robotron

Anleitung für den
Programmierer

Anleitung für den
Bediener

Programmpaket für modulare Programmierung

C 1014-0003-1 M 3030

Arbeitsplatzcomputer A 7100 Betriebssystem SCP 1700

SYSTEMUNTERLAGEN-DOKUMENTATION 6/86	Programmpaket für modulare Programmierung – Anleitung für Bediener und Programmierer	MOS
		SCP 1700

Anleitung für Bediener und Programmierer

Programmpaket für modulare Programmierung

AC A7100

VEB Robotron-Projekt Dresden

Die vorliegende Systemunterlagendokumentation, Anleitung für Bediener und Programmierer Programmpaket für modulare Programmierung, entspricht dem Stand von 6/86.

Die Ausarbeitung erfolgte durch ein Kollektiv des VEB Robotron-Elektronik Dresden .

Herausgeber:

VEB Robotron-Projekt Dresden 8010 Dresden, Leningrader Str. 9

Kurzreferat

In der vorliegenden Schrift "Anleitung für Bediener und Programmierer" des SCP 1700 werden die zum "Programmpaket für modulare Programmierung" gehörenden Systemprogramme beschrieben. Dieses Programmpaket enthält:
- den Assembler RASM86
- das Cross-Referenz-Programm XREF86
- den Linker LINK86 und
- den Bibliothekar LIB86.

In Abschnitt 1. wird die Bedienung und der Aufruf von RASM86 erläutert.
In Abschnitt 2. werden die Elemente der RASM86-Assemblersprache definiert, wie z. B. Konstanten, Symbole, Operatoren, Ausdrücke und Zeichenketten.
In Abschnitt 3. werden die zur Assemblersprache gehörenden Direktiven definiert. Sie dienen zur Steuerung des Übersetzungsvorganges, wie z. B. die Definition von Datenelementen, bedingte Assemblierung, Einstellung des Listenformats usw.
In Abschnitt 4. ist der RASM86-Befehlssatz nach funktionellen Gruppen dargestellt. Die Mnemoniks sind die gleichen, wie sie der Assembler ASM86 des Betriebssystems BOS1810 benutzt, mit Ausnahme von vier Mnemoniks (siehe Anlage 1).
In Abschnitt 5. wird die Syntax und die Anwendung von Code-Makros erläutert.
In Abschnitt 6. wird das Cross-Referenz-Programm XREF86 erläutert, das aus der Listen- und Symboldatei des RASM86 eine Übersetzungsliste mit Zeilenzählung und den Cross-Referenz-Anhang liefert.
In Abschnitt 7. wird der Linker LINK86 erläutert, der aus den verschieblichen Objektdateien eine Kommandodatei erzeugt, die unter dem Betriebssystem SCP 1700 lauffähig ist.
In Abschnitt 8. wird der Bibliothekar LIB86 erläutert. LIB86 kann Moduln erzeugen, hinzufügen, ersetzen, herauslösen oder löschen und Informationen über den Inhalt der Bibliothek ausgeben.

Inhaltsverzeichnis

Bildverzeichnis

Tabellenverzeichnis Seite

1. Bedienung von RASM86

1.1. Assembleroperationen

RASM86 übersetzt eine Quelldatei in drei Durchläufen und erzeugt einen Objektcode. RASM86 kann wahlweise drei Ausgabedateien liefern, die in Bild 1 dargestellt sind.

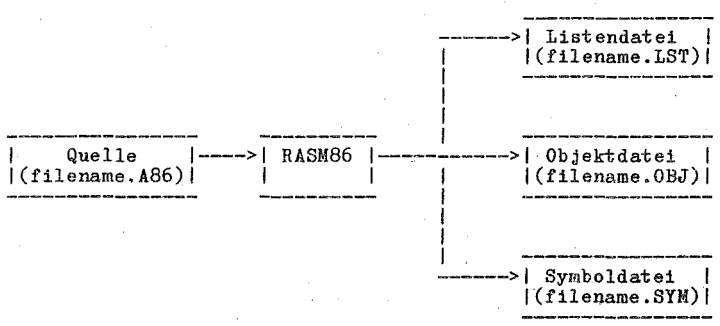


Bild 1: RASM86-Quell- und Objektdateien

Die Listendatei LST enthält die Assemblerliste mit den Fehlermeldungen. Die Objektdatei OBJ enthält den Objektcode in einem verschieblichen Objektformat, das kompatibel zu dem von INTEL ist. Die Symboldatei SYM führt die vom Nutzer definierten Symbole. Die drei Ausgabedateien haben den gleichen Dateinamen wie die Quelldatei. Wenn z. B. die Quelldatei den Namen BIOS88.A86 hat, entstehen durch RASM86 die Dateien BIOS88.OBJ, BIOS88.LST und BIOS88.SYM.

1.2. Aufruf

Der Aufruf von RASM86 erfolgt in der Form:

```
RASM86 filespec [␣parameter ...]
```

Die Angabe der Quelldatei filespec hat die Form:

```
[d:]filename[.typ]
```

d:	wahlfreie Angabe des Laufwerkes für die Quelldatei; ist nicht erforderlich, wenn sich die Quelle auf dem aktuellen Laufwerk befindet
filename	zulässiger Dateiname aus 1 bis 8 Zeichen
typ	zulässiger Dateityp aus 1 bis 3 Zeichen, gewöhnlich A86

RASM86 akzeptiert eine Quelle mit beliebigem Dateityp. Wenn die Angabe des Dateityps in der Kommandozeile fehlt, nimmt RASM86 den Dateityp A86 an.

Beispiele zulässiger RASM86-Kommandos:

```
A>RASM86 B:BIOS88
A>RASM86 BIOS88.A86 ¤AA OB PB SB
A>RASM86 D:TEST
```

Nach dem Aufruf meldet sich RASM86 mit der Ausschrift:

```
RASM86 V x.x
```

wobei x.x die Versionsnummer angibt. RASM86 sucht dann nach der Quelldatei. Existiert die Datei nicht auf dem angegebenen Laufwerk oder hat nicht den richtigen Dateityp, gibt RASM86 die Meldung:

```
NO FILE
```

aus und bricht die Übersetzung ab.

Standardmäßig erzeugt RASM86 die Ausgabedateien auf dem aktuellen Laufwerk. Durch die Benutzung der wahlfreien Parameter bzw. durch eine Laufwerksangabe im Dateinamen der Quelle kann die Ausgabedatei anders gelenkt werden.

Am Ende eines Übersetzungslaufes teilt RASM86 mit:

```
END OF ASSEMBLY. NUMBER OF ERRORS: n USE FACTOR:pp%
```

Der Nutzungsfaktor (USE FACTOR) gibt an, wieviel Platz in der Symboltabelle während der Übersetzung gebraucht wurde. Der Nutzungsfaktor wird in Prozent angegeben (0 bis 99).

1.3. Optionale Laufzeitparameter

Das Dollarzeichen ¤ kennzeichnet eine wahlfreie Zeichenkette der Laufzeitparameter. Ein Parameter besteht aus einem Buchstaben für das Parameterkennzeichen, gefolgt von einem Buchstaben für die Gerätespezifikation. Die Parameter zeigt Tabelle 1.

Tabelle 1: Zusammenfassung der Laufzeitparameter

Parameter	Bedeutung	zulässige Argumente
A	Quelldatei-Gerät	A, B, C, ... P
L	lokale Symbole in der Objektdatei	O
O	Objektdatei-Gerät	A, ... P, Z
P	Listendatei-Gerät	A, ... P, X, Y, Z
S	Symboldatei-Gerät	A, ... P, X, Y, Z

Alle Parameter sind wählbar und können in der Kommandozeile beliebig angeordnet werden. Das Dollarzeichen steht nur einmal am Anfang der Parameterkette. Leerzeichen können die Parameter trennen, sind aber nicht gefordert. Zwischen dem Parameter und dem Gerätenamen darf kein Leerzeichen stehen.
Wenn ein ungültiger Parameter in der Parameterkette steht, gibt RASM86 folgende Meldung aus:

SYNTAX ERROR

Anschließend wird das Kommando ab der Stelle ausgegeben, wo der Fehler steht und mit Fragezeichen abgeschlossen.

Den Parametern A, O, P und S muß ein Gerätename folgen. Die Geräte werden bezeichnet mit:

A, B, C, ... P oder X, Y, Z.

Die Gerätenamen A bis P bezeichnen die Laufwerke A bis P. X spezifiziert die Bedienkonsole (CON:), Y spezifiziert den Drucker (LST:) und Z unterdrückt die Ausgabe (NUL:).
Falls die Ausgabe zur Konsole geführt wird, kann sie vorübergehend mit CTRL/S angehalten und durch Eingabe von CTRL/Q fortgesetzt werden.
Der LO-Parameter bewirkt, daß lokale Symbole in der Objektdatei eingeschlossen werden und in der durch LINK86 erzeugten Symboldatei erscheinen. Andernfalls erscheinen nur PUBLIC-Symbole in der Symboldatei. Die SYM-Datei kann beim Debugger SID86 benutzt werden, um ein Programm einfach zu testen.

Tabelle 2: Beispiele von RASM86-Kommandozeilen

Kommandozeile	Ergebnis
RASM86 IO	Assemblieren der Datei IO.A86, Erstellen von IO.OBJ, IO.LST und IO.SYM, alle auf dem Standardlaufwerk
RASM86 IO.ASM ¤ AD SZ	Assemblieren der Datei IO.ASM auf Laufwerk D, Erstellen von IO.LST und IO.OBJ, Symboldatei wird unterdrückt
RASM86 IO ¤ PY SX	Assemblieren der Datei IO.A86, Erstellen von IO.OBJ, Listenausgabe direkt auf dem Drucker, Ausgabe der Symboldatei auf der Konsole
RASM86 IO ¤ LO	Einschließen lokaler Symbole in IO.OBJ

1.4. Abbruch

RASM86 kann zu jeder Zeit durch Eingabe eines beliebigen Zeichens über die Konsole abgebrochen werden. Wenn eine Eingabe erfolgt, anwortet RASM86 mit der Frage:

```
STOP RASM86 (Y/N)?
```

Ein Y bricht sofort den Übersetzungslauf ab und die Steuerung geht an das Betriebssystem zurück. Ein N setzt die Arbeit von RASM86 fort.

2. Elemente der RASM86-Assemblersprache

2.1. Zeichensatz

RASM86 benutzt eine Untermenge des KOI7-Zeichensatzes. Die zulässigen Zeichen sind die alphanumerischen, die Spezialzeichen und nichtdruckbaren Zeichen:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

a b c d e f g h i j k l m n o p q r s t u v w x y z

0 1 2 3 4 5 6 7 8 9

+ - * / = ( ) [ ] ; ' . ! , _ : @ ¤ ?

Leerzeichen, Tabulator, Wagenrücklauf, Zeilenschaltung

Kleinbuchstaben werden wie Großbuchstaben behandelt, ausgenommen innerhalb von Zeichenketten, in denen nur alphanumerische, spezielle und Leerzeichen benutzt werden dürfen.

2.2. Komponenten und Trennzeichen

Eine Komponente (token) ist die kleinste signifikante Einheit des RASM86-Quellprogramms. Benachbarte Komponenten werden durch ein Leerzeichen getrennt. Überall, wo ein Leerzeichen erlaubt ist, dürfen auch mehrere Leerzeichen stehen. RASM86 erkennt Horizontaltabulatoren als Trennzeichen an und interpretiert sie als Leerzeichen. In der Listendatei werden Tabulatoren durch Leerzeichen ersetzt. Die Tabulatorhalts sind auf jeder achten Spalte.

2.3. Begrenzer

Begrenzer (delimiter) markieren das Ende einer Komponente und geben einem Befehl (instruction) eine spezielle Bedeutung, im Gegensatz zu Trennzeichen, die lediglich das Ende einer Komponente markieren. Beim Auftreten von Begrenzern brauchen Trennzeichen nicht verwendet werden. Trennzeichen nach Begrenzern können jedoch Programme leichter lesbar machen.
Tabelle 3 beschreibt die Trennzeichen und Begrenzer des RASM86. Einige Begrenzer sind außerdem Operatoren und werden näher in Abschnitt 2.6. erklärt.

Tabelle 3: Trennzeichen und Begrenzer

Zeichen	Name	Benutzung
20H	Leerzeichen	Trennzeichen
09H	Tabulator	Trennzeichen, erlaubt in Quelldateien, wirksam in Listendateien
(CR)	Wagenrücklauf	beendet eine Quellzeile
(LF)	Zeilenschaltung	erlaubt nach (CR); innerhalb einer Quellzeile wird es als Leerzeichen interpretiert
;	Semikolon	Beginn des Kommentarfeldes
:	Doppelpunkt	kennzeichnet eine Marke; wird benutzt bei der Segment-Override-Spezifikation
.	Punkt	bildet Variable aus Zahlen
¤	Dollarzeichen	aktueller Speicherplatzzähler; wird ignoriert in Bezeichnern und Zahlen
+	Plus	Arithmetikoperator für Addition
-	Minus	Arithmetikoperator für Subtraktion
*	Stern	Arithmetikoperator für Multiplikation
/	Schrägstrich	Arithmetikoperator für Division
@	kommerzielles a	zulässig in Bezeichnern
_	Unterstreichung	zulässig in Bezeichnern
!	Ausrufezeichen	logisches Ende für eine Anweisung; erlaubt mehrere Anweisungen auf einer Zeile
'	Apostroph	begrenzt Zeichenkonstanten

2.4. Konstanten

Eine Konstante ist ein zur Zeit der Übersetzung bekannter Wert, der sich während der Programmausführung nicht ändert. Eine Konstante kann entweder eine Integer-Zahl oder eine Zeichenkette sein.

2.4.1. Numerische Konstanten

Eine numerische Konstante ist ein 16-bit-Wert, der sich auf eine von mehreren Basen bezieht. Die Zahlenbasis (radix) wird durch einen der Zahl folgenden Indikator angegeben. Die Zahlenbasis-Indikatoren zeigt Tabelle 4.

Tabelle 4: Zahlenbasis-Indikatoren für Konstanten

Indikator	Konstantentyp	Basis
B	binär	2
O	oktal	8
Q	oktal	8
D	dezimal	10
H	hexadezimal	16

RASM86 setzt voraus, daß jede Konstante, der kein Zahlenbasis-Indikator folgt, dezimal ist. Zahlenbasis-Indikatoren können mit Klein- oder Großbuchstaben angegeben werden.
Eine Konstante ist demzufolge eine Folge von Ziffern mit einem nachfolgenden Zahlenbasis-Indikator, wobei sich die Ziffern im Bereich der Zahlenbasis bewegen. Binärkonstanten bestehen aus Nullen und Einsen. Oktalkonstanten enthalten Ziffern von 0 ... 7, dezimale Konstanten enthalten Ziffern von 0 ... 9. Hexadezimale Konstanten enthalten dezimale Ziffern und die hexadezimalen Ziffern A (10D), B (11D), C (12D), D (13D), E (14D) und F (15D). Es ist zu beachten, daß das führende Zeichen einer hexadezimalen Konstanten eine Dezimalziffer oder Null sein muß, da RASM86 sonst nicht in der Lage ist, eine hexadezimale Konstante von einem Bezeichner zu unterscheiden.

Beispiele zulässiger Konstanten:

```
1234     1234D    1100B    1111000011110000B
1234H    0FFEH    33770    13772Q
33770    0FE3H    1234D    0FFFFH
```

2.4.2. Zeichenketten

Eine Zeichenkettenkonstante ist eine Reihe von KOI7-Zeichen, die durch Apostrophe begrenzt werden. Alle RASM86-Befehle, die numerische Konstanten als Argumente zulassen, akzeptieren nur Zeichenkonstanten aus einem oder zwei Zeichen als zulässige Argumente. Alle Befehle behandeln eine Ein-Zeichen-Konstante als 8-bit-Wert. Eine Zwei-Zeichen-Kette wird als 16-bit-Wert dargestellt mit dem Wert des zweiten Zeichens im niederwertigen und dem Wert des ersten Zeichens im höherwertigen Byte.
Der numerische Wert eines Zeichens ist sein KOI7-Code. RASM86 übersetzt innerhalb von Zeichenketten nicht, so daß sowohl Klein- als auch Großbuchstaben verwendet werden können. Es ist zu beachten, daß in Zeichenketten nur alphanumerische, Sonder- und Leerzeichen verwendet werden dürfen.
Die DB-Direktive ist die einzige Anweisung, die mehr als zwei Zeichen in einer Zeichenkette zuläßt. Die Zeichenkette darf 255 Bytes nicht überschreiten. Soll innerhalb der Zeichenkette ein Apostroph gedruckt werden, ist der Apostroph doppelt anzugeben. RASM86 interpretiert zwei Apostrophe als einen.
In den folgenden Beispielen werden zulässige Zeichenketten und ihre Wirkung bei Ausführung gegenübergestellt.

```
            'a'   -->  a
        'Ab''Cd'  -->  Ab'Cd
   'I like SCP'   -->  I like SCP
           ''''   -->  '
'ONLY UPPER CASE' -->  ONLY UPPER CASE
'only lower case' -->  only lower case
```

2.5. Bezeichner

Die folgenden Regeln gelten für alle Bezeichner (identifier):

- Bezeichner können bis zu 80 Zeichen lang sein.

- Das erste Zeichen muß ein Buchstabe sein, oder eines der Spezialzeichen ?, @, _(underscore).

- Jedes nachfolgende Zeichen kann entweder ein Buchstabe, eine Zahl oder ein Spezialzeichen ?, @, _ oder ¤ sein. RASM86 ignoriert das Spezialzeichen ¤ in Bezeichnern, so daß die Lesbarkeit langer Bezeichner verbessert werden kann. So wird z. B. der Bezeichner INTERRUPT¤FLAG von RASM86 als INTERRUPTFLAG behandelt.

Es gibt zwei Arten von Bezeichnern. Die erste Art sind Schlüsselworte, die für den Assembler eine vorgegebene Bedeutung haben. Die zweite Art von Bezeichnern sind Symbole, die vom Nutzer definiert werden. Die folgenden sind zulässige Bezeichner:

```
NOLIST
WORD
AH
THIRD_STREED
HOW_ARE_YOU_TODAY
VARIABLE@NUMBER@1234567890
```

2.5.1. Schlüsselworte

Schlüsselworte sind für die Benutzung des RASM86 reserviert. Der Nutzer darf keinen Bezeichner definieren, der mit einem Schlüsselwort identisch ist. In Anlage 4 ist eine komplette Liste aller Schlüsselworte zusammengestellt.
RASM86 erkennt fünf Arten von Schlüsselworten:

- Befehle
- Direktiven
- Operatoren
- Register
- vordefinierte Zahlen

Die Befehlsschlüsselworte und ihre Wirkung werden in Abschnitt 4. beschrieben. Direktiven werden in Abschnitt 3. und Operatoren in Abschnitt 2.6. beschrieben. Die Tabelle 5 enthält die RASM86-Schlüsselworte, die Register darstellen.
Drei Schlüsselworte sind vordefinierte Zahlen: BYTE, WORD und DWORD. Die Werte dieser Zahlen sind 1, 2 und 4. Mit jeder dieser Zahlen ist ein Typattribut verbunden. Das Typattribut ist gleich dem numerischen Wert des Schlüsselwortes.

Tabelle 5: Registerschlüsselworte

Register-symbol	Größe in Bytes	numerischer Wert	Bedeutung
AH	1	100B	Akkumulator, höherwertiges Byte
BH	1	111B	Basisregister, höherwertiges Byte
CH	1	101B	Zählregister, höherwertiges Byte
DH	1	110B	Datenregister, höherwertiges Byte
AL	1	000B	Akkumulator, niederwertiges Byte
BL	1	011B	Basisregister, niederwertiges Byte
CL	1	001B	Zählregister, niederwertiges Byte
DL	1	010B	Datenregister, niederwertiges Byte
AX	2	000B	Akkumulator, Wort
BX	2	011B	Basisregister, Wort
CX	2	001B	Zählregister, Wort
DX	2	010B	Datenregister, Wort
BP	2	101B	Basiszeiger (Basepointer)
SP	2	100B	Kellerzeiger (Stackpointer)
SI	2	110B	Quellindex (Source-Index)
DI	2	111B	Zielindex (Destination-Index)
CS	2	01B	Code-Segmentregister
DS	2	11B	Data-Segmentregister
SS	2	10B	Stack-Segmentregister
ES	2	00B	Extra-Segmentregister

2.5.2. Symbole und ihre Attribute

Ein Symbol ist ein vom Nutzer definierter Bezeichner. Seine Attribute geben Auskunft über die Informationsart des Symbols. Es gibt drei Kategorien von Symbolen:

- Variable
- Marken
- Zahlen

Eine Variable identifiziert die gespeicherten Daten in einem bestimmten Speicherplatz. Alle Variablen haben die folgenden drei Attribute:

- Segment: legt fest, in welchem Segment die Variable definiert wurde.

- Offset: legt fest, wieviel Bytes sich zwischen dem Anfang des Segments und der Speicheradresse befinden, die diese Variable bezeichnet.

- Typ: legt fest, wieviel Bytes behandelt werden, falls zu dieser Variablen zugegriffen wird.

Ein Segment kann ein Code-Segment, Data-Segment, Stack-Segment oder Extra-Segment sein. Das hängt von seinem Inhalt und dem Register ab, das die Startadresse des Segments enthält (siehe Abschnitt 3.2.). Die Startadresse des Segments ist eine Zahl zwischen 0 und 65535D. Diese Zahl stellt die Paragraphadresse im Speicher dar, wenn entweder das Programm übersetzt, verbunden (linked) oder geladen wird.
Der Offset einer Variablen ist die Adresse der Variablen relativ zur Startadresse im Segment. Der Offset einer Variablen kann eine Zahl zwischen 0 und OFFFFH oder 65535D sein.
Eine Variable hat eines der folgenden Typattribute:

- BYTE
- WORD
- DWORD

BYTE gibt eine Ein-Byte-Variable, WORD eine Zwei-Byte-Variable und DWORD eine Vier-Byte-Variable an. Die Direktiven DB, DW und DD definieren Variable dieser drei Typen (siehe Abschnitt 3.). Eine Variable wird z. B. als Name vor einer Speicher-Direktive definiert:

```
MY_VARIABLE DB 0
```

Über die EQU-Anweisung kann auch zu einer anderen Variablen zugegriffen werden.

```
ANOTHER_VARIABLE EQU MY_VARIABLE
```

Marken kennzeichnen Speicherplätze, die Befehlsanweisungen enthalten. Sprünge (jumps) und Rufe (calls) beziehen sich auf Marken. Alle Marken haben zwei Attribute:

- Segment
- Offset

Markenattribute sind im wesentlichen gleich denen der Variablen. Eine Marke wird im allgemeinen einem Befehl vorangestellt und durch einen Doppelpunkt von ihm getrennt.

```
MY_LABEL:  ADD  AX,BX
```

Eine Marke kann auch vor der EQU-Anweisung stehen (ohne Doppelpunkt), der eine andere Marke zugeordnet wird.

```
ANOTHER_LABEL  EQU  MY_LABEL
```

Zahlen können auch wie Symbole definiert werden. RASM86 behandelt solch ein Symbol so, als wäre die vom Symbol dargestellte Zahl explizit kodiert.

```
NUMBER_FIVE  EQU  5
             MOV  AL,NUMBER_FIVE
```

entspricht:

```
MOV  AL,5
```

In Abschnitt 2.6. werden die Operatoren und ihre Wirkung auf Zahlen und Zahlensymbole beschrieben.

2.6. Operatoren

Es gibt verschiedene Arten von Operatoren:

- arithmetische Operatoren
- logische Operatoren
- Vergleichsoperatoren
- Segment-Override-Operatoren
- Attributoperatoren

Tabelle 6 definiert die RASM86-Operatoren. In dieser Tabelle stellen a und b zwei Elemente des Ausdrucks dar. Die Gültigkeitsspalte definiert den Typ der Operanden, die der Operator behandeln kann.
In dieser Tabelle beziehen sich Zahlen auf absolute Zahlen. Das sind Zahlen, deren Werte zur Übersetzungszeit bekannt sind, so wie numerische Konstanten. Eine verschiebliche Zahl ist eine Zahl, deren Wert zur Übersetzungszeit unbekannt ist, weil sie sich während des Link-Vorganges ändern kann. So ist z. B. der Offset einer Variablen, die in einem Segment Speicherplatz belegt, zur Link-Zeit eine verschiebliche Zahl, wenn das Segment mit anderen verbunden werden soll.

Tabelle 6: RASM86-Operatoren

Syntax	Ergebnis	Gültigkeit
	logische Operatoren	
a XOR b	logisches exklusives ODER von a und b, Bit für Bit	a,b = Zahl
a OR b	logisches ODER von a und b, Bit für Bit	a,b = Zahl
a AND b	logisches AND von a und b, Bit für Bit	a,b = Zahl
NOT a	logische Inversion von a; 0 --> 1, 1 --> 0	a = 16-bit-Zahl
	Vergleichsoperatoren	
a EQ b	Ergebnis OFFFFH, falls a = b, sonst Null	a,b = vorzeichenlose Zahl, Marke, Variable oder verschiebliche Zahl aus gleichem Segment
a LT b	Ergebnis OFFFFH, falls a < b, sonst Null	a,b = vorzeichenlose Zahl, Marke, Variable oder verschiebliche Zahl aus gleichem Segment
a LE b	Ergebnis OFFFFH, falls a <= b, sonst Null	a,b = vorzeichenlose Zahl, Marke, Variable oder verschiebliche Zahl aus gleichem Segment
a GT b	Ergebnis OFFFFH, falls a > b, sonst Null	a,b = vorzeichenlose Zahl, Marke, Variable oder verschiebliche Zahl aus gleichem Segment
a GE b	Ergebnis OFFFFH, falls a >= b, sonst Null	a,b = vorzeichenlose Zahl, Marke, Variable oder verschiebliche Zahl aus gleichem Segment
a NE b	Ergebnis OFFFFH, falls a <> b, sonst Null	a,b = vorzeichenlose Zahl, Marke, Variable oder verschiebliche Zahl aus gleichem Segment

Tabelle 6: (Fortsetzung)

Syntax	Ergebnis	Gültigkeit
	arithmetische Operatoren	
a + b	arithmetische Summe von a und b	a = Zahl, Variable, Marke, verschiebliche Zahl oder externe Größe, b = Zahl
a - b	arithmetische Differenz von a und b	a = Zahl, Variable, Marke, verschiebliche Zahl oder externe Größe, b = Zahl, Variable, Marke oder verschiebliche Zahl im gleichen Segment wie a
a * b	vorzeichenlose Multiplikation von a und b	a,b = Zahl
a / b	vorzeichenlose Division von a und b	a,b = Zahl
a MOD b	Rest von a / b	a,b = Zahl
a SHL b	Wert, der sich durch b-malige Linksverschiebung von a ergibt	a,b = Zahl
a SHR b	Wert, der sich durch b-malige Rechtsverschiebung von a ergibt	a,b = Zahl
+a	ergibt a	a = Zahl
-a	ergibt 0 - a	a = Zahl
	Segment-Override	
seg-reg: addr-exp	Änderung des Segmentregisters für Assembler	seg-reg = CS, DS, SS, ES

Tabelle 6: (Fortsetzung)

Syntax	Ergebnis	Gültigkeit
	Attribut-Operatoren	
SEG a	ergibt eine Zahl, deren Wert der Segment-Wert der Variablen oder Marke ist	a = Marke oder Variable
OFFSET a	ergibt eine Zahl, deren Wert der Offset-Wert der Variablen oder Marke ist	a = Marke oder Variable
TYPE a	ergibt eine Zahl, die den Wert 1, 2 oder 4 hat, wenn a vom Typ BYTE, WORD oder DWORD ist	a = Marke oder Variable
LENGTH a	ergibt eine Zahl, deren Wert das Längenattribut von a ist. Das Längenattribut ist die zur Variablen gehörende Byteanzahl.	a = Variable
LAST a	falls Länge a > 0, dann LAST a = LENGTH a - 1; falls a = 0, dann LAST a = 0	a = Variable
a PTR b	ergibt eine virtuelle Variable mit dem Typ von a und den Attributen von b	a = BYTE, WORD oder DWORD b = addr-exp
.a	ergibt eine Variable mit dem Offset-Attribut von a. Das Segment-Attribut ist das aktuelle Data-Segment.	a = Zahl
$	ergibt eine Marke mit dem Offset zum aktuellen Speicherplatz. Das Segment-Attribut wird das aktuelle Segment.	kein Argument

2.6.1. Operatorbeispiele

Logische Operatoren akzeptieren nur Zahlen als Operanden. Sie führen die Boolschen logischen Operationen AND, OR, XOR und NOT aus.

Beispiel:

```
  00FC            MASK    EQU     0FCH
  0080            SIGNBIT EQU     80H
0000 B180                 MOV     CL,MASK AND SIGNBIT
0002 B003                 MOV     AL,NOT MASK
```

Vergleichsoperatoren behandeln alle Operanden als vorzeichenlose Zahlen. Die Vergleichsoperatoren sind EQ (equal), LT (less than), LE (less than or equal), GE (greater than or equal), GT (greater than) und NE (not equal). Jeder Operator vergleicht zwei Operanden und liefert den Wert OFFFFH, wenn der angegebene Vergleich 'wahr' ist, bzw. den Wert Null, falls der Vergleich 'falsch' ist.

Beispiel:

```
  000A            LIMIT1 EQU      10
  0019            LIMIT2 EQU      25
                         .
                         .
                         .
0004 B8FFFF              MOV      AX,LIMIT1 LT LIMIT2
0007 B80000              MOV      AX,LIMIT1 GT LIMIT2
```

Die Additions- und Subtraktionsoperatoren berechnen die arithmetische Summe bzw. Differenz von zwei Operanden. Der erste Operand kann eine Variable, Marke oder Zahl, der zweite Operand muß bei Addition eine Zahl sein. Bei Subtraktion kann der zweite Operand eine Zahl oder eine Variable bzw. Marke aus dem Segment des ersten Operanden sein.
Wenn eine Zahl zu einer Variablen oder Marke addiert wird, so ist das Ergebnis eine Variable oder Marke, deren Offset der numerische Wert des zweiten Operanden plus dem Offset des ersten Operanden ist.
Die Subtraktion von einer Variablen oder Marke liefert eine Variable oder Marke, deren Offset die Differenz des Offsets des ersten Operanden und der im zweiten Operanden angegebenen Zahl ist.

Beispiel:

```
  0002            COUNT   EQU     2
  0005            DISP1   EQU     5
000A FF           FLAG    DB      0FFH
                          .
                          .
                          .
000B A00B00               MOV     AL,FLAG+1
000E 8A0E0F00             MOV     CL,FLAG+DISP1
0012 B303                 MOV     BL,DISP1-COUNT
```

Die Multiplikations- und Divisionsoperatoren *, /, MOD, SHL und SHR akzeptieren nur Zahlen als Operanden. * und / behandeln alle

Operatoren als vorzeichenlose Zahlen.

Beispiel:

```
0016 BE5500                MOV       SI,256/3
0019 B310                  MOV       BL,64/4
  0050          BUFFERSIZE EQU       80
001B B8A000                MOV       AX,BUFFERSIZE*2
```

Einstellige Operatoren können sowohl vorzeichenlose, als auch Operatoren mit Vorzeichen sein.

Beispiel:

```
001E B123                  MOV       CL,+35
0020 B007                  MOV       AL,2--5
0022 B2F4                  MOV       DL,-12
```

Wenn Variable behandelt werden, entscheidet der Assembler, welches Segmentregister zu benutzen ist. Diese Entscheidung kann durch Spezifizierung eines anderen Registers mittels des sogenannten Segment-Override übergangen werden. Die Syntax für den Segment-Override-Operator ist:

segment-register : address-expression

Für segment-register steht CS, DS, SS oder ES.

Beispiel:

```
0024 368B472D              MOV       AX,SS:WORDBUFFER[BX]
0028 268B0E5B00            MOV       CX,ES:ARRAY
002D 26A4                  MOVS      BYTE PTR[DI],ES:[SI]
```

Ein Attributoperator erstellt eine Zahl, die gleich einem Attributwert seines Variablenoperanden ist. SEG bildet den Segment-Wert der Variablen, OFFSET ihren Offset-Wert, TYPE ihren Typwert (1, 2, 4) und LENGTH die Byteanzahl, die zur Variablen gehört. LAST vergleicht die Länge der Variablen mit 0 und, falls sie größer ist, wird die Länge LENGTH um 1 vermindert. Falls LENGTH gleich 0 ist, erfolgt keine Änderung. Attributoperatoren akzeptieren nur Variable als Operatoren.

Beispiel:

```
002D 000000000000  WORDBUFFER DW        0,0,0
0033 0102030405    BUFFER     DB        1,2,3,4,5
                              .
                              .
                              .
0038 B80500                   MOV       AX,LENGTH BUFFER
003B B80400                   MOV       AX,LAST BUFFER
003E B80100                   MOV       AX,TYPE BUFFER
0041 B80200                   MOV       AX,TYPE WORDBUFFER
```

Der PTR-Operator erstellt eine virtuelle Variable oder Marke, die nur während der Befehlsausführung zulässig ist. Die Operanden werden nicht verändert. Das temporäre Symbol hat das gleiche Typattribut wie der linke Operator und alle anderen Attribute das des rechten Operators.

Beispiel:

```
0044 C60705                MOV       BYTE PTR [BX],5
0047 8A07                  MOV       AL,BYTE PTR [BX]
0049 FF04                  INC       WORD PTR [SI]
```

Der Punktoperator (.) erstellt eine Variable im aktuellen Data-Segment. Die neue Variable hat ein Segment-Attribut gleich dem des aktuellen Data-Segments und ein Offset-Attribut gleich seinem Operanden. Sein Operand muß eine Zahl sein.

Beispiel:

```
004B A10000                MOV       AX,.0
004E 268B1E0040            MOV       BX,ES:.4000H
```

Der Dollaroperator (¤) erstellt eine Marke mit einem Offset-Attribut gleich dem Wert des aktuellen Speicherplatzzählers (location-counter). Der Segment-Wert der Marke ist der gleiche wie der des aktuellen Segments. Der Operator benötigt keinen Operanden.

Beispiel:

```
0053 E9FDFF                JMP       ¤
0056 EBFE                  JMPS      ¤
0058 E9FD2F                JMP       ¤+3000H
```

2.6.2. Vorrang bei Operatoren

In Ausdrücken werden Variable, Marken oder Zahlen mit Operatoren kombiniert. RASM86 erlaubt verschiedene Arten von Ausdrücken, die in Abschnitt 2.7. erklärt werden. In diesem Abschnitt wird die Reihenfolge definiert, in der Operationen ausgeführt werden, die mehr als einen Operator in einem Ausdruck aufweisen.
RASM86 wertet die Ausdrücke von links nach rechts aus, wobei Operatoren mit höherem Vorrang vor denen mit niederem Vorrang ausgeführt werden. Wenn zwei Operatoren den gleichen Vorrang haben, wird von links nach rechts abgearbeitet. Mit runden Klammern können die Vorrangregeln übergangen werden. Falls Klammern gesetzt sind, wird die innerste zuerst ausgeführt. Es sind maximal fünf Schachtelungen mit Klammern zulässig.

Beispiel:

```
15 / 3 + 18 / 9 = 5 + 2 = 7
15 / (3 + 18 / 9) = 15 / (3 + 2) = 15 / 5 = 3
(20 * 4) + ((27 / 9 - 4 / 2)) = (20 * 4) + (3 - 2) = 80 + 1 = 81
```

Tabelle 7: Vorrang der Operatoren bei RASM86

Reihenfolge	Operatortyp	Operatoren
1	logisch	XOR, OR
2	logisch	AND
3	logisch	NOT
4	Vergleichsoperatoren	EQ, LT, LE, GT, GE, NE
5	Addition/Subtraktion	+, -
6	Multiplikation/Division	*, /, MOD, SHL, SHR
7	einstellige Operatoren	+, -
8	Segment-Override	segment-override:
9	Attributoperatoren	SEG, OFFSET, PTR, TYPE, LENGTH, LAST
10	runde, eckige Klammern	( ), [ ]
11	Punkt und Dollar	., ¤

2.7. Ausdrücke

RASM86 erlaubt Adreß-, numerische und Klammerausdrücke.
Ein Adreßausdruck ergibt eine Speicheradresse und hat drei Komponenten:

- Segment-Wert
- Offset-Wert
- Typ

Sowohl Variable als auch Marken sind Adreßausdrücke. Die Komponenten eines Adreßausdruckes sind Zahlen. Sie können mit Operatoren kombiniert werden, z. B. mit PTR, um einen Adreßausdruck zu erhalten.
Ein numerischer Ausdruck darf weder Variable noch Marken enthalten, nur Zahlen und Operanden. Das Ergebnis eines numerischen Ausdrucks ist eine Zahl.
Ausdrücke in eckigen Klammern geben Basis- und Indexadressierungsmodi an. Die Basisregister sind BX und BP, die Indexregister DI und SI. Ein Ausdruck in eckigen Klammern kann aus einem Basisregister, einem Indexregister oder aus einem Basis- und Indexregister bestehen.

Beispiel:

```
MOV     VARIABLE[BX],0
MOV     AX,[BX+DI]
MOV     AX,[SI]
```

Der Operator + muß zwischen einem Index- und Basisregister zur Darstellung der Index- und Basisregisteradressen benutzt werden.

2.8. Anweisungen

Eine Anweisung (statement) kann sowohl ein Befehl (instruction) als auch eine Direktive (directive) sein. RASM86 übersetzt Befehle in die Maschinensprache, Direktiven aber nicht. Direktiven sind Informationen für den Assembler, um gewisse Funktionen auszuführen.
Jede Assembleranweisung muß mit Wagenrücklauf (CR) und Zeilenschaltung (LF) oder mit Ausrufezeichen (!) abgeschlossen werden. RASM86 versteht diese Zeichen als Zeilenende (end-of-line). Mehrere Anweisungen können auf einer Zeile geschrieben werden, wenn sie durch ! getrennt werden. Nur die letzte Anweisung einer Zeile kann Kommentar enthalten, wenn er die Zeile abschließt.

Der RASM86-Befehlssatz wird genauer im Abschnitt 4. beschrieben. Die Syntax für Befehlsanweisungen ist folgende:

```
[label:]  [prefix] mnemonic [operand ...][;comment]
```

Es bedeuten:

label — Ein Symbol, gefolgt von(:), definiert eine Marke für den aktuellen Wert des Adreßzählers (location-counter) im aktuellen Code-Segment. Das Feld ist wahlweise belegbar.

prefix — Bestimmte Maschinenbefehle, wie z. B. LOCK und REP, können vor Maschinenbefehlen stehen. Das Feld ist wahlweise belegbar.

mnemonic — Ein Symbol, das als Maschinenbefehl definiert wird. Das kann sowohl vom Assembler selbst als auch durch eine EQU-Direktive erfolgen. Dieses Feld ist frei wählbar, falls es nicht durch einen Präfixbefehl eingeleitet wird. Wird es weggelassen, dürfen keine Operanden angegeben werden, aber die anderen Felder können erscheinen.

operand — Ein Befehlsmnemonik kann weitere Symbole fordern, die zum Befehl gehörende Operanden darstellen. Befehle können keinen, einen oder zwei Operanden enthalten.

comment — Jedes Semikolon außerhalb einer Zeichenkette definiert den Beginn eines Kommentars, der mit Wagenrücklauf endet. Kommentare verbessern die Lesbarkeit und das Testen von Programmen. Das Feld ist frei wählbar.

RASM86-Direktiven werden in Abschnitt 3. beschrieben. Die Syntax für Direktiveanweisungen ist folgende:

```
[name] directive operand ... [;comment]
```

Es bedeuten:

name	Namen sind erlaubt vor CSEG-, DSEG-, ESEG-, SSEG-, GROUP-, DB-, DW-, DD-, RB-, RW-, RD-, RS- und EQU-Direktiven. Vor EQU und GROUP muß ein Name stehen, bei den übrigen Direktiven ist er nicht erforderlich. Im Gegensatz zum Markenfeld eines Befehls wird das Namenfeld einer Direktive niemals mit einem Doppelpunkt (:) begrenzt.
directive	Direktiveschlüsselwort
operand	Es besteht Analogie zu den Operanden der Befehlsmnemoniks. Einige Direktiven lassen jeden Operanden zu, wie DB und DW, während andere spezielle Forderungen stellen.
comment	Kommentar (wie bei Maschinenbefehlen)

3. Assembler-Direktiven

Assembler-Direktiven steuern den Übersetzungsvorgang durch Erfüllung gewisser Funktionen, wie die Zuweisung von Codeteilen zu logischen Segmenten, die bedingte Assemblierung, die Definition von Datenelementen, Speicherzuweisungen, Einstellen des Listenformats und Einfügen von Quelltext externer Dateien.

Assembler-Direktiven werden in folgende Kategorien unterteilt:

- Segmentsteuerung
  CSEG DSEG ESEG SSEG GROUP

- Modulsteuerung
  NAME PUBLIC EXTRN END

- Bedingte Assemblierung
  IF ELSE ENDIF

- Symboldefinition
  EQU

- Datendefinition und Speicherzuweisung
  DB DW DD RS RB RW RD

- Listenausgabe
  EJECT IFLIST/NOIFLIST LIST/NOLIST PAGESIZE PAGEWIDTH SIMFORM TITLE

- weitere Direktiven
  INCLUDE ORG

3.1. Segmente

Die Zentraleinheit kann bis ein Megabyte (1 048 576 Bytes) adressieren. Dieser gesamte Adreßbereich kann willkürlich in eine Anzahl kleiner Einheiten unterteilt werden, die Segmente genannt werden. Diese können bis 64K Bytes lang sein. Jedes Segment besteht aus aneinandergrenzenden Speicherplätzen, die eine logisch selbständige und seperate Adreßeinheit bilden. Jedes Segment muß eine Basisadresse haben, die die Startadresse im Speicher angibt. Jede Segmentbasisadresse muß an einer durch 16 teilbaren Grenze beginnen, wobei es aber keine anderen Einschränkungen über Segmentgrenzen gibt. Jeder Speicherplatz im Speicherbereich hat eine physische und logische Adresse. Die physische Adresse ist ein 20-bit-Wert, der ein einziges Byte innerhalb des Speicherbereiches adressiert. Die logische Adresse ist eine Kombination von einem 16-bit-Segmentbasiswert und einem 16-bit-Offset-Wert. Dieser Offset-Wert ist die relative Adresse zur Segmentbasis. Zur Laufzeit ist jede Speicherbeziehung die Kombination von Segmentbasiswert und einem Offset-Wert, die die physische 20-bit-Adresse bilden. Die physische Adresse kann in mehr als einem logischen Segment enthalten sein.
Die Zentraleinheit kann zur Laufzeit zu vier Segmenten zugreifen. Die Basisadresse von jedem Segment ist in einem Segmentregister enthalten. Das CS-Register zeigt zum aktuellen Code-Segment, das die Befehle enthält. Das DS-Register zeigt zum aktuellen Data-Segment, das gewöhnlich die Programmvariablen enthält. Das SS-Register zeigt zum aktuellen Stack-Segment, wo Stack-Operationen, wie zeitweiliges Zwischenspeichern oder Parametereintragungen,

gemacht werden. Das ES-Register zeigt zum aktuellen Extra-Segment, das Daten enthält.
Mit RASM86-Segment-Direktiven ist es möglich, das Assemblerquellprogramm in Segmente aufzuteilen entsprechend den Speichersegmenten, in die der Objektcode zur Laufzeit geladen wird.
Die Größe und der Typ von Segmenten, die in einem Programm benutzt werden, bestimmen den Typ des vom Betriebssystem benutzten Speichermodells.
Es kann der gesamte Code mit den Daten in einem einzigen 64K-Segment vereint werden, oder es können getrennte Code- und Data-Segmente gebildet werden, jedes bis 64K lang. Mit RASM86 kann auch eine willkürliche Anzahl von Code-, Data-, Stack- und Extra-Segmenten gebildet werden, um den gesamten Adreßbereich des Prozessors zu nutzen. Deshalb können mehr als 64K für Code und Daten mittels verschiedener Segmente und deren Verwaltung durch die Assembler-Direktiven genutzt werden.

3.2. Segment-Direktive

Jeder Befehl und jede Variable in einem Programm müssen in einem Segment enthalten sein. Befehlsanweisungen müssen dem Code-Segment zugewiesen sein, während Direktiveanweisungen beliebigen Segmenten zugewiesen sein können. Erzeugt und benannt wird ein Segment mit folgender Segment-Direktive:

```
[segment-name] segment [align-type] [combine-typ] ['class-name']
```

wobei segment eines der folgenden Segmente sein kann:

- CSEG (Code-Segment)
- DSEG (Data-Segment)
- ESEG (Extra-Segment)
- SSEG (Stack-Segment)

Beispiel:

```
DATASEG   DSEG   PARA    'DATA'
CODE1     CSEG   BYTE
XYZ       DSEG   WORD    COMMON
```

3.2.1. Segmentname (segment-name)

Der Segmentname kann ein beliebiger RASM86-Bezeichner sein. Wenn kein Segmentname angegeben ist, setzt RASM86 einen Standardnamen ein. Tabelle 8 zeigt die Standardnamen.

Tabelle 8: Standard-Segmentnamen

Segment-Direktive	Standardnamen
CSEG	CODE
DSEG	DATA
ESEG	EXTRA
SSEG	STACK

Wenn einmal eine Segment-Direktive benutzt wurde, weist RASM86 dem angegebenen Segment solange Anweisungen zu, bis eine andere Segment-Direktive auftritt. RASM86 verbindet alle Segmente mit dem gleichen Segmentnamen, auch wenn sie nicht zusammenhängend im Quellcode sind.

3.2.2. Zuordnungstyp (align-type)

Der Zuordnungstyp legt für den Linker eine spezielle Segmentgrenze fest. Der Linker benutzt diese Information, um Segmente zu verbinden, die eine arbeitsfähige Datei erzeugen. Folgende vier Typen sind möglich:

- BYTE (Bytezuweisung)
- WORD (Wortzuweisung)
- PARA (Paragraphzuweisung)
- PAGE (Seitenzuweisung)

Wenn der Zuordnungstyp angegeben wird, muß dies in der ersten Segmentdefinition sein. Bei nachfolgenden Segmenten kann der Zuordnungstyp weggelassen werden, wenn sie den gleichen Namen haben, aber der ursprüngliche Wert darf nicht geändert werden. Wenn kein Zuordnungstyp angegeben wird, setzt RASM86 einen Standardwert ein. Tabelle 9 zeigt die Standardnamen.

Tabelle 9: Standard-Zuordnungstypen

Segment-Direktive	Standard-Zuordnungstypen
CSEG	BYTE
DSEG	WORD
ESEG	WORD
SSEG	WORD

BYTE-Zuweisung bedeutet, daß das Segment ab dem nächsten Byte beginnt, das dem vorhergehenden Segment folgt.
WORD-Zuweisung bedeutet, daß das Segment an einer geraden Grenze beginnt. Eine gerade Grenze ist eine hexadezimale Adresse, die auf 0, 2, 4, 6, 8, A, C oder E endet. In gewissen Fällen kann die WORD-Zuweisung die Zugriffsgeschwindigkeit erhöhen, weil die ZVE nur einen Speicherzyklus benötigt, wenn ein Zugriff zu WORD-Variablen im Segment erfolgt, die gerade Adressen haben.
PARA-Zuweisung bedeutet, daß das Segment an einer Paragraphgrenze beginnt, d. h. eine Adresse, deren vier niedrigste Bit Null sind.
PAGE-Zuweisung bedeutet, daß das Segment an einer Page-Grenze beginnt. Das ist eine Adresse, deren niederwertiges Byte gleich Null ist.

3.2.3. Verbindetyp (combine-typ)

Der Verbindetyp bestimmt, wie der Linker ein Segment mit anderen Segmenten gleichen Namens verbinden kann. Folgende fünf verschiedene Typen können angegeben werden:

- PUBLIC
- COMMON
- STACK
- LOCAL
- nnnn (absolutes Programm)

Wenn der Verbindetyp angegeben wird, muß das in der ersten Segmentdefinition erfolgen. Er kann bei nachfolgenden Segment-Direktiven auch weggelassen werden, wenn sie den gleichen Namen haben, aber der ursprüngliche Typ darf nicht gewechselt werden. Wenn kein Verbindetyp angegeben wurde, setzt RASM86 den Standardtyp PUBLIC ein.

PUBLIC bedeutet, daß der Linker Segmente miteinander verbinden kann, die gleiche Namen haben. Alle Segmente mit dem Verbindetyp PUBLIC werden miteinander in der Reihenfolge ihres Auftretens durch den Linker verkettet. Durch den Zuordnungstyp können sich eventuell zwischen den einzelnen Segmenten Lücken ergeben.

COMMON bedeutet, daß das Segment gleiche Speicherplätze mit anderen Segmenten gleichen Namens teilt. Offsets sind innerhalb eines COMMON-Segments absolut, wenn sich das Segment nicht in einer GROUP-Direktive befindet (siehe Abschnitt 3.3.).

Der STACK-Verbindetyp ist ähnlich dem von PUBLIC, wo die Speicherzuweisung für STACK-Segmente die Summe der STACK-Segmente jedes Moduls ist. Aber stattdessen überlagert der Linker die STACK-Segmente von geketteten Segmenten gleichen Namens zu höheren Speicheradressen hin, weil Stacks abwärts von höheren zu niedrigeren Adressen wachsen, wenn das Programm läuft.

LOCAL bedeutet, daß das Segment lokal im übersetzten Programm ist. Der Linker verbindet es nicht mit anderen Segmenten.
Für ein absolutes Segment bestimmt RASM86 die Ladeadresse des Segments während der Übersetzung. Dessen Position wird dann durch den Linker zum Ladezeitpunkt bestimmt.

3.2.4. Klassenname (class-name)

Der Klassenname kann ein zulässiger RASM86-Bezeichner sein. Mit dem Klassennamen können gekennzeichnete Segmente im gleichen Bereich in einer vom Linker erzeugten CMD-Datei eingeordnet werden. Wenn nicht durch eine GROUP-Direktive oder ein bestimmtes Linker-Kommando eine Umstellung veranlaßt wird, ordnet der Linker diese Segmente in die CMD-Datei ein, wie es folgende Tabelle zeigt:

Tabelle 10: Standard-Klassennamen für Segmente

Segment-Direktive	Standard-Klassennamen	CMD-Bereich
CSEG	CODE	CODE
DSEG	DATA	DATA
ESEG	EXTRA	EXTRA
SSEG	STACK	STACK

3.3. GROUP-Direktive

```
group-name GROUP segment-name-1, segment-name-2, ...
```

Die GROUP-Direktive veranlaßt RASM86, die aufgeführten Segmente in einer Gruppe miteinander zu verbinden. Die Länge kann maximal 64K betragen. Die Offsets innerhalb des Segments einer Gruppe sind relativ zum Anfang der Gruppe, ähnlich denen zum Segmentanfang.
Die Reihenfolge der Segmentnamen in der Direktive ist die Reihenfolge, in welcher der Linker die Segmente in die CMD-Datei einordnet.

3.4. ORG-Direktive

```
ORG  numeric-expression
```

Die ORG-Direktive setzt den Offset des Speicherplatzzählers (location-counter) im aktuellen Segment auf den angegebenen Wert im numerischen Ausdruck (numeric-expression). Alle Elemente des Ausdruckes müssen vor der Benutzung der ORG-Direktive definiert werden. Der numerische Ausdruck muß eine absolute Zahl ergeben. Er ist relativ zum Speicherplatzzähler innerhalb des Segments zum Ladezeitpunkt. Folglich zeigt der numerische Ausdruck den aktuellen Offset innerhalb des Segments an, wenn die ORG-Direktive in einem Segment benutzt wird, das der Linker nicht mit anderen Segmenten verbindet, wie bei LOCAL- oder absoluten Segmenten.
Wenn aber das Segment zur Link-Zeit mit anderen Segmenten verbunden wird, wie bei PUBLIC-Segmenten, dann ist der numerische Ausdruck kein absoluter Offset. Er ist relativ zum Segmentanfang des übersetzten Programms.
Die Benutzung von Gruppen kann einen effektiveren Code ergeben, weil eine Anzahl von Segmenten von einem einfachen Segmentregister adressiert werden kann, ohne den Inhalt des Segmentregisters ändern zu müssen.

3.5. END-Direktive

```
END  [start-label]
```

Die END-Direktive markiert das Ende einer Quelldatei. RASM86 ignoriert weitere nachfolgende Zeilen. 'Die END-Direktive ist wahlweise angebbar. Wenn sie fehlt, übersetzt RASM86 die Quelldatei bis er ein Dateiendezeichen (EOF) findet.
Die wahlweise angebbare Startmarke (start-label) erfüllt zwei Funktionen. Erstens wird der aktuelle Modul als Hauptprogramm festgelegt. Wenn der Linker die Moduln miteinander verbindet, kann nur eines das Hauptprogramm sein. Zweitens gibt die Startmarke auch die Startadresse nach dem Laden an. Fehlt die Startmarke, beginnt das Programm am ersten CSEG der verbundenen Datei.

3.6. NAME-Direktive

```
NAME 'module-name'
```

Die NAME-Direktive weist dem Objektmodul, der von RASM86 erzeugt wird, einen Namen zu. Der Modulname (module-name) kann ein zulässiger Bezeichner sein. Wird keine NAME-Direktive angegeben, übernimmt RASM86 den Namen der Quelldatei. Sowohl der Linker als auch der Bibliothekar benutzen den Objektnamen um den Objektmodul zu identifizieren.

3.7. PUBLIC-Direktive

```
PUBLIC  name [, name, ...]
```

Die PUBLIC-Direktive legt fest, daß mit PUBLIC definierte Namen zu anderen miteinander verbundenen Programmen Bezug nehmen können. Jeder Name kann eine Marke, Variable oder Zahl sein, die im übersetzten Programm definiert ist.

3.8. EXTRN-Direktive

```
EXTRN external-id [, external-id, ...]
```

Mit der EXTRN-Direktive ermöglicht RASM86, daß innerhalb des übersetzten Programms zu jedem externen Symbol (external-symbol) zugegriffen werden kann, das in einem anderen Programm definiert ist. Das externe Symbol besteht aus zwei Teilen: einem Symbol und einem Typ. Das Symbol kann eine Variable, Marke oder Zahl sein. Der Typ ist einer der folgenden:

- Variable: BYTE, WORD, DWORD
- Marke: NEAR, FAR
- Zahl: ABS

Beispiel:

```
EXTRN  FCB:BYTE, BUFFER:WORD, INIT:FAR, MAX:ABS
```

RASM86 bestimmt die Segment-Attribute der externen Variablen und Marken des Segments, das die EXTRN-Direktive enthält. Die EXTRN-Direktive für ein gegebenes Symbol muß in dem Segment erscheinen, in welchem das Symbol steht, das in einem anderen Modul definiert ist.

3.9. IF, ELSE und ENDIF-Direktive

```
IF          numeric-expression
            source-line 1
            source-line 2
                        .
                        .
                        .
            source-line n

[ELSE]
            alternate-source-line 1
            alternate-source-line 2
                        .
                        .
                        .
            alternate-source-line n
ENDIF
```

Die IF- und ENDIF-Direktiven ermöglichen es unter bestimmten Bedingungen, Quellzeilen vom Übersetzungsprozeß ein- oder auszuschließen. Die wahlweise angebbare ELSE-Direktive ermöglicht es, eine alternative Anzahl von Quellzeilen anzugeben. Diese Bedingungsdirektiven können benutzt werden, um verschiedene Versionen eines einzigen Quellprogramms zu übersetzen. Die IF-Direktive kann bis zu fünf Stufen tief verschachtelt werden.
Wenn RASM86 eine IF-Direktive findet, wertet er den numerischen Ausdruck nach IF aus. Alle Elemente des numerischen Ausdrucks müssen vor Auswertung der IF-Direktive definiert sein. Wenn der Wert des Ausdrucks nicht Null ist, übernimmt RASM86 die Quellzeilen source-line 1 bis source-line n in den Übersetzungsvorgang. Ist der Wert Null, dann werden alle Zeilen gedruckt, aber nicht übersetzt.
Wenn der Wert des Ausdrucks Null ergibt und sich eine ELSE-Direktive zwischen IF und ENDIF befindet, übersetzt RASM86 die alternativen Quellzeilen.

3.10. EQU-Direktive

```
symbol  EQU     numeric-expression
symbol  EQU     address-expression
symbol  EQU     register
symbol  EQU     instruction-mnemonic
```

Die EQU-Direktive weist den vom Nutzer definierten Symbolen Werte und Attribute zu. Es darf kein Doppelpunkt hinter dem Symbol stehen. Ist einmal ein Symbol definiert, kann es nicht mit einer nachfolgenden EQU- oder anderen Direktive umdefiniert werden. Es müssen alle Elemente vor Benutzung der EQU-Direktive definiert sein, wenn sie in dem numerischen oder Adreßausdruck angegeben werden.
Die erste Form weist dem Symbol einen numerischen Wert zu, die zweite eine Speicheradresse. Die dritte Form weist einem Register einen neuen Namen zu, und die vierte Form definiert einen neuen Befehl, bzw. Unterbefehl.

Beispiel:

```
  0005              FIVE    EQU     2*2+1
  0033              NEXT    EQU     BUFFER
```

```
0001                COUNTER EQU     CX
                    MOVVV   EQU     MOV
                            .
                            .
                            .
005D 8BC3                   MOVVV   AX,BX
```

3.11. DB-Direktive

[symbol] DB {numeric-expression | string-constant},...

Die DB-Direktive definiert und initialisiert Speicherbereiche im Byteformat. RASM86 verschlüsselt numerische Ausdrücke in 8-bit-Werte und speichert sie nacheinander in der Objektdatei ab. Zeichenkettenkonstanten werden in der Objektdatei entsprechend den in Abschnitt 2.4.2. definierten Regeln abgespeichert. Innerhalb der Zeichenketten erfolgt keine Umwandlung von Klein- in Großbuchstaben.
Die DB-Direktive ist die einzige RASM86-Anweisung, die Zeichenketten mit mehr als zwei Bytes akzeptiert. Es können mehrere Ausdrücke oder Konstanten, die durch Komma getrennt sind, der Definition hinzugefügt werden, wenn sie nicht die physische Zeilenlänge überschreiten. Um auf das definierte Datenfeld zuzugreifen, wird ein wahlweises Symbol benutzt.
Das Symbol hat vier Attribute:
- Die Segment- und Offset-Attribute bestimmen die Speicherbezüge des Symbols.
- Das Typattribut gibt die einzelnen Bytes an.
- Das Längenattribut gibt an, wieviel Bytes reserviert sind.

Beispiel:

```
005F 544350207379   TEXT    DB      'SCP system',0
     7374656D00
006A E1             AA      DB      'a'+80H
006B 0102030405     X       DB      1,2,3,4,5
                            .
                            .
                            .
0071 B90B00                 MOV     CX,LENGTH TEXT
```

3.12. DW-Direktive

[symbol] DW {numeric-expression | strin-constant},...

Die DW-Direktive initialisiert Zwei-Byte-Worte im Speicher. Sie initialisiert Speicherplatz genau wie die DB-Direktive, nur daß jeder numerische Ausdruck oder jede Zeichenkettenkonstante zwei Bytes im Speicher belegen, in dem das niedere Byte zuerst steht. Konstantenfolgen mit mehr als zwei Bytes sind unzulässig.

Beispiel:

```
0074 0000           CNTR    DW      0
0076 63C166C169C1   JMPTAB  DW      SUBR1,SUBR2,SUBR3
007C 010002000300           DW      1,2,3,4,5,6
     040005000600
```

3.13. DD-Direktive

[symbol] DD address-expression [, address-expression ...]

Die DD-Direktive initialisiert vier Bytes im Speicher. Das Offset-Attribut des Adreßausdrucks wird in den beiden niederen Bytes abgespeichert, das Segment-Attribut in den beiden höheren Bytes. DD folgt der gleichen Prozedur wie DB.

Beispiel:

```
                                  CSEG
0000 6CC100006FC1  LONG-JMPTAB DD   ROUT1,ROUT2
     0000
0008 72C1000075C1              DD   ROUT3,ROUT4
     0000
```

3.14. RS-Direktive

[symbol] RS numeric-expression

Die RS-Direktive reserviert Speicherplatz, aber initialisiert ihn nicht. Der numerische Ausdruck gibt die Bytezahl an, die zu reservieren ist. Die RS-Direktive gibt für das wahlweise anzugebende Symbol kein Byteattribut an.

Beispiel:

```
0010                BUF     RS      80
0060                        RS      4000H
4060                        RS      1
```

3.15. RB-Direktive

[symbol] RB numeric-expression

Die RB-Direktive reserviert Bytes im Speicher ohne Initialisierung. Die RB-Direktive ist identisch zur RS-Direktive, mit dem Unterschied, daß sie ein Byteattribut liefert.

3.16. RW-Direktive

[symbol] RW numeric-expression

Die RW-Direktive reserviert Zwei-Byte-Worte im Speicher ohne Initialisierung. Der numerische Ausdruck gibt die Anzahl der zu reservierenden Worte an.

Beispiel:

```
4061                BUFF    RW      128
4161                        RW      4000H
C161                        RW      1
```

3.17. RD-Direktive

```
[symbol] RD numeric-expression
```

Die RD-Direktive reserviert ein Doppelwort (vier Bytes) im Speicher ohne Initialisierung.

Beispiel:

```
C163                DW      RD      4
C173                        RD      1
```

3.18. EJECT-Direktive

```
EJECT
```

Die EJECT-Direktive führt zu einem Seitenvorschub während des Druckvorganges. Die EJECT-Direktive selbst wird auf der ersten Zeile der nächsten Seite ausgegeben.

3.19. NOIFLIST- und IFLIST-Direktive

```
NOIFLIST
IFLIST
```

Die NOIFLIST-Direktive unterdrückt die Ausgabe des Inhalts des während der bedingten Assemblierung zu übersetzenden Blockes, wenn keine Übersetzung angewiesen wurde. Die IFLIST-Direktive weist die Ausgabe dieses Blockes an.

3.20. NOLIST- und LIST-Direktive

```
NOLIST
LIST
```

Die NOLIST-Direktive unterdrückt die Ausgabe der folgenden Zeilen. Die LIST-Direktive bewirkt die Fortsetzung der Ausgabe.

3.21. PAGESIZE-Direktive

```
PAGESIZE numeric-expression
```

Die PAGESIZE-Direktive definiert die Zahl der Zeilen pro Druckseite. Der Standardwert ist 66 Zeilen pro Seite.

3.22. PAGEWIDTH-Direktive

```
PAGEWIDTH numeric-expression
```

Die PAGEWIDTH-Direktive definiert die Zahl der zu druckenden Spalten quer über die Seite der Ausgabedatei. Der Standardwert ist 120, wenn die Ausgabe nicht über die Konsole erfolgt, andernfalls beträgt er 79.

3.23. SIMFORM-Direktive

SIMFORM

Die SIMFORM-Direktive ersetzt ein FF-Zeichen (form-feed) in der Listendatei durch eine bestimmte Anzahl von LF (line-feed). Diese Direktive wird dann benutzt, wenn die Ausgabe auf einem Drucker erfolgt, der FF-Zeichen nicht versteht.

3.24. TITLE-Direktive

TITLE string-constant

RASM86 druckt die Zeichenkette, die mit der TITLE-Direktive definiert wird, als Kopfzeile auf jede Listenseite in der Listendatei. Die Zeichenkette darf bis zu 30 Zeichen lang sein.

Beispiel: TITLE 'SCP 1700 SYSTEM'

3.25. INCLUDE-Direktive

INCLUDE filespec

Die INCLUDE-Direktive schließt eine andere RASM86-Quelldatei in den Quelltext ein.

Beispiel: INCLUDE EQUALS.A86

Die INCLUDE-Direktive kann benutzt werden, wenn das Quellprogramm groß ist und aus mehreren Dateien besteht. INCLUDE-Direktiven können nicht geschachtelt werden, d. h. eine Quelldatei, die über INCLUDE angefordert wurde, darf keine andere INCLUDE-Direktive enthalten.
Wenn die in der INCLUDE-Direktive gerufene Datei keinen Dateityp hat, übernimmt RASM86 den Dateityp A86. Wenn kein Laufwerk mit der Datei angegeben ist, greift RASM86 zum Laufwerk der Quelldatei zu.

4. RASM86-Befehlssatz

4.1. Einführung

Der RASM86-Befehlssatz enthält alle Maschinenbefehle. Die allgemeine Syntax für Befehlsanweisungen wurde im Abschnitt 2.8. erläutert. Die folgenden Abschnitte behandeln die spezifische Syntax und die für jeden Befehl erforderlichen Operanden, ohne auf Marken und Kommentare einzugehen. Die Befehle werden in Tabellen leicht verständlich dargestellt.
Die Tabellen mit den Befehlsdefinitionen zeigen die RASM86-Befehle als Kombination von Mnemoniks und Operanden. Das Mnemonik ist eine symbolische Darstellung des Befehls und seine Operanden sind die geforderten Parameter. Befehle können ohne oder mit einem oder zwei Operanden versehen sein. Wenn zwei Operanden vorhanden sind, dann ist der linke Operand der Zieloperand (destination). Beide Operanden sind durch Komma getrennt.
Die Tabellen der RASM86-Befehle sind in funktionelle Gruppen unterteilt. Innerhalb der Tabellen sind die Befehle alphabetisch geordnet. Die Tabelle 11 zeigt die Symbole, die in den Tabellen zur Definition der Operanden benutzt werden.

Tabelle 11: Symbole für Operandentypen

Symbol	Operandentyp
numb	beliebiger numerischer Ausdruck
numb8	beliebiger numerischer Ausdruck, der einen 8-bit-Wert ergibt
acc	Akkumulatorregister, AX oder AL
reg	allgemeines Register, kein Segmentregister
reg16	allgemeines 16-bit-Register, kein Segmentregister
segreg	beliebiges Segmentregister: CS, DS, SS oder ES
mem	beliebiger Adressausdruck, mit oder ohne Basis- und/oder Indexadressierungsmodi, wie: VARIABLE VARIABLE[BX+SI] VARIABLE+3 [BX] VARIABLE[BX] [BP+DI] VARIABLE[SI]
simpmem	beliebiger Adreßausdruck ohne Basis- und Indexadressierungsmodi, wie: VARIABLE VARIABLE+4
mem|reg	beliebiger Ausdruck, symbolisiert durch mem|reg
mem|reg16	beliebiger Ausdruck, symbolisiert durch mem|reg, aber Register/Speicher müssen 16 bit sein
label	beliebiger Adreßausdruck, der eine Marke ergibt
lab8	beliebige Marke, bis ±128 Bytes vom Befehl entfernt

Die ZVE besitzt 9 Einzelbit-Flagregister, die den Status der ZVE anzeigen. Der Anwender kann zu diesen Registern nicht direkt zugreifen, aber er kann sie im Ergebnis einer Operation austesten. Die Wirkung der Befehle auf die Flagregister wird in den Tabellen der Befehlsdefinitionen beschrieben. Dabei werden die in der Tabelle 12 dargestellten Symbole benutzt.

Tabelle 12: Flagregistersymbole

Symbol	Bedeutung
AF	Auxiliary-Carry-Flag
CF	Carry-Flag
DF	Direction-Flag
IF	Interrupt-Enable-Flag
OF	Overflow-Flag
PF	Parity-Flag
SF	Sign-Flag
TF	Trap-Flag
ZF	Zero-Flag

4.2. Datenübertragungsbefehle

Es gibt vier Klassen von Operationen zur Datenübertragung:

- allgemeine Operationen
- akkumulatorspezifische Operationen
- Adreßoperationen
- Flagoperationen

Nur SAHF und POPF beeinflussen die Flags. In Tabelle 13 ist zu beachten, daß bei acc = AL ein Byte, aber bei acc = AX ein Wort übertragen wird.

Tabelle 13: Datenübertragungsbefehle

Syntax	Ergebnis
IN acc,numb8	Datenübertragung vom Eingabekanal zum Akkumulator, angegeben durch numb8 (0-255)
IN acc,DX	Datenübertragung vom Eingabekanal zum Akkumulator, angegeben durch DX (0-FFFFH)
LAHF	Übertragung der Flags zum AH-Register
LDS reg16,mem	Übertragung des Segmentteils der Speicheradresse (DWORD-Variable) zum DS-Segmentregister, Übertragung des Offset-Teils zu einem allgemeinen Register (16 bit)
LEA reg16,mem	Übertragung des Offsets einer Speicheradresse zu einem 16-bit-Register
LES reg16,mem	Übertragung des Segmentteils der Speicheradresse zum ES-Segmentregister, Übertragung des Offset-Teils zu einem allgemeinen Register (16 bit)
MOV reg,mem|reg	überträgt Speicher oder Register zum Register
MOV mem|reg,reg	überträgt Register zum Speicher oder Register
MOV mem|reg,numb	überträgt Direktdaten (immediate-data) zum Speicher oder Register
MOV segreg,mem|reg16	überträgt Speicher oder Register zum Segmentregister
MOV mem|reg16,segreg	überträgt Segmentregister zum Speicher oder Register
OUT numb8,acc	überträgt Daten vom Akkumulator zum Ausgabekanal (0-255), angegeben durch numb8
OUT DX,acc	überträgt Daten vom Akkumulator zum Ausgabkanal (0-0FFFFH), angegeben durch DX
POP mem|reg16	überträgt oberes Stackelement zum Speicher oder Register

Tabelle 13: (Fortsetzung)

Syntax	Ergebnis
POP segreg	überträgt oberes Stackelement zum Segmentregister, wobei das CS-Segment nicht erlaubt ist
POPF	überträgt oberes Stackelement zu den Flags
PUSH mem|reg16	legt Speicher oder Register als oberes Stackelement ab
PUSH segreg	legt Segmentregister als oberes Stackelement ab
PUSHF	legt Flags als oberes Stackelement ab
SAHF	überträgt das AH-Register zu den Flags
XCHG reg,mem|reg	wechselt Register und Speicher oder Register
XCHG mem|reg,reg	wechselt Speicher oder Register und Register
XLAT mem|reg	führt eine Tabellenübertragung aus. Die Anfangsadresse der Tabelle muß in BX stehen. AL wird durch den AL-Offset von BX ersetzt.

4.3. Arithmetische, logische und Verschiebebefehle

Die ZVE führt die vier mathematischen Grundoperationen auf verschiedenen Wegen aus. Sie unterstützt sowohl 8- als auch 16-bit-Operationen und führt außerdem die Arithmetik mit und ohne Berücksichtigung des Vorzeichens aus.
Sechs der neun Flagbits werden im Ergebnis der meisten arithmetischen Operationen gelöscht oder gesetzt.
Tabelle 14 faßt die Wirkungen der arithmetischen Befehle auf die Flagbits zusammen.
Tabelle 15 definiert die arithmetischen Befehle und Tabelle 16 die logischen und Verschiebebefehle.

Tabelle 14: Wirkung der Arithmetikbefehle auf die Flags

Flagbit	Ergebnis
CF	wird gesetzt, falls die Operation einen Übertrag (Addition) oder ein Borgen (Subtraktion) in das höchste Bit des Ergebnisses bewirkt; andernfalls ist CF gelöscht
AF	wird gesetzt, falls die Operation einen Übertrag (Addition) oder ein Borgen (Subtraktion) in das niedrigste Bit des Ergebnisses bewirkt; andernfalls ist AF gelöscht
ZF	wird gesetzt, wenn das Ergebnis der Operation Null ist; andernfalls bleibt ZF gelöscht
SF	wird gesetzt, wenn das Ergebnis negativ ist
PF	wird gesetzt, falls die Modulo-2-Summe der niedrigsten 8 bit des Ergebnisses der Operation Null ist (gerade Parität); andernfalls wird PF gelöscht (ungerade Parität)
OF	wird gesetzt, wenn das Ergebnis der Operation einen Überlauf ergibt, wobei die Größe des Ergebnisses die Speicherkapazität der Zieladresse überschreitet

Tabelle 15: Arithmetische Befehle

Syntax	Ergebnis
AAA	Anpassung ungepackter BCD (KOI7) für Addition - AL-Anpassung (BCD = binär codierte Dezimalzahlen)
AAD	Anpassung ungepackter BCD (KOI7) für Division - AL-Anpassung
AAM	Anpassung ungepackter BCD (KOI7) für Multiplikation - AX-Anpassung
AAS	Anpassung ungepackter BCD (KOI7) für Subtraktion - AL-Anpassung
ADC reg,mem|reg	Addition mit Übertrag (carry) Speicher oder Register zu Register
ADC mem|reg,reg	Addition mit Übertrag (carry) Register zu Speicher oder Register
ADC mem|reg,numb	Addition mit Übertrag (carry) von Direktdaten zu Speicher oder Register
ADD reg,mem|reg	Addition Speicher oder Register zu Register
ADD mem|reg,reg	Addition Register zu Speicher oder Register
ADD mem|reg,numb	Addition von Direktdaten zu Speicher oder Register
CBW	Konvertierung eines Bytes in AL zu einem Wort in AX durch Vorzeichenexpansion
CMP reg,mem|reg	Vergleich Register mit Speicher oder Register
CMP mem|reg,reg	Vergleich Speicher oder Register mit Register
CMP mem|reg,numb	Vergleich Direktdaten mit Speicher oder Register
CWD	Konvertierung eines Wortes in AX zu einem Doppelwort in DX/AX durch Vorzeichenexpansion
DAA	Dezimalanpassung für Addition (AL)
DAS	Dezimalanpassung für Subtraktion (AL)

Tabelle 15: (Fortsetzung)

Syntax	Ergebnis
DEC mem|reg	Subtraktion von 1 vom Speicher oder Register (Dekrement)
DIV mem|reg	Division (ohne Vorzeichen) Akkumulator (AX oder AL) durch Speicher oder Register. Bei Byteergebnis enthält AL = Quotient und AH = Rest. Bei Wortergebnis ist: AX = Quotient und DX = Rest.
IDIV mem|reg	Division (mit Vorzeichen) Akkumulator (AX oder AL) durch Speicher oder Register. Quotient und Rest werden wie bei DIV abgespeichert.
IMUL mem|reg	Multiplikation (mit Vorzeichen) Speicher oder Register mit Akkumulator (AX oder AL). Falls Byteergebnis, dann Ergebnis in AH und AL. Falls Wortergebnis, dann Ergebnis in DX und AX.
INC mem|reg	Addition von 1 zu Speicher oder Register (Increment)
MUL mem|reg	Multiplikation (ohne Vorzeichen) Speicher oder Register mit Akkumulator (AX oder AL). Das Ergebnis wird wie bei IMUL abgespeichert.
NEG mem|reg	Zweierkomplement von Speicher oder Register
SBB reg,mem|reg	Subtraktion (mit Borgen) Speicher oder Register von Register
SBB mem|reg,reg	Subtraktion (mit Borgen) Register von Speicher oder Register
SBB mem|reg,numb	Subtraktion (mit Borgen) Direktdaten von Speicher oder Register
SUB reg,mem|reg	Subtraktion Speicher oder Register von Register
SUB mem|reg,reg	Subtraktion Register von Speicher oder Register
SUB mem|reg,numb	Subtraktion Direktdaten von Speicher oder Register

Tabelle 16: Logische und Verschiebebefehle

Syntax	Ergebnis
AND reg,mem|reg	bitweises logisches UND von Register und Speicher oder Register
AND mem|reg,reg	bitweises logisches UND von Speicher oder Register und Register
AND mem|reg,numb	bitweises logisches UND von Speicher oder Register und Direktdaten
NOT mem|reg	Einerkomplement von Speicher oder Register
OR reg,mem|reg	bitweises logisches ODER von Register und Speicher oder Register
OR mem|reg,reg	bitweises logisches ODER von Speicher oder Register und Register
OR mem|reg,numb	bitweises logisches ODER von Speicher oder Register und Direktdaten
RCL mem|reg,1	zyklische Verschiebung Speicher oder Register um 1 bit nach links durch das Übertragflag (C-Flag)
RCL mem|reg,CL	zyklische Verschiebung Speicher oder Register nach links durch C-Flag; Verschiebezahl in CL-Register
RCR mem|reg,1	zyklische Rechtsverschiebung um 1 bit durch C-Flag
RCR mem|reg,CL	zyklische Rechtsverschiebung Speicher oder Register durch C-Flag; Verschiebezahl in CL-Register
ROL mem|reg,1	zyklische Linksverschiebung Speicher oder Register um 1 bit
ROL mem|reg,CL	zyklische Linksverschiebung Speicher oder Register; Verschiebezahl in CL-Register
ROR mem|reg,1	zyklische Rechtsverschiebung Speicher oder Register um 1 bit
ROR mem|reg,CL	zyklische Rechtsverschiebung Speicher oder Register; Verschiebezahl in CL-Register

Tabelle 16: (Fortsetzung)

Syntax	Ergebnis
SAL mem|reg,1	Linksverschiebung Speicher oder Register um 1 bit; in niederes Bit wird Null aufgefüllt
SAL mem|reg,CL	Linksverschiebung Speicher oder Register; Verschiebezahl in CL-Register; in niedere Bits werden Nullen aufgefüllt
SAR mem|reg,1	Rechtsverschiebung Speicher oder Register um 1 bit und Nachschieben des ursprünglich höchsten Bits
SAR mem|reg,CL	Rechtsverschiebung von Speicher oder Register; Verschiebezahl in CL-Register; Nachschieben des ursprünglich höchsten Bits
SHL mem|reg,1	Linksverschiebung Speicher oder Register um 1 bit; Nachschieben von Null in niederes Bit (SHL=SAL)
SHL mem|reg,CL	Linksverschiebung Speicher oder Register; Verschiebezahl in CL-Register; Nachschieben von Nullen in die niederen Bits (SHL=SAL)
SHR mem|reg,1	Rechtsverschiebung Speicher oder Register um 1 bit und Nachschieben von Null ins höchste Bit
SHR mem|reg,CL	Rechtsverschiebung Speicher oder Register; Verschiebezahl in CL-Register und Nachschieben von Nullen in die höheren Bits
TEST reg,mem|reg	bitweises logisches UND von Register und Speicher oder Register; setzt die Flags, aber verändert Zieloperanden nicht
TEST mem|reg,reg	bitweises logisches UND von Speicher oder Register und Register; setzt die Flags, aber verändert Zieloperanden nicht
TEST mem|reg,numb	bitweises logisches UND von Speicher oder Register und Direktdaten; setzt die Flags, aber verändert Zieloperanden nicht

Tabelle 16: (Fortsetzung)

Syntax	Ergebnis
XOR reg,mem|reg	bitweises logisches exklusives ODER von Register und Speicher oder Register
XOR mem|reg,reg	bitweises logisches exklusives ODER von Speicher oder Register und Register
XOR mem|reg,numb	bitweises logisches exklusives ODER von Speicher oder Register und Direktdaten

4.4. Zeichenkettenbefehle

Zeichenkettenbefehle (string-instructions) haben keinen, einen oder zwei Operanden. Die Operanden geben nur den Operandentyp an, ob es eine Byte- oder Wortoperation ist. Wenn zwei Operanden angegeben sind, wird der Quelloperand durch das SI-Register und der Zieloperand durch das DI-Register adressiert. Die DI- und SI-Register werden immer für die Adressierung benutzt. Es ist zu beachten, daß sich bei Zeichenkettenbefehlen der durch DI adressierte Zieloperand immer im Extra-Segment (ES) befinden muß.
Der Quelloperand wird im allgemeinen durch das DS-Register adressiert. Jedoch kann durch die Benutzung eines Segment-Override-Präfix ein anderes Register festgelegt werden.

Beispiel:

```
MOVS  WORD PTR[DI],CS:WORD PTR[SI]
```

Tabelle 17: Zeichenkettenbefehle

Syntax	Ergebnis
CMPS mem|reg,mem|reg	Subtraktion der Quelle vom Ziel und Setzen der Flags; Operanden werden nicht verändert
CMPSB	CMPS mit Byteoperand
CMPSW	CMPS mit Wortoperand
LODS mem|reg	Übertragung eines Bytes oder Wortes vom Quelloperanden zum Akkumulator
LODSB	LODS mit Byteoperand
LODSW	LODS mit Wortoperand
MOVS mem|reg,mem|reg	Übertragung eines Bytes oder Wortes von der Quelle zum Ziel
MOVSB	MOVS mit Byteoperand
MOVSW	MOVS mit Wortoperand
SCAS mem|reg	Subtraktion des Zieloperanden vom Akkumulator (AX oder AL); Setzen der Flags, aber Rückkehr ohne Ergebnis
SCASB	SCAS mit Byteoperand
SCASW	SCAS mit Wortoperand
STOS mem|reg	Übertragung eines Bytes oder Wortes vom Akkumulator zum Zieloperanden
STOSB	STOS mit Byteoperand
STOSW	STOS mit Wortoperand

Tabelle 18 definiert die Präfixe für Zeichenkettenbefehle. Ein Präfix wiederholt seinen Zeichenkettenbefehl entsprechend der Anzahl im CX-Register, dessen Inhalt nach jeder Iteration um 1 erniedrigt wird. Präfixmnemoniks stehen vor dem Zeichenkettenbefehlsmnemoniks der Anweisungszeile.

Tabelle 18: Präfixbefehle

Syntax	Ergebnis
REP	Wiederholung bis CX-Register Null ist
REPE	Wiederholung bis CX-Register Null und das Z-Flag ungleich Null ist
REPNE	Wiederholung bis CX-Register Null und das Z-Flag Null ist
REPNZ	siehe REPNE
REPZ	siehe REPE

4.5. Steuerübertragungsbefehle

Es gibt vier Klassen von Steuerübertragungsbefehlen:

- Rufe, Sprünge und Rücksprünge
- bedingte Sprünge
- Wiederholungssteuerung
- Unterbrechungen (interrupts)

Alle Steuerübertragungsbefehle veranlassen die Programmfortsetzung ab einem bestimmten neuen Platz im Speicher, möglicherweise in einem neuen Code-Segment. Die Übertragung der Steuerung kann absolut oder in Abhängigkeit einer bestimmten Bedingung erfolgen. Tabelle 19 definiert die Steuerübertragungsbefehle. In den Definitionen bedingter Sprünge, vorwärts und rückwärts, handelt es sich um vorzeichenlose Werte. Größer als und kleiner bezieht sich auf Werte mit Vorzeichen.

Tabelle 19: Steuerübertragungsbefehle

Syntax		Ergebnis
CALL	label	Ablegen der Offset-Adresse des nächsten Befehls auf dem Stack und Sprung zur Zielmarke (label).
CALL	mem|reg16	Ablegen der Offset-Adresse des nächsten Befehls auf dem Stack und Sprung zum Speicherplatz, der durch den Inhalt des Speicherplatzes bzw. Registers angezeigt ist.
CALLF	label	Ablegen des CS-Registerinhaltes auf dem Stack; Ablegen der Offset-Adresse des nächsten Befehls auf dem Stack (nach CS); Sprung zur Zielmarke
CALLF	mem	Ablegen des CS-Registerinhaltes auf dem Stack; Ablegen der Offset-Adresse des nächsten Befehls auf dem Stack und Sprung zum Speicherplatz, der durch den Inhalt des Doppelwortes im Speicher angegeben ist.
INT	numb8	Ablegen des Flag-Registers (wie PUSHF) auf dem Stack; Löschen der TF- und IF-Flags und Übertragung der Steuerung mit einem Indirektruf über einen der 256 Unterbrechungsvektorelemente (benutzt drei Ebenen des Stacks).
INTO		Falls das OF-Flag (Overflow-Flag) gesetzt ist, wird das Flagregister (wie PUSHF) auf dem Stack abgelegt. TF und IF werden gelöscht und die Steuerung mit einem Indirektruf über Unterbrechungsvektorelement 4 (Platz 10H) übertragen. Falls OF gelöscht ist, wird keine Operation ausgeführt.
IRET		Übertragung der Steuerung zur Rückkehradresse, die durch die vorhergehende Unterbrechungsoperation gerettet wurde. Die geretteten Flagregister werden zurückgespeichert, ebenso CS und IP (gibt drei Ebenen des Stacks frei).

Tabelle 19: (Fortsetzung)

Syntax		Ergebnis
JA	lab8	Sprung, falls "nicht darunter oder gleich" oder "darüber" ((CF oder ZF) = 0)
JAE	lab8	Sprung, falls "nicht darunter" oder "darüber oder gleich" (CF = 0)
JB	lab8	Sprung, falls "darunter" oder "nicht darüber oder gleich" (CF = 1)
JBE	lab8	Sprung, falls "darunter oder gleich" oder "nicht darüber" ((CF oder ZF) = 1)
JC	lab8	siehe JB
JCXZ	lab8	Sprung zur Zielmarke, wenn CX-Register gleich Null
JE	lab8	Sprung, falls "gleich" oder "Null" (ZF = 1)
JG	lab8	Sprung, falls "nicht kleiner oder gleich" oder "größer" (((SF xor OF) or ZF) = 0)
JGE	lab8	Sprung, falls "nicht kleiner" oder "größer oder gleich" ((SF xor CF) = 0)
JL	lab8	Sprung, falls "kleiner" oder "nicht größer oder gleich" ((SF xor OF) = 1)
JLE	lab8	Sprung, falls "kleiner oder gleich" oder "nicht größer" (((SF xor OF) or ZF) = 1)
JMP	label	Sprung zur Zielmarke
JMP	mem|reg16	Sprung zum Speicherplatz, der durch den Inhalt von Speicher oder Register angegeben ist
JMPF	label	Sprung zur Zielmarke, möglicherweise in einem anderen Code-Segment
JMPS	lab8	Sprung zur Zielmarke innerhalb einer Distanz von +/-128 Bytes von dem Befehl

Tabelle 19: (Fortsetzung)

Syntax		Ergebnis
JNA	lab8	siehe JBE
JNAE	lab8	siehe JB
JNB	lab8	siehe JAE
JNBE	lab8	siehe JA
JNC	lab8	siehe JAE
JNE	lab8	Sprung, falls "nicht gleich" oder "nicht Null" (ZF = 0)
JNG	lab8	siehe JLE
JNGE	lab8	siehe JL
JNL	lab8	siehe JGE
JNLE	lab8	siehe JG
JNO	lab8	Sprung, falls "kein Überlauf" (OF = 0)
JNP	lab8	Sprung, falls "keine Parität" oder "ungerade Parität" (PF = 0)
JNS	lab8	Sprung, falls "kein Vorzeichen" (SF = 0)
JNZ	lab8	siehe JNE
JO	lab8	Sprung, falls "Überlauf" (OF = 1)
JP	lab8	Sprung, falls "Parität" oder "gerade Parität" (PF = 1)
JPE	lab8	siehe JP
JPO	lab8	siehe JNP
JS	lab8	Sprung, falls "Vorzeichen" (SF = 1)
JZ	lab8	siehe JE
LOOP	lab8	Dekrement von CX um 1 und Sprung zur Zielmarke, wenn CX nicht Null
LOOPE	lab8	Dekrement von CX um 1 und Sprung zur Zielmarke, falls CX nicht Null und das ZF-Flag gesetzt ist - "Schleife, solange Null" oder "Schleife, solange gleich"

Tabelle 19: (Fortsetzung)

Syntax	Ergebnis
LOOPNE lab8	Dekrement von CX um 1 und Sprung zur Zielmarke, falls CX ungleich Null und ZF = 0 - "Schleife, solange ungleich Null" oder "Schleife, solange nicht gleich"
LOOPNZ lab8	siehe LOOPNE
LOOPZ lab8	siehe LOOPE
RET	Rücksprung zur Rückkehradresse, die vom vorhergehenden CALL auf dem Stack abgelegt wurde; Weiterzählen des Stackpointers um 2
RET numb	siehe RET, aber Weiterzählen des Stackpointers um 2 + numb
RETF	Rücksprung zur Adresse, die von CALLF auf dem Stack abgelegt wurde; Weiterzählen des Stackpointers um 4
RETF numb	Rücksprung zur Adresse, die vom letzten CALLF auf dem Stack abgelegt wurde; Weiterzählen des Stackpointers um 4 + numb

4.6. Prozessorsteuerbefehle

Prozessorsteuerbefehle beeinflussen die Flagregister. Darüber hinaus können einige von diesen Befehlen die ZVE mit externer Gerätetechnik synchronisieren.

Tabelle 20: Prozessorsteuerbefehle

Syntax	Ergebnis
CLC	Löschen des CF-Flag
CLD	Löschen des DF-Flag; veranlaßt, daß bei Zeichenkettenbefehlen das Operandenregister selbständig erhöht wird
CLI	Löschen des IF-Flag; damit sind externe Unterbrechungen nicht maskierbar
CMC	Komplement des CF-Flag
ESC numb8,mem|reg	keine Operation, außer, daß die effektive Adresse berechnet und auf den Adreßbus gebracht wird; (ESC wird vom Numeric-Co-Prozessor benutzt) numb8 muß im Bereich von 0 ... 63 liegen
HLT	Prozessor geht in den Haltzustand, bis eine Unterbrechung erkannt wird
LOCK	Präfix-Befehl, der den Prozessor veranlaßt, für die Dauer der Operation, die dem LOCK folgt, den Bus zu sperren (bus-lock). Der LOCK-Präfix darf jedem anderen Befehl vorangesetzt werden. Bus-lock hindert Co-Prozessoren, den Bus zu benutzen.
NOP	keine Operation
STC	Setzen des CF-Flag
STD	Setzt das DF-Flag. Veranlaßt, daß bei Zeichenkettenbefehlen das Operandenregister selbständig erniedrigt wird.
STI	Setzen des IF-Flag; externe Unterbrechungen werden maskierbar
WAIT	Der Prozessor geht in den Wartezustand, falls das Signal auf seinen Test-Pin nicht behauptet wird.

5. Code-Makro-Möglichkeiten

5.1. Einführung

Mit RASM86 hat der Anwender die Möglichkeit, eigene Befehle oder Befehlsfolgen zu definieren. Das erfolgt mit Hilfe der Code-Makros. Diese haben allerdings nur eine geringe Ähnlichkeit mit den herkömmlichen Assembler-Makros, die gewöhnlich beliebige Assemblerbefehle enthalten können.
Die Code-Makros von RASM86 enthalten nur Code-Makro-Direktiven. Während die herkömmlichen Assembler-Makros in der Symboltabelle des Assemblers geführt werden, sind die Code-Makros in einer anderen (internen) Symboltabelle definiert.
Mit der herkömmlichen Makrotechnik kann die wiederholte Anwendung gleicher Befehlsblöcke im gesamten Programm vereinfacht werden. Code-Makros generieren dagegen nur eine Bitfolge in die Objektdatei. So sind die gesamten Maschinenbefehle über Code-Makros definiert und weitere neue Befehle oder Befehlsfolgen können zusätzlich entstehen.
RASM86 behandelt einen Code-Makro wie einen Befehl. Ein Code-Makro wird auch wie ein Befehl aufgerufen.
Das folgende Beispiel zeigt, wie der Code-Makro MYCODE aufgerufen wird. MYCODE ist ein durch einen Code-Makro definierter Befehl.

Beispiel:

```
        .
        .
XCHG    BX,WORD3
MYCODE  PARM1,PARM2
MUL     AX,WORD4
        .
        .
```

Der Aufruf hat zwei Operanden, welche die aktuellen Parameter des Code-Makros sind. Aktuelle Parameter sind immer Symbole, Konstanten oder Register. Bei der Definition des Code-Makros werden zwei Operanden als formale Parameter (Platzhalter) benutzt. RASM86 klassifiziert diese zwei Operanden in Typ, Größe usw. Die Namen der formalen Parameter sind nicht festgelegt, sie sind frei wählbar. Beim Aufruf der Code-Makros ersetzt RASM86 die formalen Parameter durch die aktuellen Parameter; formale Parameter sind nur Platzhalter. Sie zeigen an, wo und wie die Operanden zu verwenden sind.
Ein Code-Makro hat die allgemeine Form:

```
CODEMACRO name [formal-parameter, ...]
    code-macro-body
ENDM
```

Ein Code-Makro wird durch das Schlüsselwort CODEMACRO eingeleitet. Der Name ist ein Bezeichner und identifiziert den Code-Makro.
Formale Parameter haben die allgemeine Form:

```
formal-name:specifier-letter[modifier-letter][range]
```

Ein formaler Parameter besteht aus dem

- formalen Namen (formal-name) und aus dem

- Attributbuchstaben (specifier-letter).

Der Modifikationsbuchstabe (modifier-letter) und die Bereichsattribute (range) sind wahlfrei anzugeben.
Mögliche Spezifikationsbuchstaben sind:
A, C, D, E, M, R, S und X.
Mögliche Modifikationsbuchstaben sind: b, d, w und sb.
Innerhalb der Code-Makros ist die Groß- und Kleinschreibung beliebig. Zur besseren Übersicht werden in diesem Abschnitt

- Spezifikatoren (specifier-letter) in Großbuchstaben und
- Modifikatoren (modifier-letter) in Kleinbuchstaben angegeben.

In den folgenden Abschnitten wird die Liste der formalen Parameter (Spezifikatoren, Modifikatoren, Bereichsangaben) im einzelnen beschrieben.
Der Körper eines Code-Makros beschreibt die Bitmuster und formalen Parameter. Nur die folgenden Direktiven sind in Code-Makros zulässig:

```
SEGFIX      RELB    DB
NOSEGFIX    RELW    DW
MODRM       DBIT    DD
```

Die Code-Makro-Direktiven DB, DW und DD sind zwar auch als RASM86-Direktiven vorhanden, haben aber in Code-Makros eine andere Bedeutung. Diese Anweisungen werden in Abschnitt 5.5.5. genauer erläutert.
CODEMACRO, ENDM und die Code-Makro-Direktiven sind reservierte Worte. Die formale Definition der Syntax eines Code-Makros wird in der Syntaxbeschreibung in der Anlage 4 definiert.
Die folgenden Beispiele sind typische Code-Makro-Definitionen:

Beispiel:

```
CODEMACRO   AAA
   DB  37H
ENDM


CODEMACRO   DIV DIVISOR:Eb
   SEGFIX   DIVISOR
   DB       0F6H
   MODRM    6,DIVISOR
ENDM


CODEMACRO   ESC OPCODE:Db(0,63),SRC:Eb
   SEGFIX   SRC
   DBIT     5(1BH),3(OPCODE(3))
   MODRM    OPCODE,SRC
ENDM
```

5.2. Attribute

Jeder formale Parameter muß einen Attributbuchstaben (specifier-letter) haben, der den benötigten Operandentyp des formalen Parameters angibt. Dieser muß beim Aufruf mit dem Typ des aktuellen Parameters übereinstimmen.
Tabelle 21 definiert die acht möglichen Attribute.

Tabelle 21: Operandenattribute von Code-Makros

Buchstabe	Operandentyp
A	Akkumulatorregister AX oder AL
C	Code, nur ein Markenausdruck
D	Daten, eine Zahl, die wie ein Direktwert genutzt wird
E	effektive Adresse, entweder M (Speicheradresse) oder R (Register)
M	Speicheradresse, die entweder eine Variable oder ein geklammerter Registerausdruck sein kann
R	nur ein allgemeines Register
S	nur ein Segmentregister
X	direkter Speicherbezug

5.3. Modifikatoren

Der Modifikationsbuchstabe (modifier-letter) ist eine weitere, wahlfreie Anforderung an den Operanden. Die Bedeutung des Modifikationsbuchstabens hängt vom Typ des Operanden ab. Für Variable muß der Modifikator folgenden Typ besitzen:

- b für Byte
- w für Wort
- d für Doppelwort
- sb für Byte mit mit Vorzeichen

Für Zahlen fordern die Modifikatoren eine bestimmte Größe:

- b von -256 bis 255 und
- w für weitere Zahlen

Tabelle 22 faßt die Code-Makro-Modifikatoren zusammen.

Tabelle 22: Operandenmodifikatoren von Code-Makros

Variable		Zahlen	
Modifikator	Typ	Modifikator	Größe
b	Byte	b	-256 bis 255
w	Wort	w	alles andere
d	D-Wort		
sb	Byte mit Vorzeichen		

5.4. Bereichsattribute

Der wahlfreie Bereich (range) wird entweder durch einen oder durch zwei Ausdrücke angegeben, die durch Komma getrennt und in runde Klammern eingeschlossen sind. Folgende Formate sind zulässig:

(numberb)
(register)
(numberb, numberb)
(numberb, register)
(register, numberb)
(register, register)

Numberb ist eine 8-bit-Zahl, keine Adresse. Das folgende Beispiel gibt an, daß der Eingabekanal (input-port) durch das DX-Register spezifiziert werden muß.

```
CODEMACRO  IN  DST:Aw, PORT:Rw(DX)
```

Im nächsten Beispiel wird angegeben, daß das Register CL den Zähler (COUNT) der zyklischen Verschiebung (rotation) enthalten muß:

```
CODEMACRO  ROR  DST:Ew, COUNT:Rb(CL)
```

Im letzten Beispiel wird gezeigt, daß der formale Parameter (OPCODE) Direktdaten darstellt, die im Bereich von 0 bis 63 liegen können:

```
CODEMACRO  ESC  OPCODE:Db(0,63), ADDS:Eb
```

5.5. Code-Makro-Direktiven

Code-Makro-Direktiven definieren die Bitmuster des generierten Codes und bestimmen, wie die Operanden zu behandeln sind. Direktiven sind reservierte Worte. Diejenigen Direktiven, die auch in der Assemblersprache vorkommen (DB, DW, DD) haben innerhalb von Code-Makros eine andere Bedeutung.
Nur die neun hier definierten Direktiven sind innerhalb von Code-Makro-Definitionen zulässig.

5.5.1. SEGFIX

SEGFIX veranlaßt RASM86 zu entscheiden, ob ein Segment-Override-Präfix-Byte für den Zugriff zu einem gegebenen Speicherplatz nötig ist. Wenn es so ist, wird es als erstes Byte des Befehls angegeben, andernfalls erfolgt durch RASM86 keine Reaktion.
SEGFIX hat die Form:

```
SEGFIX  formal-name
```

wobei formal-name der Name eines formalen Parameters ist, der die Speicheradresse repräsentiert. Da der formale Parameter eine Speicheradresse repräsentiert, muß er eines der Attribute E, M, oder X haben.

5.5.2. NOSEGFIX

NOSEGFIX wird für Operanden in Befehlen benutzt, die das ES-Register für diesen Operanden benutzen müssen. Diese Anwendung beschränkt sich nur auf den Zieloperanden folgender Befehle: CMPS, MOVS, SCAS, STOS.
NOSEGFIX hat die Form:

```
NOSEGFIX  segreg, formal-name
```

wobei segreg eines der Segmentregister ES, CS, SS oder DS ist, und formal-name ist der Name der Speicheradresse als formaler Parameter. Dieser muß ein Attribut E, M oder X haben. Von dieser Direktive wird kein Code generiert, aber ein Fehlertest wird ausgeführt.

Beispiel:

```
CODEMACRO   MOVS   SI_PTR:Ew,DI_PTR:Ew
   NOSEGFIX        ES,DI_PTR
   SEGFIX          SI_PTR
   DB              0A5H
ENDM
```

5.5.3. MODRM

Diese Direktive veranlaßt RASM86,das MODRM-Byte zu generieren, das dem Operationscode-Byte (opcode-byte) in vielen Befehlen folgt. Das MODRM-Byte enthält entweder den Indextyp oder die Registernummer, die im Befehl benutzt wird. Es gibt außerdem an, welches Register zu benutzen ist oder gibt mehr Informationen zur Spezifizierung eines Befehls.
Das MODRM-Byte enthält die Informationen in drei Feldern:

Das Modusfeld (mod) belegt die zwei höchsten Bits des Bytes und bildet, verbunden mit dem Register/Speicherfeld (r/m), 32 mögliche Werte: 8 Register und 24 Indexierungsmodi.

Das Registerfeld (reg) belegt die drei nächsten Bits nach dem Modusfeld. Es gibt entweder eine Registerzahl oder drei weitere Bits der Opcode-Information an. Die Bedeutung des Registerfeldes ist durch das Opcode-Byte bestimmt.

Das Register/Speicherfeld (r/m) belegt die letzten 3 Bit im Byte. Es spezifiziert ein Register als einen Platz eines Operanden oder bildet einen Teil des Adressierungsmodus in Verbindung mit dem oben beschriebenen Modusfeld.
MODRM hat die Formen:

```
MODRM  formal-name, formal-name
MODRM  NUMBER7, formal-name
```

wobei NUMBER7 ein Wert von 0 bis 7 ist und formal-name ist der Name des formalen Parameters. Die folgenden Beispiele zeigen die Anwendung von MODRM:

Beispiel:

```
CODEMACRO  RCR DST:Ew,COUNT:Rb(CL)
   SEGFIX       DST
   DB           0D3H
   MODRM        3,DST
ENDM

CODEMACRO  OR DST:Rw,SRC:Ew
   SEGFIX       SRC
   DB           0BH
   MODRM        DST,SRC
ENDM
```

5.5.4. RELB und RELW

Diese Direktiven werden für Sprungbefehle verwendet. RASM86 generiert einen Abstand (displacement) zwischen dem Ende des Befehls und der Marke, die als Operand angegeben ist. RELB generiert ein Byte und RELW zwei Bytes für den Abstand.
Die Direktiven haben folgende Form:

```
RELB  formal-name
RELW  formal-name
```

wobei formal-name der Name des formalen Parameters mit einem C(Code)-Attribut ist.

Beispiel:

```
CODEMACRO  LOOP  PLACE:Cb
   DB             0E2H
   RELB           PLACE
ENDM
```

5.5.5. DB, DW und DD

Diese Direktiven unterscheiden sich von denjenigen, die außerhalb der Code-Makros angewendet werden.
Die Direktiven haben folgende Formen:

```
DB  formal-name | NUMBERB
DW  formal-name | NUMBERW
DD  formal-name
```

wobei NUMBERB eine Einbytezahl ist. NUMBERW ist eine Zweibytezahl und formal-name ist der Name des formalen Parameters.

Beispiel:

```
CODEMACRO  XOR  DST:Ew,SRC:Db
   SEGFLX        DST
   DB            81H
   MODRM         6,DST
   DW            SRC
ENDM
```

5.5.6. DBIT

Diese Direktive behandelt Bits innerhalb eines Bytes oder einer kleineren Speichereinheit.
Die Form ist:

```
DBIT field-description, ...
```

wobei field-description zwei Formen haben kann:

```
number combination
number (formal-name(rshift))
```

wobei number im Bereich von 1 bis 16 liegt und die Zahl der zu setzenden Bits darstellt. Die gewünschte Bitkombination wird durch combination angegeben. Die Gesamtheit aller numbers, die in der field-description aufgelistet sind, darf 16 nicht überschreiten.
Die zweite darunter gezeigte Form enthält einen formalen Parameternamen (formal-name), der den Assembler veranlaßt, eine bestimmte Zahl auf eine bestimmte Position zu legen. Diese Zahl bezieht sich normalerweise auf das in der ersten Zeile des Code-Makros angegebene Register. Die Zahlen, die in diesem speziellen Fall für jedes Register benutzt werden, sind:

```
AL: 0   AH: 4   AX: 0   SP: 4   ES: 0
CL: 1   CH: 5   CX: 1   BP: 5   CS: 1
DL: 2   DH: 6   DX: 2   SI: 6   SS: 2
BL: 3   BH: 7   BX: 3   DI: 7   DS: 3
```

Der Parameter rshift in der innersten Klammer gibt eine Zahl von Rechtsverschiebungen an.

Beispiel: 0 keine Verschiebung
1 verschiebt um 1 Bit nach rechts
2 verschiebt 2 Bit nach rechts usw.

Die folgende Definition benutzt diese Form:

```
CODEMACRO  DEC  DST:Rw
   DBIT  5(9H),3(DST(0))
ENDM
```

Die ersten 5 Bit des Bytes haben den Wert 9H. Falls die Restbits Null sind, wird der hexadezimale Wert des Bytes 48H.
Wenn der Befehl

```
     DEC  DX
```

übersetzt wird und DX hat den Wert 2H, dann ist 48H + 2H = 4AH, was den Endwert des Bytes zur Ausführung darstellt.
Ist aber folgende Definition gegeben:

```
     DBIT  5(9H),3(DST(1))
```

dann würde die Registerzahl einmal nach rechts verschoben werden und das Ergebnis würde 48H + 1H = 49H sein, was falsch ist.

6. XREF86

6.1. Einführung

XREF86 ist ein Cross-Referenz-Programm der Assemblersprache. Es erzeugt eine Cross-Referenz-Datei, die die Nutzung aller Symbole des gesamten Programms aufzeigt. XREF86 verarbeitet zwei von RASM86 erzeugte Dateien. XREF86 setzt voraus, daß diese Eingabedateien die Dateitypen LST und SYM haben und sich beide auf dem gleichen Laufwerk befinden. XREF86 erzeugt eine Ausgabedatei vom Dateityp XRF.
Bild 2 erläutert die XREF86-Funktionen.

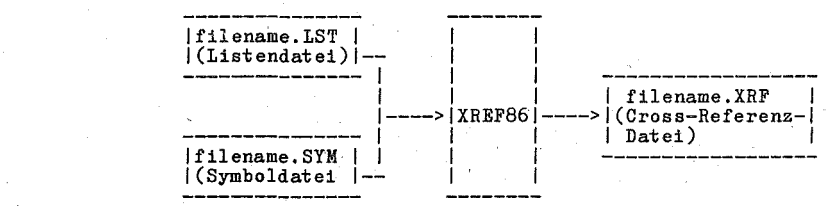


Bild 2: XREF86-Funktionen

6.2. Aufruf

XREF86 wird mit folgendem Kommando aufgerufen:

```
XREF86 filename
```

XREF86 liest die Listendatei filename.LST Zeile für Zeile ein und versieht jede Zeile mit einer Zeilennummer am Anfang. Jede dieser Zeilen wird in die Ausgabedatei filename.XRF geschrieben. Während dieses Vorgangs durchsucht XREF86 jede Zeile nach Symbolen, die in der Datei filename.SYM stehen.
Nach Vollendung dieser Kopieroperation hängt XREF86 an die Datei filename.XRF einen Cross-Referenzteil an, der alle Zeilennummern enthält, in denen jedes Symbol aus der Datei filename.SYM erscheint. XREF86 kennzeichnet jeden Zeilenzahlbezug mit einem # (Doppelkreuz), wo das bezugnehmende Symbol als erstes in der Zeile erscheint.
XREF86 listet auch die Werte von jedem Symbol auf, die von RASM86 bestimmt und in die Symboltabellendatei filename.SYM eingetragen sind.
Beim Aufruf von XREF86 kann wahlweise ein Laufwerk mit dem Dateinamen angegeben werden. Wenn XREF86 mit einem dem Dateinamen filename vorangestellten Laufwerk aufgerufen wird, sucht es nach der Eingabedatei und erzeugt die Ausgabedatei auf dem angegebenen Laufwerk. Andernfalls greift XREF86 zum Standardlaufwerk zu.
XREF86 kann auch die Ausgabedatei über das Standardlistengerät ausgeben. Dazu muß das Zeichen ¤p an die Kommandozeile angehängt werden.

Beispiel:

```
A>XREF86 BIOS ¤p
A>XREF86 B:BLST
```

7. Linker LINK86

7.1. Einführung

LINK86 ist ein Verbindeprogramm, das verschiebliche Objektdateien zu einer Kommandodatei verbindet. Die Objektdatei kann durch einen Sprachübersetzer wie RASM86 erzeugt werden oder durch andere Übersetzer, die Objektdateien erzeugen, welche eine kompatible Untermenge des Objektmodulformats verwenden.
LINK86 akzeptiert zwei Typen von Objektdateien. Der erste Typ ist eine Objektdatei, die einen einzigen Objektmodul enthält. Dieser Typ hat im allgemeinen den Dateityp OBJ und wird durch den Sprachübersetzer erzeugt. Der zweite Typ ist eine Bibliotheksdatei, und zwar eine indexierte Bibliothek von Objektmoduln. Eine Bibliotheksdatei hat den Dateityp L86 und wird vom Bibliothekar LIB86 im WM86-Objektmodulformat erzeugt. LINK86 kann eine solche Bibliotheksdatei durchsuchen und nur jene Moduln auswählen, die von anderen gelinkten Programmen benötigt werden.

LINK86 erzeugt drei Dateien:

- Kommandodatei (Typ CMD)
- Symboltabellendatei (Typ SYM)
- Listendatei (Typ MAP)

Die CMD-Datei enthält ein Speicherabbild des Programms, das direkt unter dem SCP 1700 läuft. Die SYM-Datei enthält eine Liste von Symbolen aus den Objektdateien und deren Offsets. Die MAP-Datei enthält Informationen über den Umfang der CMD-Datei.

LINK86 teilt alle unaufgelösten Symbole auf dem Terminal mit. Unaufgelöste Symbole sind Symbole, auf die Bezug genommen wurde, die aber in den gelinkten Dateien nicht definiert wurden. Diese Symbole müssen aufgelöst werden, bevor das Programm korrekt läuft.

Zum Abschluß der Verarbeitung schreibt LINK86 die Größe jeder Sektion der CMD-Datei und den Nutzungsfaktor, einen dezimalen Prozentsatz, der die Größe des verfügbaren und von LINK86 genutzten Hauptspeichers angibt.

Bild 3 veranschaulicht die Arbeitsweise des Linkers.

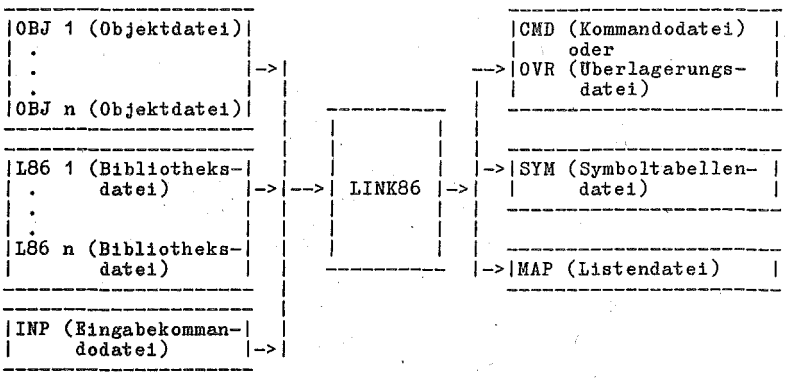


Bild 3: Arbeitsweise des Linkers

7.2. Aufruf

LINK86 wird mit einem Kommando folgender Form aufgerufen:

```
LINK86 {file=}file1{, file2, ... , filen}
```

Wenn ein Dateiname links vom Gleichheitszeichen eingegeben wird, so erzeugt LINK86 die Ausgabedateien mit diesem Namen und den zugehörigen Dateitypen. Das Kommando

```
A>LINK86 MYFILE = PART1, PART2, PART3
```

erzeugt die Dateien MYFILE.CMD und MYFILE.SYM.

Wird kein neuer Name angegeben, erzeugt LINK86 die Ausgabedateien unter Verwendung des ersten Dateinamens aus der Kommandozeile. Zum Beispiel das Kommando

```
A>LINK86 PART1, PART2, PART3
```

erzeugt die Dateien PART1.CMD und PART1.SYM.

LINK86 kann auch angewiesen werden, seine Kommandozeile aus einer Datei zu lesen, um zu ermöglichen, lange oder allgemein genutzte LINK86-Kommandos auf der Platte zu speichern (siehe Abschnitt 7.11.).

7.3. Anhalten

LINK86 kann während der Verarbeitung durch Betätigen irgendeiner Terminaltaste angehalten werden. LINK86 schreibt folgende Mitteilung aus:

```
STOP LINK86 (Y/N)?
```

Bei der Eingabe des Buchstabens Y beendet LINK86 unbedingt die Bearbeitung und gibt die Steuerung an das Betriebssystem zurück. Die Eingabe des Buchstabens N veranlaßt LINK86, die Verarbeitung wieder aufzunehmen.

7.4. Definitionen

In diesem Abschnitt werden folgende Ausdrücke verwendet, um zu beschreiben, wie LINK86 Objektdateien verarbeitet und die CMD-Datei erzeugt.

Segment — Ein Segment ist eine Zusammenstellung von Code- oder Datenbytes, dessen Länge kleiner als 64K Byte ist. Ein Segment ist die kleinste Einheit, die LINK86 bei der Erzeugung der CMD-Datei behandelt.

Segmentname*) — Ein Segmentname kann jeder gültige RASM86-Bezeichner sein. LINK86 verbindet alle Segmente mit dem gleichen Namen aus verschiedenen Objektdateien.

Klassenname*) — Ein Klassenname kann jeder gültige RASM86-Bezeichner sein. LINK86 verwendet den Klassennamen, um das Segment in die richtige Sektion der CMD-Datei zu positionieren.

Zuordnungstyp*) — Der Zuordnungstyp kennzeichnet, an welchem Grenztyp das Segment beginnen soll. Die Zuordnungstypen sind Byte, Wort, Paragraph und Seite (page). LINK86 verwendet den Zuordnungstyp auf zwei Arten: erstens, wenn er Teile von Segmenten aus verschiedenen Segmenten verbindet und zweitens, wenn er Segmente zu Gruppen oder Sektionen der CMD-Datei verbindet.

Verbindetyp*) — Der Verbindetyp bestimmt, wie LINK86 Teile von Segmenten mit dem gleichen Namen aus verschiedenen Moduln verbindet. Die Verbindetypen sind: PUBLIC, COMMON, STACK, ABSOLUTE und LOCAL.

Sektion*) — Eine Sektion ist ein Teil von maximal acht Teilen einer Kommandodatei (CMD-Datei), wobei ein einziger von ihnen bis zu einem Megabyte lang sein kann.

Gruppe*) — Eine Gruppe ist eine Zusammenstellung von Segmenten, deren vollständige Länge kleiner als 64K ist und die durch ein einziges Segmentregister adressierbar ist. Gruppen erlauben es, ein Programm so in Segmente aufzugliedern, wobei es noch erlaubt ist, daß die Segmente ohne eine

Veränderung des Inhalts des Segmentregisters adressiert werden können. Dieses Verfahren ergibt einen kürzeren und schnelleren Befehlscode als eine Adressierung der Segmente mit 32-bit-Zieladressen.

*) Wenn in einer höheren Programmiersprache programmiert wird, so weist der Compiler automatisch den Segmentnamen, den Klassennamen, die Gruppe, den Zuordnungstyp und den Verbindetyp zu. Wird in der Assemblersprache programmiert, so wird auf den Abschnitt 3. verwiesen, in dem beschrieben wird, wie diese Attribute zuzuordnen sind.

7.5. Verbindeprozeß

Der Verbindeprozeß schließt zwei unterschiedliche Phasen ein, Sammeln der Segmente in den Objektdateien und danach ihre Positionierung in der CMD-Datei.

7.5.1. Phase 1 - Sammeln

In der Phase 1 sammelt LINK86 zuerst Segmente aus den verschiedenen zu verbindenden Dateien, die den gleichen Segmentnamen und den gleichen Klassennamen haben und verbindet dann die Segmente entsprechend der Zuordnungs- und Verbindeattribute.

Es sind z. B. drei Objektdateien gegeben: FILEA.OBJ, FILEB.OBJ und FILEC.OBJ. Jede Datei definiert ein Segment mit dem Namen DATASEG durch die Anweisung:

```
DATASEG   DSEG
```

Bild 4 veranschaulicht den PUBLIC-Verbindetyp.

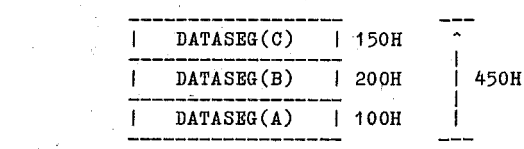


Bild 4: Verbinden der Segmente mit dem PUBLIC-Verbindetyp

LINK86 verbindet diese Segmente durch Verketten der Teile von den Segmenten, die in den verschiedenen Objektdateien gefunden und die mit dem zugehörigen Leerraum zwischen den Teilen geordnet werden, der durch den Zuordnungstyp (siehe unten) gekennzeichnet wird. PUBLIC ist der allgemeinste Verbindetyp und RASM86 sowie die meisten Übersetzer für höhere Programmiersprachen erzeugen ihn standardmäßig.

Bild 5 veranschaulicht den COMMON-Verbindetyp. Die drei Dateien FILEA.OBJ, FILEB.OBJ und FILEC.OBJ enthalten jeweils ein Segment mit dem Namen DATASEG, das mit folgender Anweisung definiert wird:

```
DATASEG   DSEG    COMMON
```

LINK86 verbindet diese Segmente so, daß alle Teile der Segmente aus den verschiedenen zu linkenden Dateien die gleiche untere Adresse im Speicher haben. Das entspricht einem COMMON-Block in höheren Programmiersprachen.

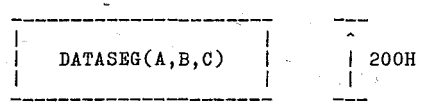


Bild 5: Verbindung von Segmenten mit dem COMMON-Verbindetyp

Bild 6 veranschaulicht den STACK-Verbindetyp. Die Gesamtlänge des Segmentes ist die Summe der Teile aus den verschiedenen zu linkenden Dateien, einschließlich jener Zwischensegmentlücken entsprechend des Zuordnungstyps. Jedoch haben alle Teile die gleiche obere Adresse gemeinsam, da der Stackbereich von hohen Speicheradressen abwärts wächst.

Die drei Dateien FILEA.OBJ, FILEB.OBJ und FILEC.OBJ enthalten jeweils ein Segment mit dem Namen STKSEG. Bild 6 zeigt, wie LINK86 die Anteile verbindet.

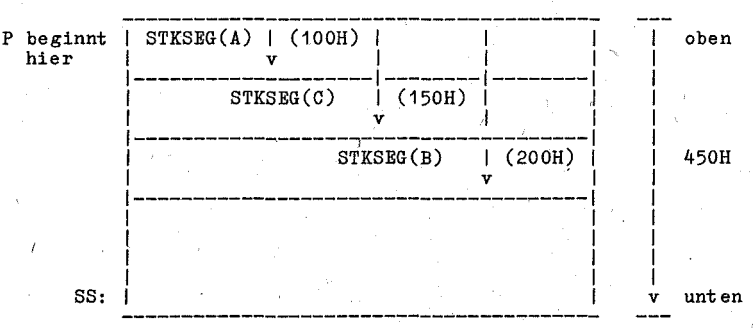


Bild 6: Verbindung von Segmenten des STACK-Verbindetyps

Der Zuordnungstyp zeigt, an welcher Speichergrenze das Einzelsegment beginnt, und bestimmt so den Umfang des Leerraumes, den LINK86 zwischen Teilen gleichnamiger Segmente frei läßt. Zum Beispiel sollen die drei Dateien FILEA.OBJ, FILEB.OBJ und FILEC.OBJ jeweils ein Segment mit dem Namen DATASEG enthalten. Bild 7 veranschaulicht, wie LINK86 den Zuordnungstyp verwendet, um diese Segmente zu verbinden.

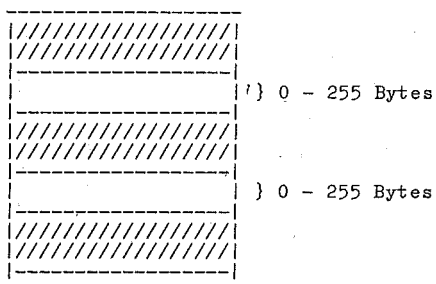


Bild 7: Verbindung von Segmenten,
die den Zuordnungstyp verwenden

Im Bild 7 wird die Lücke zwischen den Segmenten durch den Zuordnungstyp bestimmt und kann bis zu 255 Byte lang sein. Beispielsweise gibt es keine Lücke, wenn der Zuordnungstyp BYTE ist. Dieser Zuordnungstyp erzeugt den kompaktesten Code.

Wenn der Zuordnungstyp WORD ist, addiert LINK86 bei Bedarf eine Ein-Byte-Lücke, um zu sichern, daß der nächste Teil des Segments an einer Wortgrenze beginnt. WORD ist der Standard-Zuordnungstyp für Data-Segmente, da der Prozessor bei Wortoperationen schneller Zugriff hat.

Die Lücke für Segmente mit Paragraphzuordnung kann maximal 15 Byte sein, während Segmente mit Seitenzuordnung bis zu 255 Byte erfordern können.

Bild 8 veranschaulicht ein spezielles Beispiel. Das Segment DATASEG hat den Zuordnungstyp PARAGRAPH. DATASEG hat eine Länge von 129H in FILEA, 10EH in FILEB und 13AH in FILEC. Es wird gezeigt, wie LINK86 die Segmente verbindet, wobei abgesichert wird, daß jedes Segment an einer Paragraphgrenze beginnt.

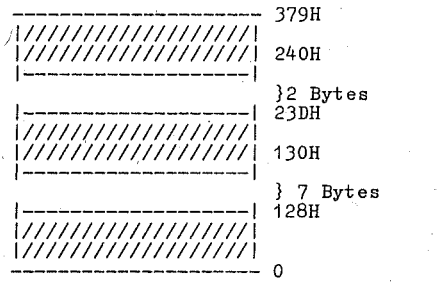


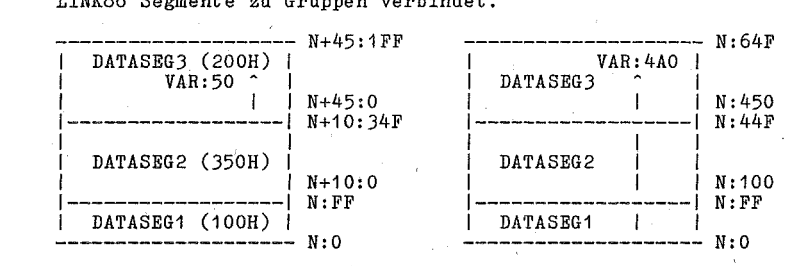
Bild 8: Paragraph-Zuordnung

LINK86 ordnet keine Segmente, die einen ABSOLUTE-Zuordnungstyp haben, weil der zur Ladezeit vorgesehene Speicherplatz dieser Segmente bereits zur Übersetzungszeit festgelegt wird.

Segmente mit dem LOCAL-Verbindetyp können nicht verbunden werden. LINK86 gibt eine Fehlermeldung aus, wenn die zu linkenden Dateien mehrere lokale Segmente mit dem gleichen Namen enthalten.

7.5.2. Phase 2 - Positionierung

In Phase 2 verbindet LINK86 Segmente, die zugehörige Teile einer Gruppe sind, und verwendet wieder den Zuordnungstyp, um die Zwischensegmentlücken zu bestimmen. Bild 9 veranschaulicht, wie LINK86 Segmente zu Gruppen verbindet.



a) Segmente ohne Gruppen

b) Segmente innerhalb einer Gruppe

Bild 9: Wirkung der Gruppierung von Segmenten

Im Bild 9 ist N:0 die Basisadresse, ab der die Segmente zur Laufzeit geladen werden (Paragraph N, Offset 0). Im linken Bild (a) beginnt jedes Segment, das hier in keiner Gruppe enthalten ist, mit dem Offset Null und kann so bis zu 64K Byte lang sein. Der Offset irgendeines ausgegebenen Speicherplatzes, in diesem Falle die Variable VAR, ist relativ zur Basis des Segments. Um so zur Laufzeit zur Variablen VAR zuzugreifen, muß das Programm ein Segmentregister mit der Basis des Segments DATASEG3 laden, weil LINK86 der Variablen VAR einen Offset von 50H zuweist.

Rechts im Bild 9 (b) werden die gleichen Segmente in einer Gruppe verbunden. Die Offsets der Segmente werden addiert und können sich nicht weiter als FFFFH ausdehnen. Der Offset der Variablen VAR ist 500H relativ zur Basis der Gruppe. Zur Laufzeit braucht das Programm kein Segmentregister zu wechseln, um DATASEG3 zu adressieren, aber das Programm kann zur VAR direkt zugreifen, wobei das Segmentregister verwendet wird, das zur Basis der Gruppe weist.

Nach dem Verbinden von Segmenten zu Gruppen weist LINK86 jedes Segment einer Sektion der CMD-Datei wie folgt zu:

1) Segmente, die zur Gruppe CGROUP gehören, werden in die CODE-Sektion der CMD-Datei eingeordnet.

2) Segmente, die zur Gruppe DGROUP gehören, werden in die DATA-Sektion der CMD-Datei eingeordnet. Es ist zu beachten, daß die Gruppennamen CGROUP und DGROUP automatisch durch Übersetzer für höhere Programmiersprachen erzeugt werden.

3) Wenn es Segmente gibt, die noch nicht entsprechend Punkt 1) und 2) verarbeitet worden sind, dann ordnet LINK86 sie so in

der CMD-Datei entsprechend ihres Klassennamens, wie es in Tabelle 23 gezeigt wird. Diese Tabelle zeigt auch die RASM86-Segmentanweisungen, die den Klassennamen standardmäßig erzeugen.

4) Segmente, die nicht durch eines der obigen Verfahren erzeugt worden sind, werden in der CMD-Datei weggelassen, weil LINK86 nicht genügend Informationen hat, sie zu positionieren.

Die Art, wie LINK86 Segmente positioniert, kann verändert werden, indem Optionen in der Kommandozeile verwendet werden, siehe dazu Abschnitt 7.7.

Tabelle 23: LINK86-Behandlungsweise von Klassennamen

Klassenname	Sektion der CMD-Datei	Segmentanweisung (bei RASM86)
CODE	CODE	CSEG
DATA	DATA	DSEG
EXTRA	EXTRA	ESEG
STACK	STACK	SSEG
X1 *)	X1	
X2 *)	X2	
X3 *)	X3	
X4 *)	X4	

*) Es gibt keine Segmentanweisung, die diesen Klassennamen standardmäßig erzeugt. Der Klassenname muß explizit angegeben werden.

7.6. Optionen der LINK86-Kommandos

Wird LINK86 aufgerufen, können in Kommandozeilen Optionen angegeben werden, die die Arbeit des LINK86 steuern. Jede Kommandooption führt zu einer von mehreren Kategorien, die abhängig vom Typ der beeinflußten Datei ist.

Die erste Kategorie von Kommandooptionen beeinflußt den Inhalt der CMD-Datei und bezieht sich deshalb auf die vollständige Linkoperation. Die zweite Kategorie von Optionen beeinflußt die Symboldatei (SYM-Datei) und die Listendatei (MAP-Datei). Diese Optionen dienen als Ein-, Aus- und Umschalter, in der Reihenfolge wie LINK86 die Kommandozeile von links nach rechts abarbeitet. Die dritte Kategorie von Optionen betrifft die Bibliotheks- und Eingabedateien und bezieht sich deshalb auf nur eine Datei in der Kommandozeile.

In Tabelle 24 sind die Kommandooptionen von LINK86 einschließlich deren Abkürzungen aufgeführt.

Tabelle 24: Kommandooptionen des LINK86

Option	Abkürzung	Bedeutung
CODE	C	steuert den Inhalt der CODE-Sektion der CMD-Datei
DATA	DA	steuert den Inhalt der DATA-Sektion der CMD-Datei
EXTRA	E	steuert den Inhalt der EXRTA-Sektion der CMD-Datei
STACK	ST	steuert den Inhalt der STACK-Sektion der CMD-Datei
X1	X1	steuert den Inhalt der X1-Sektion der CMD-Datei
X2	X2	steuert den Inhalt der X2-Sektion der CMD-Datei
X3	X3	steuert den Inhalt der X3-Sektion der CMD-Datei
X4	X4	steuert den Inhalt der X4-Sektion der CMD-Datei
FILL	F	Einfügen von Nullen und nicht initialisierter Daten in die CMD-Datei
NOFILL	NOF	kein Einfügen nicht initialisierter Daten in die CMD-Datei
INPUT	I	Lesen einer Kommandozeile aus einer Plattendatei
MAP	M	Erzeugen einer MAP-(Listen-)Datei
LIBSYMS	LI	Einfügen von Symbolen aus Bibliotheksdateien in die SYM-Datei
NOLIBSYMS	NOLI	kein Einfügen von Symbolen aus Bibliotheksdateien in die SYM-Datei
LOCALS	LO	Einfügen lokaler Symbole in die SYM-Datei
NOLOCALS	NOLO	kein Einfügen lokaler Symbole in die SYM-Datei
SEARCH	S	Durchsuchen einer Bibliothek und Linken der Moduln, auf die Bezug genommen wird

Die Kommandooptionen, die unmittelbar einem Dateinamen folgen, werden in eckige Klammern eingeschlossen. Zum Beispiel:

```
A>LINK86 TEST1 [MAP], TEST2 [NOLOCALS]
```

Leerzeichen können verwendet werden, um die Lesbarkeit der Kommandozeile zu verbessern und in den eckigen Klammern können mehrere, durch Kommas getrennte, Optionen angegeben werden. Zum Beispiel:

```
A>LINK86 TEST1 [MAP, NOLOCALS], TEST2 [LOCALS]
```

Die folgenden Unterabschnitte beschreiben die Funktion und Syntax für jede der Kommandooptionen.

7.7. Optionen der CMD-Datei

Die folgenden Kommandooptionen beeinflussen den Inhalt der CMD-Datei, die von LINK86 erzeugt wird:

```
CODE DATA STACK EXTRA
X1   X2   X3    X4
FILL NOFILL
```

Es ist zu beachten, daß diese Optionen in der Kommandozeile nach jedem Dateinamen auftreten können.

Die ersten acht Optionen steuern die Art, wie LINK86 Segmente in der CMD-Datei und den Inhalt des CMD-Datei-Headers zuordnet. Die FILL- und NOFILL-Optionen weisen LINK86 an, was mit nicht initialisierten Daten zu tun ist, die am Ende einer Sektion der CMD-Datei auftreten können.

Eine CMD-Datei besteht aus einem 128-Byte-Header, dem bis acht Sektionen folgen können. Eine Sektion kann bis zu 1 Megabyte, dem Umfang des Gesamtspeichers, lang sein. Diese Sektionen, die CODE, DATA, STACK, EXTRA, X1, X2, X3, X4 genannt werden, entsprechen den LINK86-Kommandooptionen. Der Header enthält Informationen wie: die Länge jeder Sektion der CMD-Datei, ihre minimalen und maximalen Speicheranforderungen und ihre Ladeadresse. Diese Informationen werden vom Betriebssystem verwendet, um die Datei korrekt zu laden (siehe Anleitung für den Systemprogrammierer, Steuerprogramm SCPX 1700).

Jede dieser Optionen, die die Sektionen der CMD-Datei beeinflussen, müssen ein oder mehrere in eckige Klammern eingeschlossene Parameter folgen. Tabelle 25 zeigt die Parameter, deren Abkürzung und Bedeutung.

Tabelle 25: Parameter von CMD-Dateioptionen

Parameter	Abkürzung	Bedeutung
ABSOLUTE	AB	absolute Ladeadresse für die Sektion der CMD-Datei
ADDITIONAL	AD	zusätzliche Speicherzuordnung für die Sektion der CMD-Datei
CLASS	C	Klassen, die in eine Sektion der CMD-Datei eingeschlossen werden
GROUP	G	Gruppen, die in eine Sektion der CMD-Datei eingeschlossen werden
MAXIMUM	M	maximale Speicherzuordnung für eine Sektion der CMD-Datei
ORIGIN	O	Anfang des ersten Segmentes in der Sektion der CMD-Datei
SEGMENT	S	Segmente, die in eine Sektion der CMD-Datei eingeschlossen werden

7.7.1. GROUP, CLASS, SEGMENT

Die Parameter GROUP, CLASS und SEGMENT enthalten jeweils eine Liste von Gruppen, Klassen oder Segmenten, die LINK86 in die benannte Sektion der CMD-Datei einordnen soll. Zum Beispiel weist das Kommando

```
A>LINK86 TEST [CODE [SEGMENT [CODE1, CODE2], GROUP [XYZ]]]
```

LINK86 an, die Segmente CODE1, CODE2 und alle Segmente in der Gruppe XZY in die CODE-Sektion der Datei TEST.CMD einzuordnen.

7.7.2. ABSOLUTE, ADDITIONAL, MAXIMUM

Die Parameter ABSOLUTE, ADDITIONAL und MAXIMUM teilen LINK86 die Werte mit, die im Kopfbereich der CMD-Datei einzutragen sind. Diese Parameter haben Vorrang vor den Standardwerten, die LINK86 normalerweise verwendet. Tabelle 26 zeigt die Standardwerte.

Jeder Parameter ist eine in eckige Klammern eingeschlossene Hexadezimalzahl. Der Parameter ABSOLUTE kennzeichnet die absolute Paragraphadresse, an die das Betriebssystem zur Laufzeit die gekennzeichnete Sektion der CMD-Datei lädt. Der Parameter ADDITIONAL kennzeichnet den Betrag des zusätzlichen Hauptspeichers (als Paragraphwert), der von der gekennzeichneten Sektion der CMD-Datei angefordert wird. Das Programm könnte diesen Speicher als Bereich einer Symboltabelle oder für E/A-Puffer nutzen, die zur Laufzeit gebraucht werden, die aber nicht in das Quellprogramm eingefügt werden, und daher nicht in der OBJ-Datei vorhanden sind. Der Parameter MAXIMUM kennzeichnet den maximalen Betrag des Hauptspeichers, den die gekennzeichnete Sektion der

CMD-Datei braucht.

Zum Beispiel erzeugt das Kommando

```
A>LINK86 TEST [DATA[ADD[100], MAX[1000]], CODE[ABS[40]]]
```

die Datei TEST.CMD, deren Header die folgenden Informationen enthält:

- Die DATA-Sektion erfordert mindestens 100H Paragraphen (100H * 10H Byte) zusätzlich zu den Daten in der CMD-Datei.
- Die DATA-Sektion kann bis zu 1000H Paragraphen des Hauptspeichers verwenden (1000H * 10H Byte).
- Die CODE-Sektion muß an die absolute Paragraphadresse 40H geladen werden.

7.7.3. ORIGIN

Der Parameter ORIGIN ist ein hexadezimaler Wert, der den Byte-Offset kennzeichnet, an dem die gekennzeichnete Sektion der CMD-Datei beginnen soll. LINK86 nimmt einen Standardwert von 0 für jede Sektion außer der Datensektion an, welche einen Standardwert von 100H hat, um einen Bereich für die Basisseite zu reservieren (siehe Anleitung für den Systemprogrammierer, Steuerprogramm SCPX 1700).

Tabelle 26 faßt die Standardwerte für alle Kommandooptionen und Parameter zusammen.

Tabelle 26: Standardwerte für Optionen und Parameter einer CMD-Datei

Option	GROUP	CLASS	SEG-MENT	ABSO-LUTE	ADDI-TIONAL	MAXIMUM	ORIGIN
CODE	CGROUP	CODE	CODE	0	0	0	0
DATA	DGROUP	DATA	DATA	0	0	1000H*)	100H
STACK	-	STACK	STACK	0	0	0	0
EXTRA	-	EXTRA	EXTRA	0	0	0	0
X1	-	X1	X1	0	0	0	0
X2	-	X2	X2	0	0	0	0
X3	-	X3	X3	0	0	0	0
X4	-	X4	X4	0	0	0	0

*) Falls es eine DGROUP gibt

7.7.4. FILL/NOFILL

Die Optionen FILL und NOFILL teilen LINK86 mit, was mit nicht initialisierten Daten (reservierten Programmbereichen) zu machen ist, die am Ende einer Sektion der CMD-Datei auftreten können. Die Option FILL weist LINK86 an, diese nicht initialisierten Daten in die CMD-Datei einzubeziehen und sie mit Nullen zu füllen. Die Option NOFILL weist LINK86 an, die nicht initialisierten Daten aus der CMD-Datei wegzulassen. Die Option FILL endet gewöhnlich mit einer größeren Datei, aber die LINK86-Arbeit ist gewöhnlich schneller, wenn FILL gefordert wird. Der Standard ist FILL. Es wird nochmals betont, daß diese Optionen sich nur auf nicht initialisierte Daten am Ende einer Sektion der CMD-Datei beziehen. Nicht initialisierte Daten, die sich nicht am Ende einer Sektion befinden, werden immer mit Null gefüllt und in die CMD-Datei eingeschlossen.

7.8. Optionen der SYM-Datei

Die folgenden Kommandooptionen beeinflussen den Inhalt der SYM-Datei, die LINK86 erzeugt:

- LOCALS
- LIBSYMS
- NOLOCALS
- NOLIBSYMS

Diese Optionen müssen in der Kommandozeile nach der bezeichneten Datei oder Dateien, auf die sie sich beziehen, erscheinen. Sie bleiben wirksam, bis sie geändert werden, und zwar in der Reihenfolge wie LINK86 die Kommandozeile von links nach rechts verarbeitet.

7.8.1. LOCALS/NOLOCALS

Die Option LOCALS weist LINK86 an, lokale Symbole in die SYM-Datei aufzunehmen, wenn sie in den zu linkenden Objektdateien vorhanden sind. Die Option NOLOCALS weist LINK86 an, lokale Symbole in den Objektdateien zu ignorieren. Die Standardoption ist LOCALS.
Zum Beispiel erzeugt das Kommando

```
A>LINK86 TEST1 [NOLOCALS], TEST2 [LOCALS], TEST3
```

eine SYM-Datei, die lokale Symbole aus TEST2.OBJ und TEST3.OBJ aber nicht aus TEST1.OBJ enthält.

7.8.2. LIBSYMS/NOLIBSYMS

Die Option LIBSYMS weist LINK86 an, in der SYM-Datei jene Symbole aufzunehmen, die aus der Bibliothek, die während der Linkarbeit durchsucht wird, kommen. Die Option NOLIBSYMS weist LINK86 an, diese Symbole in die SYM-Datei nicht aufzunehmen. Eine solche Bibliotheksdurchsuche betrifft die Laufzeit-Unterprogrammbibliothek einer höheren Programmiersprache als typischen Fall. Weil die Symbole in einer solchen Bibliothek für den Progammierer gewöhnlich uninteressant sind, ist der Standard NOLIBSYMS.

7.9. Optionen der MAP-Datei

Die Option MAP weist LINK86 an, eine MAP-Datei (Listendatei) zu erzeugen, die Informationen über die Segmente in der CMD-Datei enthält. Der Umfang der Informationen, die LINK86 in die MAP-Datei bringt, wird durch die wahlfreien Parameter

```
OBJMAP      NOOBJMAP
L86MAP      NOL86MAP
ALL
```

gesteuert, die in eckige Klammern eingeschlossen werden und der Optionen MAP folgen. Der Parameter OBJMAP weist LINK86 an, Segmentinformationen über Objektdateien in die MAP-Datei zu bringen. Der Parameter NOOBJMAP unterdrückt diese Informationen. Ähnlich weist der Schalter L86MAP den Linker an, Segmentinformationen von L86-Dateien in die MAP-Datei zu bringen. Der Parameter NOL86MAP unterdrückt diese Informationen. Der Parameter ALL weist LINK86 an, alle Informationen in die MAP-Datei zu bringen.

Sobald LINK86 angewiesen wird, eine MAP-Datei zu erzeugen, können die Parameter zur MAP-Option an verschiedenen Stellen in der Kommandozeile geändert werden. Zum Beispiel:

```
A> LINK86 ACCOUNT [MAP[ALL]], SCREEN, GRAPH.L86 [S,MAP[NOL86MAP]]
```

Wenn die Option MAP ohne Parameter ausgegeben wird, so verwendet LINK86 als Standard OBJMAP und NOL86MAP.

7.10. Optionen der L86-Datei

Die Option SEARCH weist LINK86 an, die vorangestellte Datei zu durchsuchen und in die CMD-Datei nur die Moduln einzufügen, die externe Bezüge aus anderen Moduln befriedigen. Zu beachten ist, daß der Linker L86-Dateien nicht automatisch durchsucht.

Wird die Option SEARCH nicht nach einem Bibliotheksdateinamen verwendet, so fügt LINK86 alle Moduln der Bibliotheksdatei in die zu erzeugende CMD-Datei ein. Zum Beispiel erzeugt das Kommando

```
A>LINK86 TEST1, TEST2, MATH.L86[SEARCH]
```

die Datei TEST1.CMD durch Verbinden der Objektdateien TEST1.OBJ, TEST2.OBJ und jener Moduln aus MATH.L86, auf die sich TEST1.OBJ oder TEST2.OBJ bezieht.

Die Moduln in der Bibliotheksdatei brauchen in keiner besonderen Reihenfolge zu sein. LINK86 führt mehrere Durchläufe durch das Bibliotheksverzeichnis aus, falls versucht wird, Bezüge aus anderen Moduln aufzulösen.

7.11. Optionen der Eingabedatei

Die Option INPUT weist LINK86 an, die Eingabe weiterer Kommandozeilen aus der gekennzeichneten Datei auszuführen. In dieser Kommandozeile dürfen neben der mit INPUT gekennzeichneten Eingabedatei keine anderen Dateien erscheinen. LINK86 beendet die Auswertung der vom Terminal eingegebenen Kommandozeile, wenn er diese Option ermittelt. Kommando-Eingabedateien können nicht mit weiteren Kommando-Eingabedateien geschachtelt werden. Das heißt, eine Kommando-Eingabedatei darf nicht die Option INPUT enthalten.

Die Eingabedatei besteht ebenso aus Dateinamen und Optionen wie eine Kommandozeile, die vom Terminal eingegeben wird. Zum Beispiel soll die Datei TEST.INP folgende Zeilen umfassen:

```
MEMTEST = TEST1, TEST2, TEST3,
IOLIB.L86[S], MATH.L86[S],
TEST4, TEST5[LOCALS]
```

Um LINK86 anzuweisen, diese Datei als Eingabedatei zu verwenden, wird das Kommando

```
A>LINK86 TEST[INPUT]
```

eingegeben. Wenn kein Dateityp für die Eingabedatei angegeben wird, nimmt LINK86 den Dateityp INP an.

7.12. Optionen der E/A-Geräte

Die Option ¤ legt die Geräte der Ursprungs- und der Zieldateien unter LINK86 fest. Die allgemeine Form der ¤-Option ist:

```
¤td
```

wobei t ein Typ und d ein Gerätekennzeichen ist.

LINK86 kennt fünf Typen. Dabei stehen:

- C für Kommandodatei (CMD oder OVR)
- L für Bibliotheksdatei (L86)
- M für Listendatei (MAP)
- O für Objektdatei (OBJ oder L86)
- S für Symboldatei (SYM)

Das Gerätekennzeichen kann ein Buchstabe in der Reihe A bis P, die einen von sechzehn logischen Geräten entsprechen, oder eines der folgenden Sonderzeichen sein:

- X - Terminal
- Y - Drucker
- Z - Nullgerät

Wenn nur einmal das Optionszeichen ¤ verwendet wird, dürfen mehrere td-Zeichenpaare nicht mit Komma getrennt werden. Kommas müssen jedoch verwendet werden, um mehrere ¤-Optionen voneinander abzugrenzen. Zum Beispiel sind die nachfolgenden drei Kommandozeilen gleichwertig:

```
A>LINK86 PART1[¤SZ, ¤OD, ¤LB], PART2
A>LINK86 PART1[¤SZODLB], PART2
A>LINK86 PART1[¤SZ OD LB], PART"
```

Die Bedeutung einer ¤-Option bleibt wirksam, bis sie geändert wird, während LINK86 die Kommandozeile von links nach rechts verarbeitet.

7.12.1. ¤Cd-Gerät der Kommandodatei

LINK86 erzeugt die CMD-Datei auf dem gleichen Gerät wie die erste Objektdatei in der Kommandozeile. Die Option ¤C weist LINK86 an, die CMD-Datei auf das Laufwerk zu bringen, das durch das Zeichen angegeben wird, das dem ¤C folgt, oder die Erzeugung einer Kommandodatei zu unterdrücken, wenn ¤Z angegeben wird.

7.12.2. ¤Ld-Gerät der Bibliotheksdatei

LINK86 durchsucht das Standardgerät nach Laufzeit-Unterprogrammbibliotheken, die automatisch verbunden werden. Die ¤L-Option weist LINK86 an, das angegebene Laufwerk nach diesen Bibliotheksdateien zu durchsuchen.

7.12.3. ¤Md-Gerät der Listendatei

LINK86 erzeugt normalerweise die Listendatei (MAP-Datei) auf dem gleichen Gerät wie die CMD-Datei. Die ¤Md-Option weist LINK86 an, die Listendatei auf das Gerät zu bringen, das dem ¤M folgt. ¤MX wird angegeben, um die Listendatei zum Terminal zu übertragen.

7.12.4. ¤Od-Gerät der Objektdatei

LINK86 sucht nach den OBJ- oder den L86-Dateien, die in der Kommandozeile auf dem Standardlaufwerk angegeben werden, außer jenen Dateien mit einer ausdrücklichen Laufwerksausgabe vor dem Dateinamen. Die ¤O-Option erlaubt, daß die Laufwerkszuordnung mehrerer OBJ- oder L86-Dateien ohne Hinzufügen eines ausdrücklichen Laufwerkskennzeichens für jeden Dateinamen angegeben wird. Zum Beispiel teilt das Kommando

```
A>LINK86 P[¤OD], Q, R, S, T, U.L86, B:V
```

dem Linker mit, daß alle Objektdateien außer dem letzten dem Laufwerk D zugeordnet werden. Das wird nicht auf Dateien angewendet, die automatisch durchsucht werden (siehe Abschnitt 7.12.2.).

7.12.5. ¤Sd-Gerät der Symboldatei

LINK86 erzeugt eine Symboldatei auf dem gleichen Gerät wie die CMD-Datei. Die ¤S-Option weist LINK86 an, die Symboldatei auf das Gerät zu bringen, das durch das dem ¤S folgende Zeichen angegeben wird, oder die Erzeugung einer Symboldatei zu unterdrücken, wenn ¤SZ angegeben wird.

7.13. Fehler in der Kommandozeile

Wenn LINK86 irgendeinen Fehler in der Kommandozeile entdeckt, schreibt er die Mitteilung

SYNTAX ERROR

(Syntaxfehler) aus, wiederholt die Kommandoparameter bis zu der Stelle, wo der Fehler auftritt und läßt dem Fehler ein Fragezeichen folgen.

Zum Beispiel:

```
A>LINK86 a, b, c; d
SYNTAX ERROR
A, B, C;?

A>LINK86 longfilename
SYNTAX ERROR
LONGFILENAME?
```

8. Bibliothekar LIB86

LIB86 ist ein Dienstprogramm zum Erzeugen und zur Wartung von Bibliotheksdateien, die Objektmoduln enthalten. Diese Moduln können durch Sprachübersetzer, wie RASM86 oder durch Compiler, die Moduln im WM86-Objektformat erzeugen, hergestellt worden sein.
LIB86 erzeugt Bibliotheken, kann aber auch Moduln zu einer bereits existierenden Bibliothek hinzufügen, ersetzen, herauslösen oder löschen. LIB86 gibt auch Informationen über den Inhalt einer Bibliotheksdatei aus.

8.1. Operationen

Beim Aufruf von LIB86 werden die angegebenen Dateien eingelesen und es wird eine Bibliotheksdatei, eine Cross-Referenz-Datei oder eine Modul-Listendatei erzeugt, je nachdem wie es in der Kommandozeile angegeben wurde. Nach Abarbeitung des Kommandos wird von LIB86 der prozentuale Ausnutzungsgrad des Hauptspeichers ausgegeben. Die Operationen von LIB86 sind in Bild 10 dargestellt.

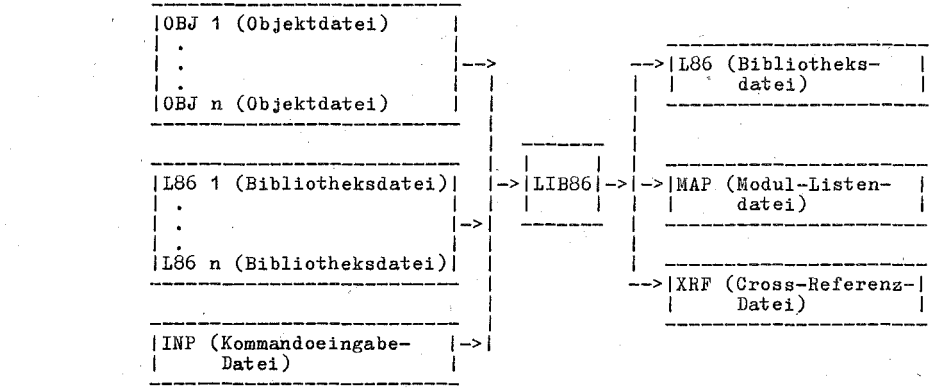


Bild 10: LIB86-Operationen

Tabelle 27 zeigt die von LIB86 benutzten Dateitypen.

Tabelle 27: LIB86-Dateitypen

Typ	Beschreibung
INP	Kommandoeingabe-Datei
L86	Bibliotheksdatei
MAP	Modul-Listendatei
OBJ	Objektdatei
XRF	Cross-Referenz-Datei

8.2. Abbruch

Um LIB86 abzubrechen, genügt es, eine beliebige Terminaltaste zu drücken. LIB86 gibt dann folgende Mitteilung aus:

```
STOP LIB86 (Y/N)?
```

Falls darauf Y eingegeben wird, bricht LIB86 sofort die Arbeit ab und gibt die Steuerung an das Betriebssystem zurück. Bei Eingabe von N setzt LIB86 die Bearbeitung fort.

8.3. Kommandos

Beim Aufruf von LIB86 können wahlweise in der Kommandozeile Parameter angegeben werden, die die Bearbeitung steuern. Tabelle 28 gibt die LIB86-Kommandos wieder. Jedes Kommando kann soweit abgekürzt werden, bis Verwechslungen ausgeschlossen sind. So kann EXTERNALS mit EXTERN, EXT, EX oder einfach mit E abgekürzt werden. Die folgenden Abschnitte beschreiben die Funktion jedes Kommandos.

Tabelle 28: LIB86-Kommandoparameter

Parameter	Benutzung	Abkürzung
DELETE	Löschen von Moduln einer Bibliotheksdatei	D
EXTERNALS	Anzeige von externen Symbolen einer Bibliotheksdatei	E
INPUT	Einlesen von Kommandos aus einer Kommandoeingabe-Datei	I
MAP	Erzeugen einer Modul-Listendatei	MA
MODULES	Anzeige der Moduln einer Bibliotheksdatei	MO
NOALPHA	Anzeige der Moduln in der Reihenfolge ihres Auftretens	N
PUBLICS	Anzeige der globalen Symbole einer Bibliotheksdatei	P
REPLACE	Ersetzung eines Moduls in einer Bibliotheksdatei	R
SEGMENTS	Anzeige der Segmente in einer Bibliotheksdatei	SEG
SELECT	Herauslösen von Moduln aus einer Bibliotheksdatei	SEL
XREF	Erzeugen einer Cross-Referenz-Datei	X

8.4. Erzeugen und Aktualisieren von Bibliotheken

Zum Erzeugen und Aktualisieren von Bibliotheken wird folgende allgemeine Form einer Kommandozeile benutzt:

```
LIB86 library-file = file 1{[switches]}{file 2, ... ,file n}
```

LIB86 erzeugt eine Bibliotheksdatei, die den durch library-file spezifizierten Namen besitzt. Falls kein Dateityp angegeben wurde, erzeugt LIB86 eine Bibliotheksdatei vom Typ L86.

LIB86 liest die durch file 1 bis file n bezeichneten Dateien ein und erzeugt eine Bibliotheksdatei. Falls bei diesen Dateien kein Typ angegeben wurde, setzt LIB86 als Standard den Dateityp OBJ voraus. Die eingelesenen Dateien können einen oder mehrere Moduln enthalten, d. h. es können OBJ- oder L86-Dateien oder eine Kombination davon sein.

Moduln in einer Bibliothek brauchen sich nicht in einer bestimmten Ordnung zu befinden, weil LINK86 die Bibliothek so oft wie nötig durchsucht, um alle Referenzen aufzulösen. LINK86 kann

jedoch mit der Bibliothek viel schneller arbeiten, wenn die Anordnung der darin befindlichen Moduln optimal ist. Dazu müssen soviel Rückwärtsreferenzen wie möglich beseitigt werden (Moduln, welche sich auf globale Symbole beziehen, die in vorhergehenden Moduln der Bibliothek vereinbart wurden), so daß LINK86 die Bibliothek nur einmal durchsuchen muß.

Die Modulnamen werden von den Sprachübersetzern vergeben. Die Methode der Namenszuweisung ist von Übersetzer zu Übersetzer verschieden, im allgemeinen ist es entweder der Dateiname oder der Name des Hauptprogramms.

8.4.1. Erzeugen einer neuen Bibliothek

Beim Erzeugen einer neuen Bibliothek wird zuerst der Name der Bibliothek eingegeben, dann ein Gleichheitszeichen, gefolgt von der Liste der Dateien, die in die Bibliothek mit einbezogen werden sollen und in der Kommandozeile durch Kommas voneinander getrennt sind.

```
A>LIB86 NEWLIB = A, B, C
A>LIB86 NEWLIB.L86 = A.OBJ, B.OBJ, C.OBJ
A>LIB86 MATH = ADD, SUB, MUL, DIV
```

Die ersten beiden Beispiele sind zueinander äquivalent.

8.4.2. Hinzufügen zu einer Bibliothek

Um einen Modul oder mehrere Moduln zu einer bereits existierenden Bibliothek hinzuzufügen, muß der Bibliotheksname auf beiden Seiten des Gleichheitszeichens in der Kommandozeile auftreten. Der Bibliotheksname, der auf der linken Seite steht, ist der Name der Bibliothek, die erzeugt werden soll. Dieser Name tritt auch auf der rechten Seite des Gleichzeitszeichens auf, dort stehen außerdem noch die Namen der Dateien, die hinzugefügt werden sollen. Zum Beispiel:

```
A>LIB86 MATH = MATH.L86, SIN, COS, TAN
A>LIB86 MATH = SQRT, MATH.L86
```

8.4.3. Ersetzen eines Moduls

Durch LIB86 können ein oder mehrere Moduln ersetzt werden, ohne daß die ganze Bibliothek wieder aus den einzelnen Objektdateien neu aufgebaut werden muß. Das Kommando für das Ersetzen von Moduln in einer Bibliothek besitzt die allgemeine Form:

```
LIB86 new-library = old-library [REPLACE [replace-list]]
```

wobei durch new-library der Name der Datei bezeichnet wird, die LIB86 erzeugt, old-library ist der Name der Datei (es kann derselbe wie new-library sein), die den Modul enthält, der ersetzt werden soll. Die replace-list enthält ein oder mehrere Modulnamen in der Form:

```
    module-name = filespec
```

Zum Beispiel bewirkt das Kommando

```
A>LIB86 MATH = MATH.L86 [REPLACE [SQRT = NEWSQRT]]
```

daß LIB86 eine neue Datei MATH.L86 unter Benutzung der alten Datei MATH.L86 als Quelle durch Ersetzen des Moduls SQRT durch die Datei NEWSQRT.OBJ erzeugt. Falls der Name des Moduls, der ersetzt werden soll, derselbe ist, wie der der Datei, die ihn ersetzen soll, so braucht der Name nur einmal angegeben werden. Zum Beispiel ersetzt das Kommando

```
A>LIB86 MATH = MATH.L86 [REPLACE [SQRT]]
```

den Modul SQRT durch die Datei SQRT.OBJ in der Bibliotheksdatei MATH.L86.

Durch ein einzelnes Kommando kann mehrfach ersetzt werden, wobei die Namen durch Kommas getrennt werden. Zum Beispiel:

```
A>LIB86 NEW = MATH.L86 [REPLACE [SIN = NEWSIN, COS = NEWCOS]]
```

Zu beachten ist, daß die Parameter DELETE und SELECT nicht in Verbindung mit REPLACE benutzt werden dürfen.

LIB86 gibt eine Fehlermitteilung aus, wenn es einen der spezifizierten Moduln oder Dateien nicht findet (siehe Anlage 9).

8.4.4. Löschen eines Moduls

Die allgemeine Form zum Löschen eines oder mehrerer Moduln aus einer Bibliothek lautet:

```
LIB86 new-library = old-library [DELETE [module-specifiers]]
```

wobei module specifiers entweder die Namen einzelner Moduln oder Gruppen von Moduln enthält, die durch die Angabe des ersten und des letzten Modulnamens der Gruppe, getrennt durch einen Bindestrich, gekennzeichnet sind. Zum Beispiel:

```
A>LIB86 MATH = MATH.L86 [DELETE [SQRT]]
A>LIB86 MATH = MATH.L86 [DELETE [ADD, SUB, MUL, DIV]]
A>LIB86 MATH = MATH.L86 [DELETE [ADD-DIV]]
```

Die Parameter REPLACE und SELECT dürfen nicht zusammen mit DELETE benutzt werden.

LIB86 gibt eine Fehlermitteilung aus, wenn es einen der spezifizierten Moduln in der Bibliothek nicht findet.

8.4.5. Herauslösen eines Moduls

Das Kommando für das Herauslösen eines oder mehrerer Moduln aus einer Bibliothek lautet in allgemeiner Form:

```
LIB86 new-library = old-library [SELECT [module-specifiers]]
```

wobei module specifiers entweder die Namen einzelner Moduln oder Gruppen von Moduln enthält, die durch die Angabe des ersten und des letzten Modulnamens der Gruppe, getrennt durch einen Bindestrich, gekennzeichnet sind. Zum Beispiel:

```
A>LIB86 ARITH = MATH.L86 [SELECT [ADD, SUB, MUL, DIV]]
A>LIB86 ARITH = MATH.L86 [SELECT [ADD-DIV]]
```

Die Parameter DELETE und REPLACE dürfen nicht zusammen mit SELECT benutzt werden.

LIB86 gibt eine Fehlermitteilung aus, wenn es einen der spezifizierten Moduln in der Bibliothek nicht findet.

8.5. Ausgabe von Bibliotheksinformationen

LIB86 kann auch Informationen über den Inhalt einer Bibliothek ausgeben. LIB86 erzeugt dabei zwei Arten von Listendateien, eine Cross-Referenz-Datei und eine Bibliotheks-Modulliste. Diese Listendateien werden auf dem Standardlaufwerk erzeugt. Durch Angabe der Optionen, die im Abschnitt 8.7. beschrieben sind, können sie aber auch direkt über das Terminal oder den Drucker ausgegeben werden.

8.5.1. Cross-Referenz-Datei

Die Cross-Referenz-Liste einer Bibliothek kann mit folgendem Kommando erzeugt werden:

```
LIB86 library-name [XREF]
```

LIB86 erzeugt die Datei library-name.XRF auf dem Standardlaufwerk oder durch Ändern der Zuweisung auf dem Terminal oder Drucker.

Die Cross-Referenz-Liste enthält eine alphabetische Aufstellung aller globalen, externen und Segmentnamensymbole, die Bestandteil der Bibliothek sind. Nach jedem Symbol folgt eine Liste der Moduln, in denen das Symbol auftritt.

LIB86 markiert dabei den oder die Moduln, in welchen das Symbol definiert wird mit einem Doppelkreuz "#" nach dem Modulnamen. Segmentnamen sind in Schrägstriche eingeschlossen, wie z. B. /CODE/. Am Ende der Cross-Referenz-Liste zeigt LIB86 die Anzahl der Moduln an, die bearbeitet wurden.

8.5.2. Bibliotheks-Modul-Listendatei

Die Modul-Liste einer Bibliothek kann mit dem Kommando

```
LIB86 library-name [MAP]
```

erzeugt werden. LIB86 erzeugt die Datei library-name.MAP auf dem Standardlaufwerk oder durch Ändern der Zuweisung auf dem Terminal oder Drucker.

Die Modul-Liste enthält eine alphabetische Liste der Moduln einer Bibliotheksdatei. Hinter jedem Modul steht dabei eine Aufstellung der Segmente des Moduls und ihrer Länge. In der Modul-Liste ist auch eine Liste der globalen Symbole, die in dem Modul definiert werden und eine Liste der externen Symbole, auf die in dem Modul Bezug genommen wird, enthalten. Am Ende der Modul-Liste zeigt LIB86 die Anzahl der Moduln an, die bearbeitet wurden.

LIB86 listet die Modulnamen alphabetisch auf. Durch den NOALPHA-Schalter kann eine Liste erzeugt werden, in der die

Moduln in der Reihenfolge ihres Auftretens in der Bibliothek stehen. Zum Beispiel:

```
A>LIB86 MATH.L86 [MAP, NOALPHA]
```

8.5.3. Partielle Bibliothekslisten

LIB86 erzeugt partielle Bibliothekslisten, auf zwei Wegen. Erstens kann eine Liste, in der nur Modulnamen, Segmentnamen, Namen von globalen oder externen Symbolen enthalten sind, durch eines der folgenden Kommandos erzeugt werden:

```
LIB86 library-name [MODULES]
LIB86 library-name [SEGMENTS]
LIB86 library-name [PUBLICS]
LIB86 library-name [EXTERNALS]
```

Zweitens kann ein SELECT-Kommando mit einem der listenerzeugenden Kommandos, die oben beschrieben wurden, kombiniert werden. Zum Beispiel:

```
A>LIB86 MATH.L86 [MAP, NOALPHA, SELECT [SIN, COS, TAN]]
```

8.6. LIB86-Kommandos auf Platte

Um die Arbeit mit LIB86 zu vereinfachen, können lange oder häufig benutzte LIB86-Kommandos in einer Plattendatei zusammengefaßt werden. Dann weist eine kurze Kommandozeile LIB86 an, den Rest dieser Kommandozeile von der Platte einzulesen. Diese Datei kann eine Anzahl von Zeilen enthalten, die aus den Namen der zu bearbeitenden Dateien und aus LIB86-Kommandoparametern bestehen. Das letzte Zeichen in dieser Datei muß ein normales Dateiendezeichen sein (1AH).

Zur Steuerung des Einlesens eines Kommandos aus einer Plattendatei wird ein Kommando der allgemeinen Form:

```
LIB86 filespec [INPUT]
```

benutzt.

Wenn filespec keinen Dateityp enthält, setzt LIB86 den Dateityp INP voraus.
Zum Beispiel könnte die Datei MATH.INP das folgende enthalten:

```
MATH = ADD [¤OC], SUB, MUL, DIV,
SIN, COS, TAN,
SQRT, LOG
```

Dann weist das Kommando

```
A>LIB86 MATH [INPUT]
```

LIB86 an, die Datei MATH.INP wie eine Kommandozeile einzulesen. Andere Kommandos können zusammen mit INPUT eingegeben werden, aber es dürfen keine anderen Dateinamen in der Kommandozeile nach der INP-Datei auftreten. Zum Beispiel:

```
A>LIB86 MATH [INPUT, XREF, MAP]
```

8.7. Ein- und Ausgabesteuerung

LIB86 setzt voraus, daß sich alle zu bearbeitenden Dateien auf dem Standardlaufwerk befinden, deshalb muß die Laufwerksangabe jeder Datei, die sich nicht auf dem Standardlaufwerk befindet, spezifiziert werden. LIB86 erzeugt eine Datei auf dem Standardlaufwerk, ohne daß ein Laufwerk angegeben werden muß. Zum Beispiel:

```
A>LIB86 E:MATH = MATH.L86, D:SIN, D:COS, D:TAN
```

LIB86 erzeugt auch die MAP- und XRF-Dateien auf demselben Laufwerk wie die L86-Datei oder auf demselben Laufwerk, auf dem sich die erste Objektdatei befindet, wenn keine Bibliothek erzeugt wird.

Durch folgende Kommandooptionen können die LIB86-Standardannahmen überschrieben werden:

```
¤Md - MAP-Dateiausgabelaufwerk
¤Od - OBJ- oder L86-Quelldateilaufwerk
¤Xd - XRF-Dateiausgabelaufwerk
```

wobei d eine Laufwerksangabe (A-P) ist. Für MAP- und XRF-Dateien kann für d auch X oder Y eingesetzt werden, wobei jeweils das Terminal oder der Drucker zugewiesen werden. Nach dem Dollarzeichen können auch mehrere Ein- und Ausgabevereinbarungen stehen. Zum Beispiel:

```
A>LIB86 TRIG [MAP, XREF, ¤OCMYXY] = SIN, COS, TAN
```

Der ¤O-Schalter gilt während LIB86 die Kommandozeile von links nach rechts abarbeitet solange, bis ein neuer ¤O-Schalter angegeben wird. Diese Eigenschaft kann vorteilhaft benutzt werden, wenn eine Bibliothek aus mehreren Dateien erzeugt werden soll, von denen sich ein Teil auf dem einen Laufwerk und der andere Teil auf einem anderen befindet. Zum Beispiel:

```
A>LIB86 BIGLIB = A1[¤OC], A2, ... , A50[¤OD], A51, ... , A100
```

Anlage 1 Mnemonikunterschiede zum BOS1810-Assembler

Der Assembler RASM86 benutzt die gleichen Befehle wie der Assembler des BOS1810, ausgenommen Mnemoniks für Fern- und Kurzsprünge, Rufe und Rückkehr.
Die folgende Tabelle zeigt die vier Unterschiede:

Tabelle 29: Mnemonikunterschiede

Mnemonikfunktion	SCP 1700	BOS1810
Kurzsprung innerhalb eines Segmentes:	JMPS	JMP
Sprung zwischen Segmenten:	JMPF	JMP
Rückkehr zwischen Segmenten:	RETF	RET
Ruf zwischen Segmenten:	CALLF	CALL

Anlage 2 Reservierte Worte

Tabelle 30: Reservierte Worte

Vordefinierte Zahlen				
BYTE	WORD	DWORD		
Operatoren				
EQ	GE	GT	LE	LT
NE	OR	AND	MOD	NOT
PTR	SEG	SHL	SHR	XOR
LAST	TYPE	LENGTH	OFFSET	
Assembler-Direktiven				
DB	DD	DW	IF	RS
RB	RW	END	ENDM	EQU
ORG	CSEG	DSEG	ESEG	SSEG
EJECT	ENDIF	TITLE	LIST	NOLIST
INCLUDE	SIMFORM	PAGESIZE	CODEMACRO	PAGEWIDTH
ELSE	EXTERN	GROUP	IFLIST	NAME
NOIFLIST	PUBLIC	RD		
Code-Makro-Direktiven				
DB	DD	DW	DBIT	RELB
RELW	MODRM	SEGFIX	NOSEGFIX	
Register				
AH	AL	AX	BH	BL
BP	BX	CH	CL	CS
CX	DH	DI	DL	DS
DX	ES	SI	SP	SS
Standard-Segment-Namen				
CODE	DATA	EXTRA	STACK	
Segment-Beschreibung				
BYTE	LOCAL	PARA	STACK	COMMON
PAGE	PUBLIC	WORD		
Externe Beschreibung				
ABS	DWORD	NEAR	BYTE	FAR
WORD				
Befehlsmnemoniks -- siehe Anlage 3				

Anlage 3 Zusammenfassung der Befehle von RASM86

Tabelle 31: Zusammenfassung der Befehle von RASM86

Mnemonik	Beschreibung	Abschnitt
AAA	ASCII adjust for Addition	4.3.
AAD	ASCII adjust for Division	4.3.
AAM	ASCII adjust for Multiplikation	4.3.
AAS	ASCII adjust for Subtraction	4.3.
ADC	Add with Carry	4.3.
ADD	Add	4.3.
AND	And	4.3.
CALL	Call (intra-segment)	4.5.
CALLF	Call (inter-segment)	4.5.
CBW	Convert Byte to Word	4.3.
CLC	Clear Carry	4.6.
CLD	Clear Direction	4.6.
CLI	Clear Interrupt	4.6.
CMC	Completment Carry	4.6.
CMP	Compare	4.3.
CMPS	Compare Byte or Word (of string)	4.4.
CMPSB	Compare Byte (of string)	4.4.
CMPSW	Compare Word (of string)	4.4.
CWD	Convert Word to Double Word	4.3.
DAA	Decimal Adjust for Addition	4.3.
DAS	Decimal Adjust for Subtraction	4.3.
DEC	Decrement	4.3.
DIV	Divide	4.3.
ESC	Escape	4.6.
HLT	Halt	4.6.
IDIV	Integer Divide	4.3.
IMUL	Integer Multiply	4.3.
IN	Input Byte or Word	4.2.
INC	Increment	4.3.
INT	Interrupt	4.5.
INTO	Interrupt on Overflow	4.5.
IRET	Interrupt Return	4.5.
JA	Jump on Above	4.5.
JAE	Jump on Above or Equal	4.5.
JB	Jump on Below	4.5.
JBE	Jump on Below or Equal	4.5.
JC	Jump on Carry	4.5.
JCXZ	Jump on CX Zero	4.5.
JE	Jump on Equal	4.5.
JG	Jump on Greater	4.5.
JGE	Jump on Greater or Equal	4.5.
JL	Jump on Less	4.5.
JLE	Jump on Less or Equal	4.5.
JMP	Jump (intra-segment)	4.5.
JMPF	Jump (inter-segment)	4.5.
JMPS	Jump (8-bit-displacement)	4.5.
JNA	Jump on Not Above	4.5.
JNAE	Jump on Not Above or Equal	4.5.
JNB	Jump on Not Below	4.5.

Tabelle 31: (Fortsezung)

Mnemonik	Beschreibung	Abschnitt
JNBE	Jump on Not Below or Equal	4.5.
JNC	Jump on Not Carry	4.5.
JNE	Jump on Not Equal	4.5.
JNG	Jump on Not Greater	4.5.
JNGE	Jump on Not Greater or Equal	4.5.
JNL	Jump on Not Less	4.5.
JNLE	Jump on Not Less or Equal	4.5.
JNO	Jump on Not Overflow	4.5.
JNP	Jump on Not Parity	4.5.
JNS	Jump on Not Sign	4.5.
JNZ	Jump on Not Zero	4.5.
JO	Jump on Overflow	4.5.
JP	Jump on Parity	4.5.
JPE	Jump on Parity Even	4.5.
JPO	Jump on Parity Odd	4.5.
JS	Jump on Sign	4.5.
JZ	Jump on Zero	4.5.
LAHF	Load AH with Flags	4.2.
LDS	Load Pointer into DS	4.2.
LEA	Load Effective Address	4.2.
LES	Load pointer into ES	4.2.
LOCK	Lock bus	4.6.
LODS	Load Byte or Word (of string)	4.4.
LODSB	Load Byte (of string)	4.4.
LODSW	Load Word (of string)	4.4.
LOOP	Loop	4.5.
LOOPE	Loop While Equal	4.5.
LOOPNE	Loop While Not Equal	4.5.
LOOPNZ	Loop While Not Zero	4.5.
LOOPZ	Loop While Zero	4.5.
MOV	Move	4.3.
MOVS	Move Byte or Word (of sring)	4.4.
MOVSB	Move Byte (of string)	4.4.
MOVSW	Move Word (of string)	4.4.
MUL	Multiply	4.3.
NEG	Negate	4.3.
NOP	No Operation	4.6.
NOT	Not	4.3.
OR	Or	4.3.
OUT	Output Byte or Word	4.2.
POP	Pop	4.2.
POPF	Pop Flags	4.2.
PUSH	Push	4.2.
PUSHF	Push Flags	4.2.
RCL	Rotate through Carry Left	4.3.
RCR	Rotate through Carry Right	4.3.
REP	Repeat	4.4.
REPE	Repeat while Equal	4.4.
REPNE	Repeat while Not Equal	4.4.

Tabelle 31: (Fortsetzung)

Mnemonik	Beschreibung	Abschnitt
REPNZ	Repaet while Not Zero	4.4.
REPZ	Repaet while Zero	4.4.
RET	Return (intra-segment)	4.5.
RETF	Return (inter-segment)	4.5.
ROL	Rotate Left	4.3.
ROR	Rotate Right	4.3.
SAHF	Store AH into Flags	4.2.
SAL	Shift Arithmetic Left	4.3.
SAR	Shift Arithmetic Right	4.3.
SBB	Subtract with Borrow	4.3.
SCAS	Scan Byte or Word (of string)	4.4.
SCASB	Scan Byte (of string)	4.4.
SCASW	Scan Word (of string	4.4.
SHL	Shift Left	4.3.
SHR	Shift Right	4.3.
STC	Set Carry	4.6.
STD	Set Direction	4.6.
STI	Set Interrupt	4.6.
STOS	Store Byte or Word (of string)	4.4.
STOSB	Store Byte (of string)	4.4.
STOSW	Store Word (of string)	4.4.
SUB	Subtract	4.3.
TEST	Test	4.3.
WAIT	Wait	4.6.
XCHG	Exchange	4.2.
XLAT	Translate	4.2.
XOR	Exclusive Or	4.3.

Anlage 4 Code-Makro-Definitionssyntax

```
codemacro ::= CODEMACRO name [formal-list]
              [list-of-macro-directives]
              ENDM

name ::= IDENTIFIER

formal-list ::= parameter-description[{,parameter-description}]

parameter-description ::= formal-name :specifier-letter
                          [modifier-letter] [(range)]

specifier-letter ::= A | C | D | E | M | R | S | X

modifier-letter ::= b | w | d | sb

range ::= single-range|double-range

single-range ::= REGISTER | NUMBERB

double-range ::= NUMBERB,NUMBERB | NUMBERB,REGISTER |
                 REGISTER,NUMBERB | REGISTER,REGISTER

list-of-macro-directives ::= macro-directive
                             {macro-directive}

macro-directive ::= db | dw | dd | segfix |
                    nosegfix | modrm | relb | relw | dbit

db ::= DB NUMBERB | DB formal-name

dw ::= DW NUMBERW | DW formal-name

dd ::= DD formal-name

segfix ::= SEGFIX formal-name
```

```
nosegfix ::= NOSEGFIX formal-name

modrm ::= MODRM NUMBER7,formal-name|
          MODRM formal-name , formal-name

relb ::= RELB formal-name

relw ::= RELW formal-name

dbit ::= DBIT field-description{,field-description}

field-description ::= NUMBER15 ( NUMBERB) |
                      NUMBER15 ( formal-name ( NUMBERB ))

formal-name ::= IDENTIFIER
```

NUMBERB ist ein 8-bit-Wert
NUMBERW ist ein 16-bit-Wert
NUMBER7 sind Werte von 0,1, ..., 7
NUMBER15 sind Werte von 0,1, ..., 15

Anlage 5 Einfaches Beispielprogramm


```
                        TITLE 'TERMINAL INPUT/OUTPUT'
                        PAGESIZE 50
                        PAGEWIDTH 79
                        SIMFORM
                        ;
                        ; *** Terminal I/O subroutines ***
                        ;
                        ; The following subroutines are included:
                        ;
                        ;
                        ; CONSTAT - console status
                        ; CONIN    - console input
                        ; CONOUT   - console output
                        ;
                        ; Each routine requires CONSOLE NUMBER
                        ; in the BL - register
                        ;
                        ; ***           ***
                        ; *  JUMP TABLE:  *
                        ; ***           ***
                        ;
                        CSEG            ;start of code-segment
                        ;
                        JMP-TAB:
0000 E90600     0009            JMP   CONSTAT
0003 E91C00     0022            JMP   CONIN
0006 E92F00     003B            JMP   CONOUT
                        ;
                        ; ***                 ***
                        ; *  I/O PORT NUMBERS   *
                        ; ***                 ***
                        ;
                        ; Terminal 1:
                        ;
```

```
0010                    INSTAT1         EQU  10H  ; input status port
0011                    INDATA1         EQU  11H  ; input port
0011                    OUTDATA1        EQU  11H  ; output port
0001                    READYINMASK1    EQU  01H  ; input ready mask
0002                    READYOUTMASK1   EQU  02H  ; output ready mask
                        ;
                        ; Terminal 2:
                        ;
0012                    INSTAT2         EQU  12H  ; input status port
0013                    INDATA2         EQU  13H  ; input port
0013                    OUTDATA2        EQU  13H  ; output port
0004                    READYINMASK2    EQU  04H  ; input ready mask
0008                    READYOUTMASK2   EQU  08H  ; output ready mask
                        ;
                        ; ***       ***
                        ; * CONSTAT   *
                        ; ***       ***
                        ;
                        ; Entry: BL-reg = terminal no
                        ; Exit:  AL-reg = 0 if not ready
                        ;                  0FFH if ready
                        ;
                        CONSTAT:
0009 53E84700    0054           PUSH BX ! CALL OKTERMINAL !
                        CONSTAT1:
000D 52                         PUSH DX
000E B600                       MOV  DH,0          ; read status port
```

```
0010 BA970000   R                       MOV  DL,INSTATUSTAB [BX]
0014 EC                                 IN   AL,DX
0015 22870600   R                       AND  AL,READYINMASKTAB [BX]
0019 7402         001D                  JZ   CONSTATOUT
001B B0FF                               MOV  AL,0FFH
                                CONSTATOUT:
001D 5A5B0AC0C3                         POP DX ! POP BX ! OR AL,AL ! RET
                                ;
                                ; ***     ***
                                ; *  CONIN  *
                                ; ***     ***
                                ;
                                ; Entry: BL-reg = terminal no
                                ; Exit:  AL-reg = read character
                                ;
0022 53E82E00     0054          CONIN: PUSH BX ! CALL OKTERMINAL !
0026 E8E4FF       000D          CONIN1:CALL CONSTAT1          ; test status
                  000D
0029 74FB         0026                  JZ   CONIN1
002B 52                                 PUSH DX               ; read character
002C B600                               MOV  DH,0
002E 8A970200   R                       MOV  DL,INDATATAB [BX]
0032 EC                                 IN   AL,DX
0033 247F                               AND  AL,7FH           ; strip parity bit
0035 5A5BC3                             POP DX ! POP BX ! RET
                                ;
                                ; ***      ***
                                ; *  CONOUT  *
                                ; ***      ***
                                ;
                                ; Entry: BL-reg = Terminal no
```

```
                                ;
                                ;       AL-reg = character to print
                                ;
0038 53E81800      0054         CONOUT:  PUSH BX ! CALL OKTERMINAL
003C 52                                 PUSH DX
003D 50                                 PUSH AX
003E B600                               MOV  DH,0               ; test status
0040 8A970000    R                      MOV  DL,INSTATUSTAB [BX]
                                CONOUT1:
0044 EC                                 IN   AL,DX
0045 22870800    R                      AND  AL,READYOUTMASKTAB [BX]
0049 74F9          0004                 JZ   CONOUT1
004B 58                                 POP  AX
004C BA970400    R                      MOV  DL,OUTDATATAB [BX]
0050 EE                                 OUT  DX,AL
0051 5A5BC3                             POP DX ! POP BX ! RET
                                ;
                                ; ***          ***
                                ; *  OKTERMINAL  *
                                ; ***          ***
                                ;
                                ; Entry: BL-reg = Terminal no
                                ;
                                OKTERMINAL:
0054 0ADB                               OR   BL,BL
0056 740A          0062                 JZ   ERROR
0058 80FB03                             CMP  BL,LENGTH INSTATUSTAB + 1
005B 7305          0062                 JAE  ERROR
005D FECB                               DEC  BL
005F B700                               MOV  BH,0
0061 C3                                 RET
0062 5B5BC3                     ERROR:  POP BX ! POP BX ! RET  ; do nothing
```


C 1014-0003-1 M 3030

```
                                ;
                                ; *** end of Code-Segment ***
                                ;

                                ;
                                ; *** Data-Segment ***
                                ;

                                        DSEG

                                ;
                                ; *** Data for each Terminal ***
                                ;
                                ;
0000 1012                       INSTATUSTAB      DB   INSTAT1,INSTAT2
0002 1113                       INDATATAB        DB   INDATA1,INDATA2
0004 1113                       OUTDATATAB       DB   OUTDATA1,OUTDATA2
0008 0104                       READYINMASKTAB   DB   READYINMASK1,READYINMASK2
000B 0208                       READYOUTMASKTAB  DB   READYOUTMASK1,READYOUTMASK2
                                ;
                                ; ***  end of file  ***
                                END

END OF ASSEMBLY.  NUMBER OF ERRORS:  0  USE FACTOR:  3%
```

Anlage 6 RASM86-Fehlermitteilungen

RASM86 liefert zwei Fehlertypen:

1. Fehler mit nachfolgendem Abbruch der Übersetzung.

   NO FILE
   DISK FULL
   DIRECTORY FULL
   DISK READ ERROR
   CANNOT CLOSE
   SYMBOL TABLE OVERFLOW
   SYNTAX ERROR

2. Fehler, die während der Analyse der Assemblerzeile auftreten. Diese Fehler werden durch Ausgabe einer Nummer vor der Assemblerzeile gekennzeichnet. Wenn eine Zeile mehr als einen Fehler enthält, so wird nur der erste Fehler angezeigt. Tabelle 32 führt die RASM86-Fehlermitteilungen auf.

Tabelle 32: RASM86-Fehlermitteilungen

Nummer	Bedeutung
0	ILLEGAL FIRST ITEM (ungültiges erstes Element auf Quellzeile)
1	MISSING PSEUDO INSTRUKTION (es fehlt der Pseudobefehl)
2	ILLEGAL PSEUDO INSTRUCTION (unerlaubter Pseudobefehl)
3	DOUBLE DEFINED VARIABLE (mehrfach definierte Variable)
4	DOUBLE DEFINED LABEL (mehrfach definierte Marke)
5	UNDEFINED INSTRUCTION (undefinierter Befehl)
6	GARBAGE AT END OF LEINE - IGNORED (unklares Zeilenende - wird ignoriert)
7	OPERAND(S) MISMATCH INSTRUCTION (Operand paßt nicht zum Befehl)
8	ILLEGAL INSTRUCTION OPERANDS (unerlaubte Befehlsoperanden)
9	MISSING INSTRUCTION (Befehl nicht vorhanden)

Tabelle 32: (Fortsetzung)

Nummer	Bedeutung
10	UNDEFINED ELEMENT OF EXPRESSION (undefiniertes Ausdruckselement)
11	ILLEGAL PSEUDO OPERAND (unerlaubter Pseudooperand)
12	NESTED IF ILLEGAL - IF IGNORED (maximale Verschachtelung mit IF überschritten)
13	ILLEGAL IF OPERAND - IF IGNORED (unerlaubter IF Operand)
14	NO MATCHING IF FOR ENDIF (keine Übereinstimmung zwischen IF und ENDIF)
15	SYMBOL ILLEGALY FORWARD REFERENCED - NEGLECTED (unerlaubte Vorwärtsreferenz in ORG, RS, EQU oder IF)
16	DOUBLE DEFINED SYMBOL - TREATED AS UNDEFINED (mehrfach definiertes Symbol)
17	INSTRUCTION NOT IN CODESEGMENT (Befehl nicht im Code-Segment)
18	FILE NAME SYNTAX ERROR (Dateiname in INCLUDE-Direktive syntaktisch falsch)
19	NESTED INCLUDE NOT ALLOWED (Geschachtelte Einbindung nicht erlaubt)
20	ILLEGAL EXPRESSION ELEMENT (unerlaubtes Ausdruckselement)
21	MISSING TYPE INFORMATION IN OPERAND(S) (Typangabe im Operand fehlt)
22	LABEL OUT OF RANGE (Marke außerhalb der zulässigen Grenze)
23	MISSING SEGMENT INFORMATION IN OPERAND (Segmentangabe im Operand fehlt)
24	ERROR IN CODEMACROBUILDING (Fehler bei der Code-Makro-Bildung)
25	NO MATCHING IF FOR ELSE (ELSE-Anweisung paßt nicht zur IF-Anweisung)

Anlage 7 XREF86-Fehlermitteilungen

Tabelle 33: XREF86-Fehlermitteilungen

Fehlermitteilung	Bedeutung
CANNOT CLOSE	XREF86 kann die Ausgabedatei nicht abschließen. (Platte im Laufwerk überprüfen, darf nicht schreibgeschützt sein)
DIRECTORY FULL	kein Platz mehr für Directory der Ausgabedatei (andere Dateien löschen oder andere Platte verwenden)
DISK FULL	Platte voll (nicht mehr benötigte Dateien löschen oder andere Platte verwenden)
NO FILE	XREF86 kann die angegebene Datei nicht finden
SYMBOL FILE ERROR	ungültige .SYM-Datei (eine Zeile ist nicht mit CR/LF abgeschlossen)
SYMBOL TABLE OVERFLOW	Symboltabelle ist voll (reduzieren der Anzahl oder der Länge der Symbole im Programm oder System mit mehr Speicherplatz verwenden)

Anlage 8 LINK86-Fehlermitteilungen

LINK86 kann während des Arbeitsablaufes Fehlermitteilungen ausgeben. Die Fehlermitteilungen und eine kurze Erklärung über deren Ursache werden in der Tabelle 34 beschrieben.

Tabelle 34: LINK86-Fehlermitteilungen

Mitteilung	Bedeutung
CANNOT CLOSE	LINK86 kann eine Ausgabedatei nicht schließen. Nach dieser Unterbrechung sollten geeignete Schritte unternommen werden, um zu sehen, ob die richtige Platte im Laufwerk ist und daß die Platte nicht schreibgeschützt ist.
DIRECTORY FULL	Verzeichnis voll Es gibt nicht genug Verzeichnisplatz für die Ausgabedateien. Man sollte entweder unnötige Dateien löschen oder eine andere Platte mit mehr Verzeichnisplatz nehmen und LINK86 erneut starten.
DISK READ ERROR	Plattenfehler LINK86 kann eine Quell- oder Objektdatei nicht richtig lesen. Das ist gewöhlich die Folge einer unerwarteten Dateiendemarkierung. Das Problem ist in der Quelldatei zu klären.
DISK WRITE ERROR	Plattenschreibfehler Eine Datei kann nicht ordentlich geschrieben werden, vielleicht auf Grund einer vollen Platte.
MULTIPLE DEFINITION	Mehrfachdefinition Das angegebene Symbol wird als PUBLIC-Symbol in mehr als einem Modul definiert. Das Problem ist in der Quelldatei zu klären.
NO FILE	keine Datei LINK86 kann die angegebene Quell- oder Objektdatei auf dem angegeben Laufwerk nicht finden.
OBJECTFILE ERROR	Objektdatei-Fehler LINK86 entdeckte einen Fehler in der Objektdatei auf dem angegebenen Gerät.

Tabelle 34: (Fortsetzung)

Mitteilung	Bedeutung
SEGMENT ATTRIBUTE ERROR	Segmentattribut-Fehler Der Zuordnungs- oder Verbindetyp des angegebenen Segments ist nicht der gleiche wie der Typ des Segments in einer vorher gelinkten Datei. Die Objektdatei ist erneut herzustellen, nachdem die Segmentattribute entsprechend der Notwendigkeit geändert wurden.
SEGMENT COMBINATION ERROR	Segment-Verbindefehler Ein Versuch wird gemacht, Segmente zu verbinden, die nicht verbunden werden können, beispielsweise LOCAL-Segmente. Die Segmentattribute sind zu ändern, bevor erneut gelinkt wird.
SYMBOL TABLE OVERFLOW	Symboltabellen-Überlauf LINK86 hat ungenügend Symboltabellenplatz. Es sind entweder die Anzahl oder die Länge der Symbole im Programm zu verringern oder es ist auf einem System mit mehr verfügbaren Speicher erneut zu linken.
SYNTAX ERROR	Syntaxfehler LINK86 entdeckte einen Syntaxfehler in der Kommandozeile, vielleicht durch einen fehlerhaften Dateinamen oder einen ungültigen Dateinamen oder eine ungültige Kommandooption. LINK86 wiederholt die Kommandozeile bis zu der Stelle, an der der Fehler gefunden wurde. Die Kommandozeile ist zu wiederholen, oder die INP-Datei ist zu verbessern.
TARGET OUT OF RANGE	Das Ziel einer Festlegung kann nicht vom Ort der Festlegung erreicht werden.
UNDFINED SYMBOLS	undefinierte Symbole Die dieser Mitteilung folgenden Symbole haben einen Bezug, sie sind aber in keinem der zu linkenden Moduln definiert.

Anlage 9 LIB86-Fehlermitteilungen

LIB86 kann während der Abarbeitung folgende Fehlerausschriften ausgeben. Zu jeder Mitteilung gibt LIB86 als Hilfe bei der Fehlersuche weitere Informationen zu diesem Fehler aus, wie den Datei- oder Modulnamen.

Tabelle 35: LIB86-Fehlermitteilungen

Mitteilung	Bedeutung
CANNOT CLOSE	LIB86 kann die Ausgabedatei nicht schließen. Es sollte geprüft werden, ob sich die richtige Platte im Laufwerk befindet und ob die Platte nicht schreibgeschützt ist.
DIRECTORY FULL	Es ist nicht mehr genügend Platz im Dateiverzeichnis vorhanden. Durch Löschen einiger unwichtiger Dateien oder Einlegen einer anderen Platte, auf der noch freier Speicherplatz für das Dateiverzeichnis ist, kann dieser Fehler beseitigt werden. Danach kann LIB86 nochmals aufgerufen werden.
DISK FULL	Es ist nicht mehr genügend Speicherplatz für die Ausgabedateien vorhanden. Durch Löschen einiger unnütiger Dateien oder Einlegen einer Platte mit mehr freiem Speicherplatz und Neustart von LIB86 kann dieser Fehler beseitigt werden.
DISK READ ERROR	LIB86 kann eine Quell- oder Objektdatei nicht einlesen. Das ist meistens das Ergebnis eines unerwarteten Dateiendes. Dazu muß die Quelldatei korrigiert werden.
INVALID COMMAND OPTION	LIB86 hat einen unbekannten Parameter in der Kommandozeile gefunden. Die Kommandozeile oder die INP-Datei ist zu korrigieren.

Tabelle 35: (Fortsetzung)

Mitteilung	Bedeutung
MODULE NOT FOUND	Der beim REPLACE-, SELECT- oder DELETE-Parameter angegebene Modulname kann nicht gefunden werden. Die Kommandozeile oder die INP-Datei ist zu korrigieren.
MULTIPLE DEFINITION	Das angegebene Symbol ist als globales Symbol in mehr als einem Modul definiert. Der Fehler ist in der Quelldatei zu beheben.
NO FILE	LIB86 findet die angegebene Datei nicht.
OBJEKT FILE ERROR	LIB86 findet einen Fehler in der Objektdatei. Dieser kann durch einen Übersetzungfehler oder eine unkorrekte Plattendatei zustande gekommen sein. Die Datei muß neu übersetzt werden.
RENAME ERROR	LIB86 kann eine Datei nicht umbenennen. Zu überprüfen ist, ob die Platte schreibgeschützt ist.
SYMBOL TABLE OVERFLOW	Es ist nicht genügend Speicherplatz für die Symboltabelle vorhanden. Die Anzahl der Parameter in der Kommandozeile ist zu verringern (MAP und XREF benötigen beide Speicher für eine Symboltabelle), oder ein System mit größerem Speicher ist zu benutzen.
SYNTAX ERROR	LIB86 findet in der Kommandozeile einen Syntaxfehler, der wahrscheinlich durch einen falschen Dateinamen oder durch einen falschen Parameter verursacht wurde. LIB86 gibt die Kommandozeile bis zu dem Punkt aus, an dem der Fehler gefunden wurde. Die Kommandozeile oder die INP-Datei ist zu korrigieren.
VERSION 2 REQUIRED	LIB86 fordert die Betriebssystemversion 2 oder eine spätere Version.

Anlage 10 WM87-Befehlssatz

Mit dem Assembler RASM86 des SCP 1700 ist es möglich, alle Befehle des Numerikprozessors WM87 zu übersetzen. Dabei sind einige Besonderheiten zu beachten. Es ist mit RASM86 nicht möglich, durch Analyse der Operanden deren Länge (short-real oder long-real) festzustellen, um daraus den erforderlichen Operationscode zu bilden. Deshalb ist diese Information mit in der mnemonischen Darstellung der Gleitkommabefehle enthalten (16, 32, 64 und 80 Bit). Weiterhin sind Zugriffe zu Speicheroperanden über "WORD POINTER" und "BYTE POINTER" möglich. Ausführliche Informationen über den WM87-Befehlssatz sowie die Funktionsweise des WM87-Numerikprozessors sind in dem Teil "Sprachbeschreibung Assembler ASM86" der "Anleitung für den Programmierer" des Betriebssystems BOS1810 enthalten.

Befehle zur Datenübertragung

Es gibt drei Arten von Operationen zur Datenübertragung:

- Übertragung von Real-Zahlen
  - FLD Load real
  - FST Store real
  - FSTP Store real and pop
  - FXCH Exchange registers
- Übertragung von Integer-Zahlen
  - FILD Integer load
  - FIST Integer store
  - FISTP Integer store and pop
- Übertragung von BCD-Zahlen
  - FBLD Packed decimal (BCD) load
  - FBSTP Packed decimal (BCD) store and pop

Diese Befehle transportieren Operanden zwischen Gleitkomma-Stack-Elementen oder dem Stack-Top und dem Speicher. Jede der sieben Datenarten kann in einer einfachen Operation in das Format temporary-real konvertiert und in den Stack geladen werden, in gleicher Weise können sie in den Speicher geladen werden. Diese Befehle aktualisieren automatisch das Tag-Wort des Prozessors, um den nach der Befehlsausführung bestehenden Stack-Zustand widerzuspiegeln.

Jede der sieben Datenarten wird beim Laden in den Stack in das Format temporary-real konvertiert. In gleicher Weise werden sie in den Speicher gebracht.

Tabelle 36: Befehle zur Datenübertragung

Syntax		Ergebnis
FLD	ST1	Kellern des Stack-Elementes ST1 nach Stack-Top ST
FLD32	M	Kellern des Speicheroperanden M (4 Byte) nach Stack-Top ST
FLD64	M	Kellern des Speicheroperanden M (8 Byte) nach Stack-Top ST
FLD80	M	Kellern des temporären Speicheroperanden M Byte) nach Stack-Top ST
FST	ST1	Übertragung des Stack-Top ST nach dem Stack-Element ST1
FST32	M	Übertragung des Stack-Top ST (4 Byte) nach dem Speicheroperanden M
FST64	M	Übertragung des Stack-Top ST (8 Byte) nach dem Speicheroperanden M
FSTP	ST1	Übertragung des Stack-Top ST nach Stack-Element ST1 und Stack-Pop
FST32P	M	Übertragung des Stack-Top ST (4 Byte) nach dem Speicheroperanden M und Stack-Pop
FST64P	M	Übertragung des Stack-Top ST (8 Byte) nach dem Speicheroperanden M und Stack-Pop
FST80P	M	Übertragung des Stack-Top ST (10 Byte) nach dem Speicheroperanden M (temporary-real) und Stack-Pop
FXCH		entspricht FXCH ST1
FXCH	ST1	Stack-Element ST1 mit Stack-Top wechseln
FILD16	M	Laden eines Integer-Wortes in den Stack-Top
FILD32	M	Laden eines langen Integer-Wortes (4 Byte) in den Stack-Top
FILD64	M	Laden eines langen Integer-Wortes (8 Byte) in den Stack-Top
FIST16	M	Speichern des Stack-Top als Integer-Wort nach Speicheroperanden M
FIST32	M	Speichern des Stack-Top als Integer-Wort (lang) nach Speicheropanden M
FIST16P	M	Speichern des Stack-Top als Integer-Wort nach Speicheroperanden M und Stack-Pop
FIST32P	M	Speichern des Stack-Top als Integer-Wort (lang) nach Speicheroperanden M und Stack-Pop
FBLD	M	Kellern des BCD-Speicheroperanden M nach Stack-Top ST
FBSTP	M	Kellern des BCD-Speicheroperanden M nach Stack-Top ST und Stack-Pop

Arithmetikbefehle

Es gibt fünf Arten von Arithmetikbefehlen:

- Addition
  - FADD — Add real
  - FADDP — Add real and pop
  - FIADD — Integer add
- Subtraktion
  - FSUB — Subtract real
  - FSUBP — Subtract real and pop
  - FISUB — Integer subtract
  - FISUBR — Integer subtract reversed
  - FSUBR — Subtract real reversed
  - FSUBRP — Subtract real reversed and pop
- Multiplikation
  - FMUL — Multiply real
  - FMULP — Multiply real and pop
  - FIMUL — Integer multiply
- Division
  - FDIV — Divide real
  - FDIVP — Divide real and pop
  - FIDIV — Integer divide
  - FDIVR — Divide real reversed
  - FDIVRP — Divide real reversed and pop
  - FDIVR — Integer divide reversed
- weitere Operationen
  - FSQRT — Square root
  - FSCALE — Scale
  - FPREM — Partial remainder
  - FRNDINT — Round to integer
  - FXTRACT — Extract exponent and significand
  - FABS — Absolute value
  - FCHS — Change sign

Die umfangreiche Zahl der Arithmetikbefehle ermöglicht eine effektive Programmierung durch minimalen Speicherzugriff und den optimalen Einsatz des Gleitkomma-Stack.
Für Subtraktion und Division existieren "reverse" Befehle, die diese Operationen "symmetrisch" machen, wie Addition und Multiplikation.
Operanden müssen in Stack-Elementen oder im Speicher stehen. Ergebnisse können in Stack-Elementen abgelegt werden. Die Operanden, die in den Datenarten long-real, short-real, short-integer oder word-integer vorkommen, werden vom Prozessor automatisch in temporary-real konvertiert.

Tabelle 37: Arithmetikbefehle

Syntax	Ergebnis
FADD	entspricht FADDP ST1,ST
FADD ST,ST1	Addition des Stack-Elementes ST1 zum Stack-Top ST
FADD ST1,ST	Addition des Stack-Top zum Stack-Element ST1
FADD32 M	Addition des Speicheroperanden M (short-real) zum Stack-Top ST
FADD64 M	Addition des Speicheroperanden M (long-rael) zum Stack-Top ST
FADDP ST1,ST	Addition des Stack-Top ST zum Stack-Element ST1 und Stack-Pop
FIADD16 M	Addition des Integer-Speicheroperanden M (short-integer) zum Stack-Top ST
FIADD32 M	Addition des Integer-Speicheroperanden M (word-integer) zum Stack-Top ST
FSUB	entspricht FSUBP ST1,ST
FSUB ST,ST1	Subtraktion des Stack-Elements ST1 vom Stack-Top ST, Differenz --> ST
FSUB ST1,ST	Subtraktion des Stack-Top ST vom Stack-Element ST1, Differenz --> ST1
FSUB32 M	Subtraktion des Speicheroperanden M (short-rael) vom Stack-Top ST, Differenz --> ST
FSUB64 M	Subtraktion des Speicheroperanden M (long-real) vom Stack-Top ST, Differenz --> ST
FSUBP ST1,ST	Subtraktion des Stack-Top vom Stack-Element ST1 und Stack-Pop, Differenz --> ST1
FISUB16 M	Subtraktion des Integer-Speicheroperanden M (short-integer) vom Stack-Top ST, Differenz --> ST
FISUB32 M	Subtraktion des Integer-Speicheroperanden M (word-integer) vom Stack-Top ST, Differenz --> ST
FISUBR16 M	Subtraktion des Stack-Top ST vom Integer-Speicheroperanden M (short-integer), Differenz --> ST
FISUBR32 M	Subtraktion des Stack-Top ST vom Integer-Speicheroperanden M (word-integer), Differenz --> ST
FSUBR	entspricht FSUBRP ST1,ST
FSUBR ST,ST1	Subtraktion des Stack-Top ST vom Stack-Element ST1, Differenz --> ST
FSUBR ST1,ST	Subtraktion des Stack-Elements ST1 vom Stack-Top ST, Differenz --> ST1

Tabelle 37: (Fortsetzung)

Syntax	Ergebnis
FSUBR32 M	Subtraktion des Stack-Top ST vom Speicheroperanden M (short-real), Differenz --> ST
FSUBR64 M	Subtraktion des Stack-Top ST vom Speicheroperanden M (long-real), Differenz --> ST
FSUBRP ST1,ST	Subtraktion des Stack-Elementes ST1 vom Stack-Top ST und Stack-Pop, Differnz --> ST1
FMUL	entspricht FMULP ST1,ST
FMUL ST1,ST	Multiplikation des Stack-Elementes ST1 mit dem Stack-Top ST, Produkt --> ST1
FMUL ST,ST1	Multiplikation des Stack-Top ST mit dem Stack-Element ST1, Produkt --> ST
FMUL32 M	Multiplikation des Stack-Top ST mit dem Speicheroperanden M (short-real), Produkt --> ST
FMUL64 M	Multiplikation des Stack-Top ST mit dem Speicheroperanden M (long-real), Produkt --> ST
FMULP ST1,ST	Multiplikation des Stack-Elementes ST1 mit dem Stack-Top ST und Stack-Pop, Produkt --> ST1
FIMUL16 M	Multiplikation des Integer-Speicheroperanden M (short-integer) mit dem Stack-Top, Produkt --> ST
FIMUL32 M	Multiplikation des Integer-Speicheroperanden M (long-integer) mit dem Stack-Top, Produkt --> ST
FDIV	entspricht FDIVP ST1,ST
FDIV ST,ST1	Division des Stack-Top ST durch das Stack-Element ST1, Quotient --> ST
FDIV ST1,ST	Division des Stack-Elementes ST1 durch den Stack-Top ST, Quotient --> ST1
FDIV32 M	Division des Stack-Top durch den Speicheroperanden M (short-real), Quotient --> ST
FDIV64 M	Division des Stack-Top durch den Speicheroperanden M (long-real), Quotient --> ST
FDIVP ST1,ST	Division des Stack-Elementes ST1 durch den Stack-Top ST und Stack-Pop, Quotient --> ST1
FIDIV16 M	Division des Stack-Top ST durch den Integer-Speicheroperanden M (short-integer), Quotient --> ST
FIDIV32 M	Division des Stack-Top ST durch den Ineger-Speicheroperanden M (long-integer), Quotient --> ST

Tabelle 37: (Fortsetzung)

Syntax	Ergebnis
FDIVR	entspricht FDIVRP ST1,ST
FDIVR ST,ST1	Division des Stack-Elements ST1 durch den Stack-Top ST, Quotient --> ST
FDIVR ST1,ST	Division des Stack-Top ST durch das Stack-Element ST1, Quotient --> ST1
FDIVR32 M	Division des Speicheroperanden M (short-real) durch den Stack-Top ST, Quotient --> ST
FDIVR64 M	Division des Speicheroperanden M (long-real) durch den Stack-Top ST, Quotient --> ST
FDIVRP ST1,ST	Division des Stack-Top ST durch das Stack-Element ST1 und Stack-Pop, Quotient --> ST1
FDIVR16 M	Division des Integer-Speicheroperanden M (short-integer) durch den Stack-Top ST, Quotient --> ST
FDIVR32 M	Division des Integer-Speicheroperanden M (long-integer) durch den Stack-Top ST, Quotient --> ST
FSQRT	Quadratwurzel des Stack-Top ST nach Stack-Top ST
FSCALE	Addition des als Integer-Zahl interpretierten Inhalts von ST1 zu dem Exponenten der Zahl im Stack-Top ST
FPREM	Modulo-Division des Stack-Top ST durch das Stack-Element ST1, Quotient --> ST
FRNDINT	Runden des Stack-Top ST auf eine Integer-Zahl
FXTRACT	Zerlegen der Zahl im Stack-Top ST in zwei Zahlen: Exponent und Basis; Exponent --> ST, Basis --> ST1
FABS	Absolutwert des Stack-Top ST
FCHS	Vorzeichenkomplement des Stack-Top ST

Vergleichsbefehle

FCOM	Compare real
FCOMP	Compare real and pop
FCOMPP	Compare real and pop twice
FICOM	Integer compare
FICOMP	Integer compare and pop
FTST	Test
FXAM	Examine

Die Vergleichsbefehle analysieren das Stack-Top-Element meistens im Verhältnis zu anderen Operanden und zeigen das Ergebnis im Bedingungscode des Statuswortes an. Die Grundoperationen sind Vergleich, Test (Vergleich mit Null) und Prüfen (Tag, Vorzeichen und Normalisierung anzeigen). Spezielle Formen der Vergleichsoperation werden zur Optimierung der Algorithmen bereitgestellt, die den direkten Vergleich mit binären Integer- und Real-Zahlen im Speicher sowie ein Pop des Gleitkomma-Stack gestatten.
Der Befehl FSTSW kann auf einen Vergleich folgend benutzt werden, um den Bedingungscode zur Auswertung in den Speicher zu übertragen.

Tabelle 38: Vergleichsbefehle

Syntax	Ergebnis
FCOM	entspricht FCOM ST1
FCOM ST1	Vergleich des Stack-Top ST mit dem Stack-Element ST1
FCOM32 M	Vergleich des Stack-Top ST mit dem Speicheroperanden M (short-real)
FCOM64 M	Vergleich des Stack-Top ST mit dem Speicheroperanden M (long-real)
FCOMP	entspricht FCOMP ST1
FCOMP ST1	Vergleich des Stack-Top ST mit dem Stack-Element ST1 und Stack-Pop
FCOM32P M	Vergleich des Stack-Top ST mit dem Speicheroperanden M (short-real) und Stack-Pop
FCOM64P M	Vergleich des Stack-Top ST mit dem Speicheroperanden M (long-real) und Stack-Pop
FCOMPP	Vergleich des Stack-Top ST mit dem Stack-Element ST1 und zweifacher Stack-Pop
FICOM16 M	Vergleich des Speicheroperanden M (short-integer) mit dem Stack-Top ST
FICOM32 M	Vergleich des Speicheroperanden M (long-integer) mit dem Stack-Top ST
FICOM16P M	Vergleich des Speicheroperanden M (short-integer) mit dem Stack-Top ST und Stack-Pop
FICOM32P M	Vergleich des Speicheroperanden M (long-integer) mit dem Stack-Top ST und Stack-Pop
FTST	Vergleich des Stack-Top mit Null
FXAM	Ermittelt den Inhalt des Stack-Top, ob positiv oder negativ, ob er normalisiert, unnormalisiert, denormalisiert oder ein NAN-Wert (Not-A-Number) ist, ob er Null oder leer ist.

Befehle für transzendente Funktionen

FPTAN	Partial tangent
FPATAN	Partial arctangent
F2XM1	$2^x - 1$
FYL2X	$y * \log_2 x$
FYL2XP1	$y * \log_2(x + 1)$

Diese Befehle führen den zeitaufwendigen Algorithmus der Berechnungen für trigonometrische, inverse trigonometrische, hyperbolische, inverse hyperbolische, logarithmische und Exponentialfunktionen aus. Die Transzendentalbefehle verarbeiten das erste oder die ersten beiden Elemente vom Stack und übergeben ihr Ergebnis wieder dem Gleitkomma-Stack. Diese Befehle setzen voraus, daß ihre Operanden gültig sind und innerhalb eines zulässigen Bereiches liegen. Ein solcher Operand muß normalisiert sein; denorma-

lisierte, unnormalisierte, unendliche und NaN-Operanden werden als ungültig angesehen.

Tabelle 39: Transzendentalbefehle

Syntax	Ergebnis
FPTAN	Berechnet die Funktion y/x = tan(z), wobei z im Bereich 0 < z < Pi/4 liegen muß. y --> ST, x --> -(ST).
FPATAN	Berechnet die Funktion z = arctan(y/x), wobei x aus dem Stack-Top ST und y aus dem Stack-Element ST1 genommen wird. y und x müssen die Ungleichung 0 < y < x < +oo erfüllen. Der Befehl führt einen Stack-Pop aus, anschließend z --> ST.
F2XM1	Berechnet die Funktion y = 2^x - 1, wobei x aus dem Stack-Top ST genommen wird und im Bereich 0 ≤ x ≤ 0.5 liegen muß, y --> ST.
FYL2X	Berechnet die Funktion z = y * $\log_2 x$, wobei x aus dem Stack-Top ST und y aus dem Stack-Element ST1 genommen wird. Die Operanden müssen im Bereich 0 < x < +oo und -oo < y < +oo liegen. Der Befehl führt einen Stack-Pop aus, anschließend z --> ST.
FYL2XP1	Berechnet die Funktion z = y * $\log_2(x + 1)$, wobei x aus dem Stack-Top ST und y aus dem Stack-Element ST1 genommen wird. x muß innerhalb des Bereiches 0 < |x| < (1 - $\sqrt{2}/2$) und y innerhalb des Bereiches -oo < y < +oo liegen. Der Befehl führt einen Stack-Pop aus, anschließend z --> ST.

Konstantenbefehle

FLDZ	Load + 0.0
FLD1	Load + 1.0
FLDPI	Load Pi
FLDL2T	Load $\log_2 10$
FLDL2E	Load $\log_2 e$
FLDLG2	Load $\log_{10} 2$
FLDLN2	Load $\log_e 2$

Jeder dieser Konstantenbefehle lädt eine allgemein benutzte Konstante in den Gleitkomma-Stack.

Tabelle 40: Konstantenbefehle

Syntax	Ergebnis
FLDZ	Kellern des Wertes +0.0 auf den Stack-Top ST, +0.0 --> ST
FLD1	Kellern des Wertes +1.0 auf den Stack-Top ST, +1.0 --> ST
FLDPI	Kellern des Wertes Pi (3.141592653589) auf den Stack-Top ST, Pi --> ST
FLDL2T	Kellern des Wertes $\log_2 10$ auf den Stack-Top, $\log_2 10$ --> ST
FLDL2E	Kellern des Wertes $\log_2 e$ auf den Stack-Top, $\log_2 e$ --> ST
FLDLG2	Kellern des Wertes $\log_{10} 2$ auf den Stack-Top, $\log_{10} 2$ --> ST
FLDLN2	Kellern des Wertes $\log_e 2$ auf den Stack-Top, $\log_e 2$ --> ST

Prozessorsteuerbefehle

FINIT/FNINIT	Initialize processor
FDISI/FNDISI	Disable interrupts
FENI/FNENI	Enable interrupts
FLDCW	Load control word
FSTCW/FNSTCW	Store control word
FSTSW/FNSTSW	Store status word
FCLEX/FNCLEX	Clear exceptions
FSTENV/FNSTENV	Store environement
FLDENV	Load environement
FSAVE/FNSAVE	Save state
FRSTOR	Restore state
FINCSTP	Increment stack-pointer
FDECSTP	Decrement stack-pointer
FFREE	Free register
FNOP	No operation
FWAIT	CPU wait

Wenn CPU-Interrupts erlaubt sind, z. B. wenn eine Task läuft, sollten die "Wait"-Formen benutzt werden. Die meisten Befehle werden aber eher bei Aktivitäten auf der Systemebene benutzt. Dazu gehören Initialisierung, Behandlung der Ausnahmebedingungen und Taskumschaltung. Die an zweiter Stelle mit N (not) gekennzeichneten Mnemoniks veranlassen den Assembler keinen WAIT-Befehl sondern einen NOP-Befehl voranzustellen.

Tabelle 41: Prozessorsteuerbefehle

Syntax	Ergebnis
FINIT	Prozessor initialisieren (nach WAIT)
FNINIT	Prozessor initialisieren (ohne WAIT)
FDISI	Setzen der Interrupt Maske (nach WAIT)
FNDISI	Setzen der Interrupt Maske (ohne WAIT)
FENI	Löschen der Interrupt-Maske im Steuerwort (nach WAIT)
FNENI	Löschen der Interrupt-Maske im Steuerwort (ohne WAIT)
FLDCW WORD PTR [BP]	Ersetzen des laufenden Prozessorsteuerwortes durch das vom Operanden definierte Wort
FSTCW M	Schreiben des aktuellen Prozessorsteuerwortes nach Speicheroperand M (nach WAIT)
FNSTCW M	Schreiben des aktuellen Prozessorsteuerwortes nach Speicheroperand M (ohne WAIT)
FSTSW M	Schreiben des aktuellen Statuswortes nach Speicheroperand M (nach WAIT)
FNSTSW M	Schreiben des aktuellen Statuswortes nach Speicheroperand M (ohne WAIT)
FCLEX	Löschen aller Exception-Flags, des Interrupt-Anforderungsflags und des Busy-Flags im Statuswort (nach WAIT)
FNCLEX	Löschen aller Exception-Flags, des Interrupt-Anforderungsflags und des Busy-Flags im Statuswort (ohne WAIT)
FSTENV M	Schreiben des Basis-Status (Steuerwort, Statuswort und Tag-Wort) und der Exception-Pointer nach Speicheroperand M (nach WAIT)
FNSTENV M	Schreiben des Basis-Status (Steuerwort, Statuswort und Tag-Wort) und der Exception-Pointer nach Speicheroperand M (ohne WAIT)
FLDENV WORD PTR 6[BP]	Laden des Prozessorzustandes vom Operanden definierten Speicherbereich (muß durch einen vorangehenden FSTENV/FNSTENV-Befehl geschrieben worden sein)

Tabelle 41: (Fortsetzung)

Syntax	Ergebnis
FSAVE M	Schreiben des vollständigen Prozessorstatus (einschließlich Register-Stack) nach Speicheroperand M und Initialisierung des Numerikprozessors (nach WAIT)
FNSAVE M	Schreiben des vollständigen Prozessorstatus (einschließlich Register-Stack) nach Speicheroperand M und Initialisierung des Numerikprozessors (ohne WAIT)
FRSTOR M	Laden des Prozessorstatus aus dem durch den Speicheroperanden M angegebenen 94-Byte-Feld (muß durch einen vorangehenden FSAVE/FNSAVE-Befehl geschrieben worden sein)
FINCSTP	Erhöhung des Stack-Top-Pointers im Statuswort um 1
FDECSTP	Verringerung des Stack-Top-Pointers im Statuswort um 1
FFREE ST1	Tag des Ziel-Stack-Elements auf "leer" setzen. Der Inhalt des Stack-Elements wird nicht beeinflußt.
FNOP	Speichern des Stack-Top in den Stack-Top --> effektiv keine Operation (no operation)
FWAIT	Warten des Prozessors

Sachwortverzeichnis

C 1014-0003-1 M 3030