beispiele fuer strukturierte Datentypvereinbarungen:

```
TYPE
     TEXT ARRAY [64 BYTE]

INTERNAL
     titel TEXT
     !titel definiert ein 64 Byte grosses Feld!

TYPE
     adr RECORD [nr WORD name TEXT]

INTERNAL
     s2 adr
     !adr definiert einen 66 Byte grossen Bereich!
     !auf  die  Feldelemente von s2 kann  ueber  die
      Notation s2.nr und s2.name zugegriffen  werden!
```

2.2.3. Variablenvereinbarungen

Variablenvereinbarungen werden benutzt, um den Typ von Variablen zu spezifizieren und ihnen wahlweise Anfangswerte zu zuweisen. Eine Variablenvereinbarung hat folgende Syntax:

```
<Variablenname>+  <Typ> [':=' <Anfangswerte>]
```

Sie koennen in folgenden Kategorien erfolgen: GLOBAL, EXTERNAL, INTERNAL (im Modulniveau) oder LOCAL (im Prozedurniveau). Dieses Format ist jedoch durch den Gueltigkeitsbereich und den angegebenen Typ beschraenkt. Externe Variablenvereinbarungen duerfen keine Anfangswertzuweisungen enthalten.
fuer die Anfangswertzuweisung gibt es verschiedene Regeln und spezielle Symbole mit einer besonderen Bedeutung. Wenn eine einzelne Vereinbarung mehr als einen Variablennamen enthaelt, so muessen die aufgelisteten Anfangswerte in eckige Klammern eingeschlossen werden. Variablen einfachen Typs werden durch die Angabe von konstanten Ausdruecken initialisiert. Variablen strukturierten Typs werden durch einen sogenannten Konstruktor, eine Liste von in eckigen Klammern eingeschlossenen Werten, initialisiert. Dabei kennzeichnet immer ein Paar von eckigen Klammern ein Verschachtelungsniveau.
Das spezielle Symbol "..." kennzeichnet, dass der vorherige Wert oder Konstruktor fuer den Rest der Variablen im aktuellen Niveau wiederholt wird. Das spezielle Symbol "?" kann in einer Liste von Anfangswerten fuer Variablen oder Komponenten einfachen Typs verwendet werden, um dieser Stelle keinen Anfangswert zuzuweisen. Der leere Konstruktor " " kennzeichnet, dass die entsprechende strukturierte Variable keinen Anfangswert erhaelt.

Beispiele fuer einfache Variablenvereinbarungen:

```
     INTERNAL
          end WORD := %FFFF
          nr,anz,step BYTE := [0...]
          A,B,C BYTE := ['x','y','z']
          D,E,F BYTE := [0,1]    !F ist undefiniert!
```

Eine Variablenvereinbarung vom Typ ARRAY hat folgende Syntax:

```
 <Name>+ ARRAY '['<Dimension>+ <Typ>']'[':='<Anfangswerte>]
```

oder

```
 <Name+ <Feldtyp> [':='<Anfangswerte>]
```

Dabei kennzeichnet Dimension die Anzahl der Dimensionen in der ARRAY-Struktur und die Anzahl der Elemente in jeder Dimension. Typ kann ein einfacher Typ, ein vorher definierter Typbezeichner oder eine ARRAY- oder RECORD-Typdefinition sein. Feldtyp muss ein vorher definierter ARRAY-Typbezeichner sein. Die Anfangswerte sind eine in eckigen Klammern eingeschlossene Liste von konstanten Ausdruecken. Die Initialisierung erfolgt von links nach rechts (z.B.: wird eine 2x2 Matrix in folgender Reihenfolge:

```
          0,0   0,1   1,0   1,1
```

initialisiert.
Bei einem eindimensionalen Feld kann anstelle der Dimension auch das Zeichen "*" verwendet werden. In diesem Fall wird die Feldlaenge aus der Groesse der Liste der Anfangswerte ermittelt. Dabei kann dann auch anstelle der in eckigen Klammern eingeschlossenen Anfangswerte eine in Hochkomma eingeschlossene Zeichenfolge verwendet werden.

Beispiele:

```
     INTERNAL
          matrix  ARRAY [20 20 WORD]
          liste   ARRAY [8 BYTE] := [1,1,0,0]
          text    ARRAY [5 BYTE] := ['N','E','Y','E','R']
          tab1,tab2 ARRAY [2 BYTE] :=[[0...][1...]]
          liste1  ARRAY [* BYTE] := [1,1,0,0]
          !liste1  ist ein Feld von 4 Bytes; das Feld liste
           ist  ein  Feld von 8 Bytes,  von denen nur  vier
           initialisiert sind!
          text1   ARRAY [* BYTE] : 'MUELLER'
```

Eine Variablenvereinbarung vom Typ RECORD hat folgende Syntax:

```
<Name>+ RECORD '['(<Feldname>+<Typ>)+']'[':='<Anfangswerte>]
```

oder

```
<Name>+ <Recordtyp> [':='<Anfangswerte>]
```

Typ kann wiederum ein einfacher Typ, ein vorher definierter Typbezeichner oder eine ARRAY- oder RECORD-Typdefinition sein. Recordtyp muss ein vorher definierter RECORD-Typbezeichner sein, Falls ein ARRAY oder ein RECORD als Komponenten auftreten, so wird diese Verschachtelung durch Einschliessen in eckige Klammern gekennzeichnet. Die Zuordnung der angegebenen Anfangswerte erfolgt von links nach rechts.

Beispiele:

```
 GLOBAL
  person RECORD [alter,groesse,gewicht BYTE
                 geburt RECORD [tag,monat,jahr BYTE]
                 pkz WORD]
  text RECORD [laenge BYTE zeichen ARRAY[64 BYTE]]:=[0[0]]

  !laenge und das erste Zeichen des Feldes werden mit Null
   initialisiert!

 TYPE
  patient RECORD [zimmer WORD
                  geburt RECORD [tag,monat,jahr BYTE]
                  alter,geschlecht BYTE]

 INTERNAL
  weiblich ARRAY [150 patient] := [[?,[],?,'W']...]

  !nur geschlecht eines jeden Records wird initialisiert!
```

Das Uebersetzerprogramm (Assembler) sorgt bei Variablen einfachen Typs automatisch dafuer, dass alle Werte groesser 6 Bit (WORD, LONG) auf geraden Speicheradressen beginnen. Dies erfolgt ebenfallls mit Anweisungen innerhalb einer Prozedur. Bei strukturierten Variablen muss der Programmierer selber auf diese Ausrichtung achten!

2.2.4. Markenvereinbarung

Neben den bisher beschriebenen Vereinbarungen existiert noch eine Markenvereinbarung. Durch sie wird ein angegebener Name als Markenname in einem Programm gekennzeichnet. Er kann in seinem Gueltigkeitsbereich fuer keinen anderen Zweck verwendet werden. Die Syntax einer Markenvereinbarung ist wie folgt:

```
          <Name>+  LABEL
```

Bei der Definition wird dem Namen kein Doppelpunkt nachgestellt. Die Markenvereinbarung kann nicht dazu benutzt werden, den Marken absolute Adressen zuzuweisen. Eine Marke kann folgende Gueltigkeitsbereiche haben: GLOBAL, EXTERNAL, INTERNAL oder LOCAL. Falls eine Marke in einem ausfuehrbaren Teil verwendet wird, ohne dass sie durch eine Markenvereinbarung spezifiziert wurde, so ist ihr Gueltigkeitsbereich der Modul, in dem sie definiert wurde (INTERNAL). Falls eine Marke in einer Prozedur lokal sein soll, so muss sie vor dem ENTRY-Schluesselwort als lokale Marke vereinbart werden. Der Gueltigkeitsbereich von Marken der Form $n (n-Dezimalzahl) ist immer nur die Prozedur, in der sie definiert wurden.

Beispiel:

```
     GLOBAL
          m2 LABEL

          bsp PROCEDURE       !Prozedur ist global!
              LOCAL
                    .
                    .
              m1 LABEL
              ENTRY
                 m1:          !m1 ist lokal zu bsp!
                    .
                    .
                 m2:          !m2 ist global!
                    .
                    .
                 m3:          !m3 ist intern zum Modul!
                    .
                    .
                 $1:          !$1 ist lokal zu bsp!
                    .
                    .
          END bsp
```

2.2.5. SIZEOF-Operator

Dem Anwender steht noch ein spezieller unaerer Operator (SIZEOF) zur Verfuegung, der auf Typnamen angewendet werden kann und als Wert die Groesse in Bytes liefert.

Beispiele:

```
     TYPE
          char BYTE
          zahl char
          feld ARRAY [5 5 WORD]
          patient RECORD [alter, groesse BYTE
                             zimmer WORD]

                                    Wert:
          SIZEOF digit                1
```

```
SIZEOF matrix            50
SIZEOF patient            4
SIZEOF patient.zimmer     2
```

2.3. Programmstrukturierende Anweisungen

2.3.1. Modulvereinbarung

Die Modulvereinbarung hat folgendes Format:

```
<modulname> MODULE
     vereinbarung
END <modulname>
```

wobei 'vereinbarung' eine Daten- oder Prozedurvereinbarung ist und <modulname> einem Namen entspricht.

2.3.2. Prozedurvereinbarung

Eine Prozedurvereinbarung definiert einen ausfuehrbaren Programmteil und verbindet dieses mit einem Namen, so dass er ueber eine CALL-Anweisung aufgerufen werden kann. Im Prozedurkopf wird der Prozedurname spezifiziert. Er kennzeichnet die erste Anweisung der Prozedur und kann wie jede andere Programmarke verwendet werden. Prozeduren koennen in folgenden Kategorien definiert werden: GLOBAL, INTERNAL oder EXTERNAL. Bei einer externen Prozedurvereinbarung wird nur der Prozedurname angegeben, da die Definition der Prozedur in einem anderen Modul erfolgt. Eine Prozedurvereinbarung kann auch lokale Variablenvereinbarungen enthalten. Der Gueltigkeitsbereich dieser Variablen ist dann nur die Prozedur, in der sie definiert wurden.
Eine Prozedurvereinbarung hat folgende Syntax:

```
<Prozedurname> PROCEDURE
     [LOCAL
          (<Variablenname>+ <Typ>)* ]*
     [ENTRY
          <Anweisung>* ]
END <Prozedurname>
```

Fuer Anweisung kann dabei eine Assembleranweisung, eine DO-, IF-, EXIT-, oder REPEAT-Anweisung stehen. Das Schluesselwort ENTRY trennt die Vereinbarung von lokalen Variablen vom ausfuehrbaren Teil der Prozedur. Zu beachten ist, dass vor dem Ende der Prozedur eine RET-Anweisung zu programmieren ist.
Zur Ablaufsteuerung innerhalb einer Prozedur existieren folgende hoehere Anweisungen:

2.3.3. DO-Anweisung

Sie liefert die Moeglichkeit, Programmschleifen zu programmieren. Die zwischen den Schluesselwoertern DO und OD eingeschlossenen Anweisungen werden so lange wiederholt, bis durch eine Schleifensteueranweisung ein Abbruch erfolgt. Eine solche Steueranweisung kann eine Assemblersprunganweisung (z.B.: JP, DJNZ), eine EXIT- oder REPEAT-Anweisung sein. Eine DO-Anweisung hat folgende Syntax:

```
[<Marke>]*
DO
  <Anweisung>*
OD
```

Fuer Anweisung kann dabei eine Assembleranweisung, eine DO-, IF-, EXIT- oder REPEAT-Anweisung stehen.

2.3.4. EXIT-Anweisung

Die EXIT-Anweisung veranlasst das Verlassen einer DO-Schleife. Sie liefert die Moeglichkeit, mit der nach der DO-Schleife programmierten Anweisung fortzusetzen. Durch Angabe der Marke einer DO-Anweisung kann auch ein bestimmter zu verlassender DO-Block spezifiziert werden. Sie besitzt folgende Syntax:

```
EXIT [FROM <Marke>]
```

Das Uebersetzerprogramm erzeugt dafuer einen unbedingten Sprung zu der nach dem OD-Schluesselwort folgenden Anweisung. Dabei wird je nach Entfernung ein relativer (JR) oder absoluter Sprung (JP) erzeugt.

2.3.5. REPEAT-Anweisung

Die REPEAT-Anweisung veranlasst einen Neubeginn der Schleife, in der sie erscheint. Durch Angabe der Marke einer DO-Anweisung kann auch ein bestimmter DO-Block spezifiziert werden, bei dem wieder begonnen werden soll. Die Syntax ist analog zur EXIT-Anweisung:

```
REPEAT [FROM <Marke>]
```

Das Uebersetzerprogramm erzeugt dafuer wiederum einen unbedingten Sprung zu dem entsprechenden DO-Schluesselwort. Dabei wird je nach Entfernung ein relativer (JR) oder absoluter Sprung (JP) erzeugt.

Beispiel:

```
M1:  DO
          .
          .
       IF Z THEN REPEAT FI
       DO
          .
          .
          IF GE THEN EXIT FROM M1 FI
          .
          .
       OD
     OD
```

2.3.6. IF-Anweisung

Sie liefert die Moeglichkeit, eine Programmverzweigung zu programmieren. Falls die nach IF folgende Bedingung wahr ist, so werden die zwischen THEN und ELSE (oder zwischen THEN und FI, falls der ELSE-Zweig fehlt) eingeschlossenen Anweisungen ausgefuehrt. Andernfalls werden die zwischen ELSE und FI eingeschlossenen Anweisungen ausgefuehrt. Falls die Bedingung nicht erfuellt ist und kein ELSE-Zweig vorhanden ist, so wird mit der nach FI folgenden Anweisung fortgesetzt. Die IF-Anweisung hat folgende Syntax:

```
IF <Bedingungskode>
THEN <Anweisung1>*
[ELSE <Anweisung2>*]
FI
```

Fuer Bedingungskode kann dabei eine der in den Befehlslosten (Abschnitte 3.1.6. und 4.1.7.) angegebenen Abkuerzungen (z.B.: Z, NZ, C, NC u.a.) stehen.

Beispiel:

```
IF NZ
  THEN CALL UP1; $P Marke1
  ELSE CALL UP2
FI
```

2.3.7. IF/CASE-Anweisung

Die IF/CASE-Anweisung stellt eine Erweiterung der IF-Anweisung dar. Sie gestattet in Abhaengigkeit vom Inhalt eines Selektorregisters eine Mehrfachverzweigung. In einer CASE-Gruppe koennen mehr als ein Ausdruck vorkommen. Nach jeder CASE-Gruppe wird ein unbedingter Sprung zum Ende der Select-Konstruktion eingefuegt. Ein ELSE-Zweig kann als Alternative zu allen CASE-Faellen spezifiziert werden. Falls kein ELSE-Zweig angegeben ist und keine Uebereinstimmung mit einer CASE-Gruppe auftritt, so wird mit der nach FI stehenden Anweisung fortgesetzt. Die IF/CASE-Anweisung hat folgende Syntax:

```
IF <Selektorregister>
  (CASE <Ausdruck>+ THEN <Anweisung>*)+
  [ELSE <Anweisung>*]
FI
```

Fuer Selektorregister steht dabei eine Registerbezeichnung. Der nach CASE anzugebende Ausdruck muss einen gueltigen Operanden in einer Vergleichsanweisung darstellen.

Beispiel:

```
IF R3
  CASE #4  THEN CALL UP1
  CASE #2,R4,JR5  THEN CALL UP2
  ELSE $P error
FI
```

2.3.8. Minimalformulierung eines PLZ/ASM-Programms

Mit den Strukturierungsanweisungen MODULE und PROCEDURE und den Datenvereinbarungsklassen lassen sich arbeitsfaehige Programme formulieren.

Beispiel:

```
<modulname> MODULE

     CONSTANT            !Konstantendefinition!
     INTERNAL            !oder GLOBAL, EXTERNAL!
          a,b  BYTE
          c,d  WORD

     <procedurename> PROCEDURE
          LOCAL
          e,f, BYTE
          ENTRY

          !hoehere Sprachelemente!
          !Assemblerprogramm!

     END <procedurename>

END <modulname>
```

2.4. Programmbeispiele

Als Programmbeispiel wurde ein einfacher Sortieralgorithmus ausgewaehlt. Ausgangspunkt fuer die Sortierprogramme ist ein externes Feld von 10 Woertern. Es wird das Feld 'list' (von list 0 bis list 9) durchgemustert , wobei immer zwei benachbarte Elemente verglichen werden. Falls list [i] > lost [i+1] ist, so erfolgt ein Austausch dieser beiden Elemente, und der Vergleichstest beginnt wieder bei list[0]. Dieser Prozess laeuft so lange ab, bis das gesamte 'Feld der Groesse nach geordnet ist.

2.4.1. U8000

Die Laenge des zu sortierenden Feldes wird der Prozedur 'sort' ueber das Register R0 mitgeteilt. Der Austauschanzeiger ist eine lokale Groesse (ein Byte) innerhalb der Prozedur 'sort'.

```
simple_sort MODULE                !Modulvereinbarung!

CONSTANT                          !Konstantenvereinbarungen!
  false:=0
  true :=1

EXTERNAL
  list ARRAY [10 WORD]            !zu sortierendes Feld!

INTERNAL

  sort PROCEDURE                  !Prozedurvereinbarung!
    LOCAL                         !lokale Variablenvereinb.!
      switch BYTE                 !Austauschanzeiger!
    ENTRY                         !Beginn des ausfuehrbaren!
    DO                            !Teils!
      LDB switch.#false           !Initialisierung von!
                                  !switch!
      CLR R1                      !Loeschen Feldzeiger i!
      DO
        CP R1,R0                  !Feldende erreicht?!
        IF UGE THEN EXIT FI
        LD R2,R1                  !Initialisierung Zeiger j!
        INC R2,#2                 !j=j+1 (doppelt da Worte)!
        LD R4,list(R1)
        LD R6,list(R6)
        CP R4,R6                  !wenn list[i]>list[j] dann!
        IF UGT THEN               !Austausch!
               LDB switch, #true
               LD list(R1),R6
               LD list(R2),R4
        FI
        INC R1,#2                 !naechste Element (Wort)!
      OD                          !Ende innere DO-Schleife!
      CPB switch,#false           !war Austausch?!
      IF EQ THEN RET FI
    OD                            !Ende aeussere DO-Schleife!
  END sort                        !Ende der Prozedur sort!

GLOBAL                            !neue Prozedurvereinbarung!

  main PROCEDURE                  !Hauptprogramm!
    ENTRY                         !keine lokalen Groessen!
    LD R0,#9*2                    !Feldlaenge!
    CALL sort                     !Aufruf der Prozedur sort!
    RET
  END main                        !Hauptprogrammende!

END simple_sort                   !Modulende!
```

2.4.2. U881/U882

Die Laenge des zu sortierenden Feldes wird der Prozedur 'sort' ueber die externe Groesse limit mitgeteilt. Der Austauschanzeiger ist eine lokale Groesse (ein Byte) innerhalb der Prozedur 'sort'.

```
simple_sort MODULE                !Modulvereinbarung!

CONSTANT                          !Konstantenvereinbarungen!
  false:=0
  true :=1

EXTERNAL
  list ARRAY [10 WORD]            !zu sortierendes Feld!
  limit BYTE

INTERNAL

  sort PROCEDURE                  !Prozedurvereinbarung!
    LOCAL                         !lokale Variablenverein-!
                                  !barungen!
      i,j BYTE                    !Elementenzeiger!
      switch BYTE                 !Austauschanzeiger!
    ENTRY                         !Beginn des ausfuehrbaren!
                                  !Teils!
      SRP #%10
      DO
        LD switch.#false          !Initialisierung Aus-!
                                  !tauschanz.!
        CLR i                     !Loeschen Feldzeiger i!
        DO
          CP i,limit              !Feldende erreicht?!
          IF UGE THEN EXIT FI
          LD j,i                  !Initialisierung Zeiger j!
          INC j
          LD R2,i
          ADD R2,R2
          LD R4,list(R2)
          LD R5,list+1(R2)        !R4/R5 = list[i]
          LD R3,j
          ADD R3,R3
          LD R6,list(R3)
          LD R7,list+1(R3)        !R6/R7 = list[i+1]
          CP R4,R6                !Vergleich byteweise!
          IF EQ THEN CP R5,R7 FI
          IF UGT THEN             !wenn list[i]>list[j] dann!
               LD switch, #true   !Austausch!
               LD list(R2),R6
               LD list+1(R2),R7
               LD list(R3),R4
               LD list+1(R3), R5
        FI
        INC i                   !naechste Feldelement!
      OD                        !Ende innere DO-Schleife!
      CP switch,#false          !war Austausch?!
      IF EQ THEN RET FI
    OD                          !Ende aeussere DO-
```

```
                                !Schleife !
  END sort                      !Ende der Prozedur sort!

GLOBAL                          !neue Prozedurvereinbarung!

  main PROCEDURE                !Hauptprogramm!
    ENTRY                       !keine lokalen Groessen!
    LD limit,#9                 !Feldlaenge!
    CALL sort                   !Aufruf der Prozedur sort!
    RET
  END main                      !Hauptprogrammende!

END simple_sort                 !Modulende!
```

3. Benutzung des U881/U882- und des U8000-Assemblers

Fuer den Einchipmikrorechner U881/U882 und fuer die Prozessoren U8001/U8002 erfolgt eine analoge Programmentwicklung.
Der Aufbau von PLZ/ASM-Programmen und die Syntax wurde in den vorhergehenden Abschnitten erlaeutert. die Vefehlsmnemoniken und Operandennotationen sind den entsprechenden Befehlslisten zu entnehmen.
Die PLZ/ASM-Assembler fuer die Einchipmikrorechner U881/U882 und fuer die 16-Bit-Mikroprozessoren U8001/U8002 uebersetzen einen mit dem Editor erzeugtes Quellfile in einen Objektmodul.
Beim Erstellen der Quellprogramme sind die jeweiligen vom Uebersetzungsprogramm reservierten Worte (Operatornamen, Flagbezeichnungen, Assemblerbefehle, Bedingungskodes, Registerbezeichnungen, Schluesselwoerter der hoeheren Sprachelemente u.a.m.) und die Zeichen mit besonderer Bedeutung (z.B.: zur Kennzeichnung der Adressierungsart) zu beachten.
Der Objektmodul kann absolut oder auch relativ ladbar erzeugt werden. Beim Arbeiten mit verschieblichen Modulen ist eine Neufestlegung der Adressen ohne nochmalige Uebersetzung moeglich. Weiterhin koennen die Programme in unbahaengige Module aufgegliedert werden, die getrennt voneinander programmiert und uebersetzt werden koennen. Dies verbessert erheblich die Strukturierung, die Dokumentation und die Wartung der Programme.
In einem anschliessenden Bindevorgang (ZLINK) kann der Objektmodul kombiniert mit anderen seperat erzeugten anderen Objektmodulen zu einem Lademodul (wieder ein Objektmodul) gebunden werden. Dadurch ist ein sogenanntes "inkrementales" Verbinden moeglich, d.h. eine Gruppe von Objektmodulen wird zu einem Modul verbunden, der wiederum als ein Objektmodul fuer weitere Linkanweisungen zur Verfuegung steht.
Das Verbinderprogramm (ZLINK) bietet weiterhin die Moeglichkeit der Manipulation von Programm- und Datenbereichen. Durch die Steueranweisung $SECTION kann der Anwender Programmbereichen Namen geben, auf die beim Linkeraufruf Bezug genommen werden kann. dies Konzept ist fuer die Strukturierung von komplexen Programmen und zur Speicheraufteulung der Anwenderprogramme (z.B. Trennung in PROM- und RAM-Bereiche) nuetzlich.
Mit Hilfe des Imagerprogramms (IMAGER) wird aus einem von dem Assembler oder Linker erzeugten Objektmodul ein Maschinenprogramm (ausfuehrbares Programm) mit absoluten Adressen.

3.1. Aufruf

Die Assembler werden durch folgende Kommandos aufgerufen:

```
U8ASM filename [optionen]

U8000ASM filename [optionen]
```

Der 'filename' muss die Endung ".S" haben (sie braucht beim Aufruf nicht mit angegeben zu werden) und enthaelt die Quelle fuer einen einzigen PLZ/ASM-Modul. Die Assemblerprogramme erzeugen implizit eine Objektkodedatei (Endung ".OBJ") und eine Listingdatei (Endung ".L").
Folgende Optionen koennen nach dem Quellprogrammnamen in beliebiger Reihenfolge angegeben werden:

D=string	'string' ist eine Zeichenkette und kann aus max. 18 Zeichen bestehen. Sie wird in der Kopfzeile des Listings angegeben.
I=filename	Der bei der Uebersetzung erzeugten Zwischendatei wird der Name 'filename' zugeordnet, sie wird nicht geloescht. Ist I nicht spezifiziert, wird die Zwischendatei geloescht.
L=filename	Anstelle des standardmaessig aus Quellfilenamen mit Erweiterung ".L" gebildeten Namen des Listingfiles bekommt das Listingfile den Namen 'filename'.
NOL	Es wird kein Listingfile erstellt.
O=filename	Anstelle des standardmaessig aus Quellfilenamen mit Erweiterung ".OBJ" gebildeten Namen des Objektfiles bekommt das Objektfile den Namen 'filename'. Falls keine Objektdatei gewuenscht wird, so ist O=$NULL anzugeben.
P	Eine Kopie des Listingfiles wird auf dem Drucker ausgegeben. Auf dem Bildschirm erscheinen keine Compilerfehler. Standardmaessig werden Fehlernummer und fehlerhafte Zeile auf dem Bildschirm ausgegeben.

Die Assembler benutzen die folgenden logischen Einheiten:

2	(CONOUT)	Meldungen zur Konsole
3	(SYSLST)	Listingkopie, wenn P angegeben
4		Quellfile
5		Listingfile
6		Objektfile
7		Zwischenfile

3.2. Listing-Format

Heading — Der Kopf der ersten Seite enthaelt die Assembler-Versionsnummer und bei Angabe der Option D die Zeichekette 'string'

LOC — Diese Spalte enthaelt die Referenzzaehler (relative Speicherplatzadresse). Zusaetzlich bei U8ASM eine Speicherbezeichnung (P=Programm, R=Register, D=Daten).

OBJ CODE — Diese Spalte enthaelt den erzeugten Objektkode. Wenn eine Anweisung keinen Kode erzeugte, ist sie leer (z.B. ENTRY).
Jedem Byte oder Word des Objektcodes folgt ein Leerzeichen (blank), ein Apostroph (') oder ein Stern (*).
Ein Leerzeichen bedeutet, dass der Kode unveraendert bleibt.
Apostroph kennzeichnet einen relativen Wert.
Ein Stern bedeutet, das der Wert von einem EXTERNAL-Symbol abhaengt.
Drei Punkte "..." werden benutzt, um anzudeuten, dass das vorhergehende Byte oder Wort wiederholt wird.

STMT — Statement Number: Ist die Nummer der Quellzeile.

SOURCE STATEMENT — Enthaelt die Quelltextzeile.

3.3. Assemblerdirektiven und Pseudobefehle

Die Direktiven sind Assemblersteueranweisungen und werden benutzt, um die Arbeitsweise des Assemblers zu steuern. Diese Direktiven sind Bestandteil des Quellprogramms. Sie muessen allein auf einer Zeile stehen. Eine Assemblersteueranweisung muss als erstes Zeichen ein "$" haben und in Spalte 1 beginnen, unmittelbar gefolgt von der speziellen Direktive. Danach koennen noch Operanden folgen. Die Direktiven erscheinen explizit im Listing.
Zusaetzlich koennen noch erweiterte Befehle innerhalb einer Prozedur zur Erzeugung von Byte-, Wort- oder Longwortgroessen benutzt werden. Diese werden in die Objektdatei eingefuegt.

$ABS [adresse] — Kennzeichnet den erzeugten Objektkode als absolut adressiert. Falls eine Adresse spezifiziert ist, so wird der Befehlskodezaehler auf diesen Wert gesetzt (sonst 0). Die Anweisung ist bis zum Auftreten einer $REL-Anweisung gueltig.

$REL [adresse] Kennzeichnet den erzeugten Objektkode als verschieblich. Falls eine Adresse angegeben ist, so kennzeichnet sie die relative Distanz vom Anfang des aktuellen Bereichs (SECTION). Die Wirkung kann durch eine $ABS-Anweisung aufgehoben werden.
(Implizit ist $REL 0 gesetzt).

$DEFAULT Beendet die Wirkung einer vorherigen $SECTION-Anweisung und stellt die Standard-Speicherzuweisung her. Muss nur angegeben werden, wenn nach $SECTION die Standardzuweisung wiederhergestellt werden soll.
<u>U8000-Assembler:</u>
Modulname_D Datenvereinbarungen
Modulname_P Prozedurvereinbarungen
<u>U881/U882-Assembler:</u>
Modulname_R REGISTER-Sektion; Datenvereinbarungen
Register koennen nicht initialisiert (mit Anfangswerten versehen) werden. Das bedeutet, dass bei der Standardzuordnung in Variablenvereinbarungen, keine Anfangswerte auftreten koennen. Ausfuehrbare Anweisungen koennen ebenfalls nicht in der Sektion REGISTER erscheinen.
Modulname_P PROGRAMM-Sektion, Prozedurvereinbarungen.

$SECTION Ein Programm kann in Abschnitte (Sektionen) eingeteilt werden, die jeweils beim Binden oder Laden in den Speicher in verschiedene Speicherbereiche gebracht werden. Prozeduren und Daten koennen in den gleichen Speicherbereich gebracht werden. Es kann aber auch notwendig sein, Prozeduren und Daten in getrennte Speicherbereiche zu bringen (EPROM und RAM).
Der U881/U882-Assembler bildet standardmaessig alle Variablenvereinbarungen (Modulname_R) in die REGISTER-Sektion und alle Prozedurvereinbarungen (Modulname_P) in die PROGRAM-Sektion ab.
Der U8000-Assembler separiert implizit ein Programm in die Bereiche "Modulname_D" fuer den Datenteil und "Modulname_P" fuer den Programmteil.
Mit der Direktive $SECTION kann dieser

Standard fuer die Assembler geaendert werden.

name <u>U8000-Assembler:</u>
Verbindet den nachfolgend erzeugten Objektkode mit dem symbolischen Namen zur spaeteren Zuordnung in bestimmte Speicherbereiche.

[name] Bereich <u>U881/U882-Assembler:</u>
Bei der Separierung in Bereiche erfolgt eine Aufteilung in drei Bereiche "PROGRAM", "REGISTER" und "DATA". Der Bereich muss bei der $SECTION-Anweisung mit angegeben werden. Die Zuweisung zur DATA-Sektion kann nur mit $SECTION vorgenommen werden. Durch die Verwendung von "name" kann der Anwender zusaetzliche Sektionen erzeugen. Gleiche "namen" muessen immer mit dem gleichen "Bereich" gekoppelt sein, sonst treten Fehler auf. Durch den hardwaremaessigen Aufbau des U881/U882 in Bezug auf Programm- und Datenspeicher ist es nur gestattet, Datenbereiche mit anderen Datenbereichen, Registerbereiche mit anderen Registerbereichen und Programmbereiche mit anderen Programmbereichen zu kombinieren.

$PAGE Bewirkt, dass im Listing eine neue Seite begonnen wird. Diese Direktive erscheint selbst nicht im Listing.

$LISTON [option]
$LISTOFF [option]
Diese Direktive steuert das Listingformat.
Ohne "option" schaltet sie das Listing aus ($LISTOFF) oder ein ($LISTON). Standard ist $LISTON. Fuer "option" gibt es:

$TTY: erzeugt ein 80-spaltiges Listingformat. Standard ist $LISTOFF $TTY, d.h. volle Zeichenlaenge.

$BEX: steuert das Format ded Objektkodes. $LISTOFF $BEX listet nur eine Objektkodezeile fuer eine Eingabezeile. Standars ist $LISTON $BEX, d.h. der gesammte Objektkode wird gelistet.

BVAL ausdruck Durch die Pseudoanweisung wird ein Bytewert definiert.

WVAL ausdruck Es wird ein Wort definiert.

LVAL ausdruck Es wird ein Longwort definiert (nur U8000-Assembler).

Bedingte Assemblierung

Fuer die bedingte Assemblierung wird die IF-Anweisung in abgewandelter Form benutzt.

```
$IF  bc   $THEN

     Assembleranweisungen

ELSE                          !optional!

     Assembleranweisungen

$FI
```

Ist bc ungleich 0 (Konstante) oder ein definierter Name, dann werden die Assembleranweisungen zwischen $THEN und $ELSE uebersetzt. Ist das nicht der Fall (Konstante=0 bzw. undefinierter Name) so werden nur die Assembleranweisungen zwischen $ELSE und $FI uebersetzt. Schatelungen sind erlaubt.

3.4. Implementationsmerkmale und -einschraenkungen

- Die PLZ/ASM-Assembler benutzen die ASCII-Zeichenmenge. Gross- und Kleinbuchstaben werden unterschieden. Schluesselwoerter muessen vollstaendig aus Klein- oder Grossbuchstaben bestehen (GLOBAL oder global, aber nicht Global). Hexadezimalzahlen und spezielle Zeichenketten koennen gross oder klein geschrieben werden.

- Quellzeilen mit mehr als 132 Zeichen werden angenommen, bei Fehlermeldungen werden jedoch nur 132 Zeichen gedruckt. Kommentare und Zeichenketten koennen ueber mehrere Zeichen verteilt sein.

- Zeichenketten der Laenge Null sind nicht zugelassen.

- Konstanten werden intern als vorzeichenlose 32-Bit-Werte dargestellt. 4/-2 ist 0, da -2 als sehr grosse, vorzeichenlose Zahl dargestellt wird. Es gibt keinen Ueberlauftest bei der Berechnung konstanter Ausdruekke. Wird eine Zeichenfolge als Konstante definiert, werden die ersten 4 Zeichen benutzt (z.B. 'ABCDE'='ABCD'). Wird eine Zeichenkette als Array-Initialisierung verwendet, kann sie bis zu 127 Zeichen lang sein.

- Namen koennen aus max. 127 Zeichen bestehen.

- Die Schachtelungstiefe von Konstruktionen wird nur von der verfuegbaren Stackgroesse begrenzt. Der STACK-SIZE-Wert kann mit dem SET-Kommando von UDOS modifiziert werden.

- Nach einem Fehler in CONSTANT, TYPE oder Variablen-Vereinbarungen springen die Assembler zum naechsten Schluesselwort, mit dem eine neue Anweisung beginnt (z.B. ein Operationskode, IF, DO, EXIT, REPEAT, END). Dadurch werden eventuell nicht alle Fehler bei einer Assemblierung bemerkt.

- Die Initialisierungssymbole "?", "..." und geschachtelte eckige Klammern ('[' und ']') sind nicht implementiert.

3.5. Fehlermeldungen

```
     <u>Warnings</u>
/  0     A '-' or '+' treated as binary operator
/  1     Missing delimiter between tokens
/  2     Array of zero elements
/  3     No fields in record declaration
/  4     Mismatched procedure name
/  5     Mismatched module name
/  6     Constant out of range for type
/  7     Conversion of unaligned address to a pointer to
         aligned type
/  8     Absolute adress warning for U8000

     <u>Token Errors</u>
/ 10     Decimal number too large
/ 11     Invalid operator
/ 12     Invalid special character after '%'
/ 13     Invalid hexadecimal digit
/ 14     Character sequence of zero length
/ 15     Invalid character
/ 16     Hexadecimal number too large

     <u>Do Loop Errors</u>
/ 20     Unmatched 'OD'
/ 21     'OD' expected
/ 22     Invalid repeat statement
/ 23     Invalid exit statement
/ 24     Invalid 'FROM' label

     <u>IF Statements Errors</u>
/ 30     Unmatched 'FI'
/ 31     'FI' expected
/ 32     'THEN' or 'CASE' expected
/ 33     Invalid selector register

     <u>Symbol Expected</u>
/ 40     ')' expected
/ 41     '(' expected
/ 42     ']' expected
/ 43     '[' expected
/ 44     ':=' expected
/ 45     '^' expected
```

```
     <u>Undefined Names</u>
/ 50     Undefined identifier
/ 51     Undefined procedure name

     <u>Declaration Errors</u>
/ 60     Type identifier expected
/ 61     Invalid module declaration
/ 62     Invalid declaration class
/ 63     Invalid use of array [*] delcaration
/ 64     Uninitialized array [*] declaration
/ 65     Invalid dimension size
/ 66     Invalid array component type
/ 67     Invalid record field declaration
/ 68     Invalid type used in pointer declaration

     <u>Procedure Declaration Errors</u>
/ 70     Invalid procedure declaration
/ 71     'ENTRY' expected
/ 72     Procedure name expected after 'END'
/ 73     Formal parameter name expected
/ 74     Invalid formal parameter type

     <u>Initialization Errors</u>
/ 80     Invalid initial value
/ 81     Too many initialization elements for declared
         variables
/ 82     Invalid initialization
/ 83     Array [*] gives single non character sequence
         initializer
/ 84     Attempted typ initializes an uninitialized data
         area

     <u>Special Errors</u>
/ 90     Invalid statement
/ 91     Invalid instruction
/ 92     Invalid operand
/ 93     Operand to large
/ 94     Relative address out of range
/ 95     ":" expected
/ 97     Duplicate record file name
/ 98     Duplicate 'CASE' constant
/ 99     Multiple declaration of identifier

     <u>Invalid Variables</u>
/100     Invalid variable
/101     Invalid operands for '#' or 'SIZEOF'
/102     Invalid field name
/103     Subscription of non-array variable
/104     Invalid use of '.'
/105     Invalid use of '^'

     <u>Expression Errors</u>
/110     Invalid arithmetic expression
/111     Invalid conditional expression
/112     Invalid constant expression
/113     Invalid select expression
/114     Invalid index expression
/115     Invalid expression in assignment
```

```
     <u>Constant Out of Bounds</u>
/120     Constant too large for 8 bits
/121     Constant too large for 16 bits
/122     Constant array index out of bounds

     <u>Call Errors</u>
/130     Invalid arithmetic expression
/131     Invalid procedure call
/132     Procedure  call with multiple out parameters  ex-
         pexted
/133     Too few out parameters
/134     Too many out parameters
/135     Too few in parameters
/136     To many in parameters

     <u>Type Incompatibility</u>
/140     Char_sequence  used to initialize array with  base
         type_length not 8 bits
/141     Type incompatibility with initialization

     <u>Type Errors</u>
/150     Type incompatibility in arithmetic expression
/151     Invalid operand type for unary operator
/152     Invalid operand type for binary operator
/153     Unassignable type
/154     Invalid index type
/156     Parameter type incompatible
/157     Invalid actual paramter
/158     Return parameter type incompatible
/159     Return value must be adress
/160     Type incompatibility in assignment
/161     Invalid operand type for relational operator
/162     Type incompatiblilty in conditional expression
/163     Invalid type conversion
/164     Invalid relational operator for structures

     <u>Segment Errors</u>
/170     Invalid operator in nonsegmented mode
/171     Mismatched short address operator
/172     Mismatched segment designator

     <u>Assembly Directives Errors</u>
/180     Inconsistent area specifier
/181     Invalid area specifier
/182     Mismatched conditional assembly directives
/183     Invalid conditional assembly directives
/184     Attempted to mix segmented and nonsegmented code
/185     Directive must appear alone on a single line

     <u>File Errors</u>
/198     EOF expected
/199     Unexpected   EOF encountered in  source.  Possibly
         missing ' or !

     <u>Implementation Restrictions</u>
/224     Too many symbols -- hash table overflow
/226     Short segmented offset out of range
/227     Object symbol table overflow
```

/228 Relocation out of range (word overflow)
/229 Unimplemented feature
/230 Character sequence or identifier too long
/231 Symbol table overflow
/232 Procedure too long
/233 Left hand side of assignment too complicated
/234 Too many initialization values
/235 Stack overflow
/236 Operand too complicated
/237 Static data overflow
/239 Too many internal or global procedures
/240 Long constants not implemented

4. Literaturempfehlungen

- 16 Bit Mikroprozessoren U8001/U8002. Kundeninformation. Erfurt: VEB Mikroelektronik "Karl Marx" 1986

- Speicherverwaltungsbaustein MMU U80100. Kundeninformation. Erfurt: VEB Mikroelektronik "Karl Marx" 1986

- Einchip-Mikrorechner Uebersicht UX881D und UX883D. Kundeninformation. Erfurt: VEB Mikroelektronik "Karl Marx" 1984

- K.-D.; Roth, M.: PAS 681 Beschreibung Einchip-Mikroprozessor. Ilmenau: technische Hochschule 1983

- Handbuch UDOS-1526. Teil:Sprachbeschreibung/Manual Cross-Software U8001/U8002 ASM. VEB Robotron-Buchungsmaschinenwerk Karl-Marx-Stadt 1983

- Handbuch UDOS-1526. Teil: Sprachbeschreibung/Manual Cross-Software U881/U882 ASM. VEB Robotron-Buchungsmaschinenwerk Karl-Marx-Stadt 1983

Z L I N K

Benutzerhandbuch

Inhaltsverzeichnis Seite

1. Einfuehrung

Das Programm ZLINK verbindet durch Assemblierung separat erzeugte Objektmodule zu einem sogenannten Lademodul, der die gleiche Struktur hat wie die zu verbindenden Objektmodule. Dadurch ist ein inkrementales Verbinden moeglich, d.h., eine Gruppe von Objektmodulen wird zu einem Lademodul verbunden, der dann aufgrund seiner Objektmodulstruktur mit weiteren Objektmodulen verbunden werden kann usw. Externe Referenzen werden in jedem Fall, also auch beim inkrementelen Verbinden, aufgeloest. Optional wird eine Link Map erstellt, die Auskunft ueber die Anordnung der Module, Sektionen und ueber die Lokalisierung der globalen Variablen gibt. Ferner werden Fehler des Linkprozesses ueber Bildschirm bzw. Link Map angezeigt. Eine weitere wichtige Eigenschaft von ZLINK ist die Faehigkeit, die durch den Assembler realisierbare Sektionsstruktur beizubehalten bzw. bei Bedarf durch Gruppenbildung von Sektionen neue, mit einem Namen versehene Sektionen zu schaffen. Wie bereits erwaehnt, hat der durch den LINKER erzeugte Lademodul eine Objektmodulstruktur, d.h., er ist noch kein ladefaehiger Maschinenkode mit absoluten Adressen. Die Verarbeitung zu solch einem ladefaehigen Maschinenprogramm erfolgt durch das Programm IMAGER.

2. Arbeitsweise von ZLINK

der Linkprozess erfolgt in zwei Durchlaeufen.

Erster Durchlauf:

Aufteilung der Module in ihre Sektionen. Sektionen mit gleichen Namen werden zusammengefasst, auch dann, wenn sie in verschiedenen Modulen enthalten sind. Das Ergebnis ist eine Zwischendatei.

Zweiter Durchlauf:

Organisation der durch den Anwender festgelegten Sektionen und Aufloesung der externen Referenzen. Nichtdefinierte externe Referenzen werden gemeldet, bleiben aber im Lademodul. Der so teilweise gebundene Lademodul kann dann fuer einen weiteren Linkprozess verwendet werden (inkrementales Verbinden)

Erfolgt die Abarbeitung der Durchlaeufe fehlerfrei, dann ist das Ergebnis der Lademodul.

3. Aufbau des Kommandos ZLINK

Der Linker kann nach Erscheinen des Promptzeichens (%) des Betriebssystems UDOS durch das folgende Kommando gestartet werden:

ZLINK Kommandolinie

- ZLINK ist der Programmname des Linkers
- Die Kommandolinie enthaelt Angaben ueber die zu verbindenden assemblierten Module und eine Reihe von Optionen.

3.1. Kommandolinie

3.2. Modulangaben

Von den zu verbindenden assemblierten Modulen sind die Namen der entsprechenden Objektdateien anzugeben. Es reicht aus, wenn nur der Name ohne die Erweiterung .OBJ notiert wird. Die Objektdatei selbst muss aber diese Erweiterung haben.

Beispiel:

Es sind die Objektdateien STEUERPROG.OBJ, SUBPROG1.OBJ und SUBPROG2.OBJ zu verbinden.

```
ZLINK STEUERPROG SUBPROG1.OBJ SUBPROG2.OBJ
```

Hinsichtlich der in den Modulen vorhandenen Sektionen bzw. der nach dem Assemblieren entstandenen Sektionen ist folgendermassen zu verfahren:

Sektionen koennen durch Klammern logisch zu Gruppen zusammengefasst und mit einem Namen versehen werden. Es entstehen also neue Sektionen.

Beispiel:

```
ALLE_DATEN=(EING_DATEN AUSG_DATEN)
```

Wird fuer eine solche Gruppe kein Name notiert, dann wird standardmaessig der erste Name der Gruppe als Gruppenname angenommen. Zur Verkuerzung der Schreibweise kann wie ueblich ein Stern (*) verwendet werden.

Beispiel:

```
ALLE_DATEN=(*_DATEN)
```

Es koennen mehrere Gruppen in der Kommandolinie auftreten. Sektionen in verschiedenen Modulen mit identischen Namen werden als eine Sektion betrachtet. Jede Sektion, die nicht in irgendeiner Form in der Kommandozeile notiert wird, erscheint nach dem Linkprozess als selbstaendige Sektion.

Beispiel:

Modulname	Objektdateiname	Sektionen im Modul
STEUERPROG	STEUERPROG1.OBJ	STEUERPROG_D TEXT INTERFACE
SUBPROG1	SUBPROG1.OBJ	SUBPROG1_D SUBPROG1_P TEXT
SUBPROG2	SUBPROG2.OBJ	SORT MERGE SORT_D MERGE_D SUBPROG2_P

Das Kommando lautet:

```
ZLINK STEUERPROG SUBPROG1 SUBPROG2
      DATA=(*_D) (TEXT) (MERGE INTERFACE SORT)
```

Nach dem Linken entsteht folgende Sektionsstruktur:

Namen der Sektionen in den Modulen	Sektionen nach dem Linkprozess
STEUERPROG_D SUBPROG1_D SORT_D MERGE_D	Sektion DATA
TEXT	Sektion TEXT
MERGE INTERFACE SORT	Sektion MERGE
SUBPROG1_P	Sektion SUBPROG1_P
SUBPROG2_P	Sektion SUBPROG2_P

Die im Kommando bei der Gruppe DATA angewendete verkuerzte Schreibweise kann auch dann angewendet werden, wenn eine oder auch mehrere Sektionen nicht in dieser Gruppe, sondern in einer weiteren oder anderen Gruppe erscheinen sollen. Diese Sektion(en) ist(sind) dann in der entsprechenden Gruppe zu notieren.

3.3. Optionen

3.3.1. Link Map

L[INKMAP] '=' dateiname | '*'

- bei L=dateiname wird ein Link Map mit dem Namen dateiname erzeugt.
- bei L=* wird ein Link Map mit einem Namen, der aus dem ersten Objektdateinamen der Kommandolinie mit der Erweiterung .MAP gebildet wird, erzeugt.

Beispiel:

```
L=mylink.map
```

3.3.2. Name des Lademoduls

N[AME] '=' dateiname | '*'

- bei N=dateiname ist der Name des Lademoduls der dateiname.
- bei N=* ist der Name des Lademoduls der Name der ersten Objektdatei der Kommandoolinie ohne die Erweiterung .OBJ.

Beispiel:

```
N=MYLOADMODULE
```

3.3.3. Name der Zwischendatei

T[EMPORARYFILE] '=' dateiname | '*'

- bei T=dateiname ist 'dateiname' der Name der Zwischendatei.
- bei T=* ist der Name der Zwischendatei der Name der ersten Objektdatei der Kommandolinie mit der Erweiterung .T
- Fehlt die Option T in der Kommandolinie, dann wird die Zwischendatei wieder geloescht.

Beispiel:

```
T=MYTEMPFILE
```

3.3.4. Eintrittspunkt

E[NTRY] '=' name | '*'

- bei E=name ist name derjenige Prozedurname oder die Marke, bei der die Programmausfuehrung beginnt.
- Fehlt die Option E in der Kommandolinie, ist das erste ENTRY-Symbol der Eintrittspunkt.
- E=* ist eine Nulloperation

Beispiel:

```
E=myentrypt
```

3.3.5. Warnung

W[ARNING] '=' ON | OFF

- bei W=ON werden Fehler der Klasse III angezeigt.
- bei W=OFF werden Warnungen weder auf dem Bildschirm noch im Link Map angezeigt.
- Fehlen der Option entspricht W=OFF.

Beispiel:

```
W=ON
```

3.3.6. KEEP

K[EEP] '=' Namensspezifikation

```
Namensspezifikation = {namensliste}
                    = ALL | NONE
                    = ALLBUT {namensliste}
```

Die Moeglichkeit des inkrementalen Verbindens kann zur Folge haben, dass externe Referenzen (globale Symbole) erst spaeter verfuegbar sind. Solche Referenzen muessen deshalb in die Symbolliste des Lademoduls aufgenommen werden.

- bei K={namensliste} sind die entsprechenden aufgefuehrten globalen Symbole im Lademodul enthalten.
- bei K=ALL sind alle globalen Symbole im Lademodul enthalten.
- bei K=NONE sind keine globalen Symbole im Lademodul enthalten
- bei K=ALLBUT {namensliste} sind alle globalen Symbole im Lademodul enthalten, die nicht in der

Namensliste aufgefuehrt sind.

- Fehlen der Optionen entspricht K=NONE

Beispiel:

K={entry1,entry2,entry3}

KEEP=ALL

KEEP=ALLBUT{entry3,parameter}

3.4. Beispiel

Es sind die Objektmodule STEUERPROG.OBJ, SUBPROG1.OBJ, SUBPROG2.OBJ zu verbinden.

Durch Optionen ist folgendes zu realisieren:

- Link Map : Abspeichern auf Diskette unter dem Namen STEUERPROG.MAP
- Lademodul : Abspeichern auf Diskette unter dem Namen STEUERPROG
- Eintritts-punkt: Die Programmausfuehrung beginnt bei der Marke START
- Warnungen : Meldung von Fehlern der Klasse III

Das Kommando lautet:

```
ZLINK STEUERPROG SUBPROG1 SUBPROG2
      L=* N=* W=ON E=START
```

4. ZLINK-Fehlermeldungen

4.1. Fehler der Klasse I

Bei Auftreten derartiger Fehler erfolgt der unmittelbare Abbruch des Linkprozesses.

ILLEGAL CHAR IN COMMAN LINE: <c>

Die Kommandozeile enthaelt das vom Linker nicht deutbare Zeichen 'c'.

ERROR IN GROUPING FORMAT: <symbol>

Die Kommandozeile enthaelt einen durch 'symbol' hervorgerufenen Fehler bei der Gruppenbildung.

OPTION NOT RECOGNIZED IN COMMAND LINE: <c>

Nichtzulaessige Option. Der erste Buchstabe der Option ist 'c'.

NAME EXCEEDS MAXIMUM: <name>

Dateiname ist zu lang.

UNABLE TO OPEN TEMPORARY FILE: <name>

Die Zwischendatei 'name' kann nicht eroeffnet werden.

UNABLE TO OPEN OBJECT FILE: <name>

Wie oben fuer Objektdatei.

UNABLE TO OPEN INPUT FILE: <name>

Wie oben fuer Eingabedatei.

UNABLE TO OPEN LINK MAPFILE: <name>

Wie oben fuer Linkdatei.

WRITE FILE ERROR, ERR CODE: <hexzahl>

Systemfehler beim Schreiben einer Datei. Anzeige des Fehlerkodes.

READ FILE ERROR, ERR CODE: <hexzahl>

Systemfehler beim Lesen einer Datei. Anzeige des Fehlerkodes.

SEEK ERROR, ERR CODE: <hexzahl>

Auftreten eines Fehlers beim Suchen einer bestimmten Position in einer Datei. Systemfehlerkode wird angezeigt.

UNABLE TO CLOSE FILE: <name>, ERR CODE: <hexzahl>

Fehler beim Schliessen der Datei. Anzeige des Systemfehlerkodes.

SYMBOL TABLE OVERFLOW

Symboltabelle ist voll. Abhaengig von Anzahl und Laenge der globalen Symbole in den zu bindenden Modulen.

GENERAL TABLE OVERFLOW

Der fuer allgemeine Informationen ueber Sektionen, Filenamen usw. vorgesehene Platz ist voll. Abhaengig von der Anzahl der Sektionen in den Modulen.

BAD CALL TO ERROR ROUTINE: <err>

Die interne Linkerfehlerroutine kann einen Fehler nicht deuten.

FEATURE NOT IMPLEMENTED: <c>

Die Kommandozeile enthaelt eine noch nicht implementierte Option. Das erste Zeichen der Option wird angezeigt.

4.2. Fehler der Klasse II

Bei Auftreten derartiger Fehler erfolgt der Abbruch nach dem ersten Durchlauf.

MULTIPLE DECLARATION OF GLOBALE: <name> <module1> <module2>

Ein in 'module1' definierter globaler Name ist schon in 'module2' definiert.

SEG AND NONSEG MODULES MIXED

Segmentierte und nichtsegmentierte Module treffen beim Verbinden zusammen.

OBJECT CODE CONTAINS ERROR: <name> <fehler>

Der Objektkode in der Datei 'name' ist fehlerhaft. 'fehler' ist der Fehlerkode.

BYTE OVERFLOW

Ein relativer Bytewert ist zu gross fuer 8 Bits.

SECTION TOO LARGE: <name>

Die Groesse der Sektion 'name' uebersteigt 65 536 Bytes

INPUT POSSIBLY U880 CODE

Eingabedatei nicht deutbar, eventuell U880-Objektmodule.

SEG TAG IN NONSEG MODULE

Eine auf segmentierten Modul hinweisende Groesse ist in einem nichtsegmentierten Modul gefunden worden.

NONSEG TAG IN SEG MODULE

umgekehrt wie oben

4.3. Fehler der Klasse III

Bei Auftreten derartiger Fehler erfolgt nur eine Warnmeldung.

GLOBAL TYPE MISMATCH: <name> <module1> <module2>

Typunterschied von 'name' in 'module1' und 'module2'.

UNDEFINED GLOBAL: <name> <module>

EXTERNAL name in 'module' ist nicht definiert.

UNDEFINED GLOBAL REFERENCE: <name>: <module>

In 'module' ist ein Bezug zum nichtdefinierten globalen Symbol 'name' vorhanden.

NO ENTRY POINT FOUND

Kein Eintrittspunkt im Modul gefunden.

UNMATCHED SECTION ATTRIBUTES

Der Linker soll Sektionen verschiedenen Typs kombinieren; z.B. REGISTER mit PROGRAM.

I M A G E R

Benutzerhandbuch

INHALTSVERZEICHNIS Seite

1. Einfuehrung

Das Programm IMAGER verarbeitet die durch Assemblierung, Compilierung oder Linkprozess erzeugten Objektmodule zu einem einzigen ladefaehigen Maschinenkodeprogramm mit absoluten Adressen. Dieses Maschinenkodeprogramm ist nach dem Ende des Imagerlaufes ab der Adresse 4400H im Speicher des Betriebssystems gespeichert. Es darf eine Laenge von 4000H Bytes nicht uebersteigen. Die nicht durch den IMAGER beschriebenen Speicherplaetze enthalten den Wert FFH. Das Maschinenkodeporgramm kann dann als Datei vom Typ 'P' mit Hilfe einer Option auf Diskette abgespeichert werden. Zu beachten ist, dass die durch den IMAGER zu bearbeitenden Objektmodule keine nichtaufgeloesten externen Referenzen haben duerfen. Der IMAGER ist kein LINKER!

2. Aufbau des Kommandos IMAGER

Der IMAGER kann nach Erscheinen des Promptzeichens (%) des Betriebssystems UDOS durch das folgende Kommando gestartet werden:

```
IMAGER Kommandolinie
```

- IMAGER ist der Programmname
- Die Kommandolinie enthaelt Angaben ueber die zu verarbeitenden Objektmodule, Sektionen, den zu ladenden Speicherbereich und eine Reihe von Optionen.

  Die Syntax der Kommandolinie ist folgende:

```
Datei (Sektionsliste) {Fenster} Optionen
```

2.1. Dateiliste

Von den durch den IMAGER zu verarbeitenden Objektmodulen sind die Namen der entsprechenden Dateien vollstaendig zu notieren; also so, wie sie auf der Diskette bezeichnet sind.

2.2. Sektionsliste

Enthaellt Angaben ueber Speicherplatz und die Reihenfolge des Kodes der Objektmodule.

Die Syntax der Sektionsliste ist folgende:

```
['-'] [Segmentnummer'=']
($=Ladeanfangsadresse der Sektion1 Sektionsname1
 $=Ladeanfangsadresse der Sektion2 Sektionsname2 usw.)
```

- Die Segmentnummer hat nur eine Bedeutung bei Anwendung des Mikroprozessors U8001. Soll ein Segment nicht in den

Speicher geladen werden, dann ist dieser Segmentnummer ein Minus voranzustellen.

- Die jeweilige Ladeanfangsadresse ist die Adresszuweisung fuer die nachfolgend notierte Sektion. Die Ladeanfangsadressen muessen aufsteigend sein. Es koennen alle vom Assembler oder Linker erzeugten Sektionen benutzt werden. Nicht notierte Sektionen werden standardmaessig an die letzte Sektion der Sektionsliste angehaengt. Soll eine Sektion nicht geladen werden, dann muss der Name dieser mit einem Minuszeichen versehen in der Sektionsliste erscheinen.

2.3. Fenster

Durch das Fenster wird der Speicherbereich des zu ladenden Programms festgelegt.

Die Syntax des Fensters ist folgende:

{Beginnadresse Endadresse}

Die Adressen sind Hexadezimalzahlen. Sollen diese Adressen die Low- bzw. Highadresse der Prozedurdatei sein, dann ist die Option 'O' (s.Abschn.2.4.1.) zu notieren. Die Beginnadresse entspricht dem Eintrittspunkt, wenn mit der Option 'E' kein Eintrittspunkt spezifiziert wird (s.Abschn.2.4.2).

2.4. Optionen

2.4.1. Dateiausgabe auf Diskette

O '=' dateiname

Der durch das Fenster begrenzte Speicherbereich wird in die Datei mit dem Namen 'dateiname' kopiert. Wird die Option nicht verwendet, dann wird keine Datei auf der Diskette erzeugt.

2.4.2. Spezifizieren des Eintrittspunktes

E '=' nnn

Die Hexadezimalzahl 'nnn' spezifiziert den Eintrittspunkt. Ist diese Option nicht angegeben, dann wird als Eintrittspunkt die Beginnadresse des Fensters festgelegt.

3. Beispiel

Es sollen die Objektdateien HAUPTPROG.OBJ und SUBPROG.OBJ durch den IMAGER bearbeitet werden. Die darin enthaltenen Sektionen sollen ab folgenden Adressen geladen werden.

```
STEUERG        Ladeanfangsadresse  100H
DATEN                    "         300H
TIME                     "         600H
PARAMETER                "         700H
```

Der Speicherbereich (Fenster) ist durch die Adressen 100H bzw. 1000 zu begrenzen. Dieser Bereich ist in die Datei mit dem Namen 'REGLER' zu kopieren. Der Eintrittspunkt soll die Adresse 200H haben.

Es ist folgendes Kommando zu notieren:

```
IMAGER HAUPTPROG.OBJ SUBPROG.OBJ ($=100 STEUERG $=300
       DATEN $=600 TIME $=700 PARAMETER) {100 1000}
       O=REGLER E=200
```

Soll die Sektion PARAMETER sich unmittelbar an die Sektion TIME anschliessen und der Eintrittspunkt 100H (Beginnadresse des Speicherbereiches) sein, dann ist das folgende Kommando zu notieren.

```
IMAGER HAUPTPROG.OBJ SUBPROG.OBJ ($=100 STEUERG $=300
       DATEN $=600 TIME) {100 1000} O=REGLER
```

Nach der Abarbeitung des IMAGERS koennen folgende Meldungen erscheinen:

```
'nnn'BYTES LOADED
```

'nnn' ist die Anzahl der geladenen Bytes

```
'nnn'BYTES LOADED 'kkk'BYTES NOT LOADED (OUT OF RANGE)
```

'nnn' ist die Anzahl der geladenen Bytes; 'kkk' ist die Anzahl der Bytes, die ausserhalb des Fensters liegen und somit nicht geladen sind.

```
IMAGER ABORT
```

Der IMAGER laesst sich nicht abarbeiten.

4. Logische Einheiten

Der IMAGER benutzt die folgende logischen Einheiten:

```
2  Fehlermeldung
4  Eingabe-Objektmodul
6  Ausgabe-Programmdatei
```

5. IMAGER-Fehlermeldungen

UNABLE TO OPEN INPUT

Das Programm ist nicht in der Lage, die Eingabe-

Objektdatei zu eroeffnen.

ILLEGAL TAG ENCOUNTER IN FILE: <name>

Der Objektmodul 'name' enthaelt fehlerhaften Objektkode.

ILLEGAL MODULE IN FILE: <name>

Fehlerhafte Moduldefinition in der Objektdatei 'name'.

SECTION ERROR IN FILE: <name>

Eine Sektionsdefinition in der Datei 'name' ist fehlerhaft.

ILLEGAL COMMAND LINE

Syntaxfehler in der Kommandolinie

SYMBOL TABLE OVERFLOW

Der fuer Symbole vorgegebene Platz ist voll.

SEGMENT AND NONSEGMENT MODULES
MAY NOT BE COMBINED

Versuch, segmentierten und nichtsegmentierten Kode zu laden.

UNEXPECTED EOF ENCOUNTERED IN FILE: <name>

Der Objektkode enthaelt fehlerhafte EOF-Marke.

UNABLE TO OPEN PROCEDURE FILE

Prozedurdatei kann nicht geoeffnet werden

PROCEDURE FILE INVALID

Prozedurdatei nicht richtig erzeugt.

UNABLE TO CLOSE PROCEDURE FILE

Prozedurdatei kann nicht abgeschlossen werden.

MESSAGE NOT FOUND

Dem Objektmodule fehlen Informationen beim Start des Moduls. Alte Assemblerversion.

POSSIBLE CODE OVERLAY IN SEGMENT <num>
AT LOCATION <number>

'number' in 'num' wurde mehr als einmal geladen, wahrscheinlich Kodeueberlagerung.

UNRESOLVED EXTERNAL REFERENCE IN FILE <name>

Der Eingabeobjektmodul 'name' enthaelt einen Bezug auf eine undefinierte externe Groesse.

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: UDOS-Software Mikroprozessorsoftware

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

Kombinat VEB

ELEKTRO-APPARATE-WERKE

BERLIN-TREPTOW >FRIEDRICH EBERT<
Hoffmannstraße 15-26, Berlin, DDR-1193
011 2263 eaw 011 2264 eaw

Die Angaben über technische Daten entsprechen dem bei Redaktionsschluß vorliegenden Stand. Änderungen im Sinne der technischen Weiterentwicklung behalten wir uns vor.

Ausgabe August 1986