# 7 DEBUG als Assembler und Werkzeug in DOS.

Zum Lieferumfang des DOS-Betriebssystems gehört auch das Programm DEBUG. Dieses Programm bietet zahlreiche Hilfsfunktionen, die unter anderem auch die Erstellung von kleineren COM-Programmen aus Assembleranweisungen erlauben. Das Produkt besitzt zwar einige Restriktionen und Unbequemlichkeiten was die Kommandoeingabe betrifft. Wer aber über keinen Assembler verfügt, kann mit DEBUG alle Übungen aus Kapitel 2 durchführen und erhält eine Reihe interessanter Utilities.

## 7.1 Die Funktion des Debuggers

Die Hauptfunktion des Debuggers besteht in der kontrollierten Ausführung des zu testenden Programmes. Hierzu gehört, daß das Programm an definierten Stellen angehalten, geändert und wieder gestartet werden kann. Für die Assemblerprogrammierung ist es weiterhin nützlich, wenn sich die Speicherinhalte anzeigen und modifizieren lassen. Funktionen zur Übersetzung von Assembleranweisungen in Maschinensprache und die Möglichkeit zur Rückübersetzung sind ebenfalls recht interessant. Alle diese Funktionen werden von dem Programm DEBUG geboten.

### 7.1.1 Der Programmstart

DEBUG besitzt folgende Aufrufsyntax:

DEBUG [File] [Parameter]

Die in Klammern stehenden Felder [File] und [Parameter] sind optional und können bei der Eingabe entfallen. In [File] kann bereits beim Aufruf der Name des zu testenden Programmes angegeben werden. DEBUG lädt dann direkt den Programmcode in den Speicher, ohne ihn jedoch auszuführen. Falls das Programm nach dem Aufruf bestimmte Parameter aus der Kommandozeile erwartet, lassen sich diese im Feld [Parameter] eintragen. Nähere Ausführungen zu den beiden Aspekten finden sich bei der Beschreibung des NAME-Befehls. Nachfolgend sind einige gültige Aufrufe für den Debugger aufgeführt:

```
DEBUG
DEBUG ASK.COM
DEBUG ASK.COM Hallo
```

Nach dem Aufruf des Debuggers aus der DOS-Ebene erscheint der PROMPT - auf dem Bildschirm.

```
C>DEBUG
-
```

Damit zeigt DEBUG die Betriebsbereitschaft an und erwartet ein Kommando. Eine Zusammenstellung der möglichen Kommandos findet sich in der folgenden Tabelle.

| Befehl | Aufrufsyntax |
|---|---|
| Assemble | A [adresse] |
| Compare | C adresse range adresse |
| Dump | D [adresse][länge] |
| Enter | E [adresse] [bitmuster] |
| Fill | F [adresse] [länge] [Bitmuster] |
| Go | G [=adresse] [breakpoint 1 .. 10] |
| Hexarithm. | H wert1 wert2 |
| Input | I portadresse |
| Load | L [adresse] [drive sektor sektor] |
| Move | M [adresse] [länge] [adresse] |
| Name | N filename [filename ..] |
| Befehl | Aufrufsyntax |
| Output | O adresse byte |
| Proceed | P [adresse] [wert] |
| Quit | Q |
| Register | R [registername] |
| Search | S [adresse] [länge] [wert] |
| Trace | T [=adresse] [wert] |
| Unassemble | U [adresse] [länge] |
| Write | W [adresse] [drive sektor sektor] |
| XA(llocate) | XA [count] |
| XD(ealloc.) | XD [handle] |
| XM(ap EMS) | XM [lpage] [ppage] [handle] |
| XS | XS |

*Tabelle 7.1: DEBUG-Befehle*

Nun noch einige Hinweise auf die Notation: Sofern im Text nichts anderes spezifiziert ist, sind alle Eingaben für DEBUG mit der RETURN-Taste abzuschließen.

DEBUG läßt sich im Kommandomode (-) durch die Eingabe:

```
-Q
```

beenden. Anschließend erscheint der DOS-PROMPT wieder. Im Speicher geladene Programme werden nicht automatisch auf Diskette gesichert. Falls in den nach-

folgenden Beispielen der Bindestrich vor dem Kommando erscheint, handelt es sich um die Ausgabe des Debuggers. Der PROMPT darf natürlich beim Abtippen der Beispiele nicht mit eingegeben werden! Bei den Kommandos dürfen wahlweise Groß- und Kleinbuchstaben verwendet werden, da DEBUG alle Zeichen in Großbuchstaben konvertiert. Ein laufendes Kommando läßt sich durch gleichzeitiges drücken der Tasten <Ctrl>+<C> abbrechen. Sollte dies nicht möglich sein, hilft nur noch ein Systemreset mittels der Tasten <Alt>+<Ctrl>+ <Del>, oder mittels des Netzschalters. Die Bildschirmausgabe läßt sich mit den Tasten <Ctrl>+<S> unterbrechen. Eine Freigabe kann dann durch jede andere beliebige Taste erfolgen. Mit den Tasten <Shift>+<PrtSc> läßt sich der aktuelle Bildschirminhalt auf dem Drucker protokollieren. Mit <Ctrl>+<P> kann die Protokollierung aller Bildschirmausgaben auf dem Drucker ein- und wieder ausgeschaltet werden. Weiterhin lassen sich die DOS I/O-Umleitungen mit DEBUG nutzen. Weitere Hinweise finden sich bei der Beschreibung des ASSEMBLE-Kommandos.

Doch nun möchte ich auf die ersten DEBUG-Befehle eingehen.

### 7.1.2 Der DUMP-Befehl

Der DUMP-Befehl stellt einen Speicherbereich als Folge von Hexzahlen auf dem Bildschirm dar. Es gilt dabei folgende Aufrufsyntax:

D [Seg:[Offs]] [Range]

Die in Klammern stehenden Parameter sind optional und müssen nicht angegeben werden. Das Feld *Seg* bezeichnet den Wert des Segments, während *Offs* dem Offsetanteil der Adresse entspricht. *Range* ist ein weiterer Parameter, der die Zahl der zu bearbeitenden Bytes angibt. Die DUMP-Ausgabe besteht im Standardformat aus 8 Zeilen:

```
-D
0F66:0100  6A 0E 8B 7E EA 8B 56 F2-89 15 8B 7E EA 8B 56 F0   j..~j.Vr...~j.Vp
0F66:0110  89 55 02 8B 7E EA 8B 56-EC 89 55 04 EB 57 8B 7E   .U..~j.Vl.U.kW.~
0F66:0120  EA 8B 55 02 83 C2 01 89-56 E8 C4 5E F4 26 8B 17   j.U..B..VhD^t&..
0F66:0130  32 F6 89 56 E6 8B 56 E8-3B 56 E6 7F 27 89 56 F0   2v.Vf.Vh;Vf.'.Vp
0F66:0140  C4 5E F4 03 5E F0 26 8B-17 8B 4E F0 83 C1 FF C4   D^t.^p&...Np.A.D
0F66:0150  5E F4 03 D9 26 88 17 8B-56 F0 42 89 56 F0 4A 3B   ^t.Y&...VpB.VpJ;
0F66:0160  56 E6 75 DC C4 5E F4 26-8B 17 32 F6 83 C2 FF C4   Vfu\D^t&..2v.B.D
0F66:0170  5E F4 26 88 17 C6 06 FE-E1 01 8B 7E EA FF 35 FF   ^t&..F.~a..~j.5.
-
```

*Bild 7.1: Ausgabe des DUMP-Befehls*

Die ersten 9 Zeichen einer Zeile geben die Speicheradresse des folgenden Bytes an. Dieses Byte, sowie weitere 15 Werte sind durch zwei Leerzeichen von der Adreßanzeige getrennt. Der Übergang zwischen dem achten und dem neunten Byte wird durch einen Bindestrich markiert. Den Abschluß einer Zeile bildet die ASCII-Darstellung der Werte. Jedem Byte wird ein entsprechender ASCII-Code zugeordnet. Ist ein Zeichen nicht darstellbar, z.B. das Zeichen 0A, wird es durch einen Punkt ersetzt. Durch eine weitere Eingabe des Zeichens:

-D

werden die nächsten acht Zeilen ausgegeben, d.h. DUMP merkt sich die zuletzt ausgegebene Adresse. Bei der Eingabe können an Stelle des Segmentwertes auch die Register CS, DS, ES oder SS stehen. Wird die Segmentadresse weggelassen, z.B.:

-D 0100

übernimmt DUMP den Inhalt des Datensegmentregisters und interpretiert den Wert 0100 als Offset. Soll ein anderes Segmentregister benutzt werden, ist dies beim Aufruf anzugeben:

-D CS:0100
-D SS:0100
-D ES:0100

Die obigen Aufrufe benutzen das Codesegment (CS), das Stacksegment (SS) und das Extrasegment (ES) als Segmentadresse.

Fehlt die Adreßangaben komplett, benutzt DEBUG das Datensegment und den zuletzt eingestellten Offset zur Adressberechnung. Nach dem Start von DEBUG wird der Offset auf den Wert 100 gesetzt, da dies der Adresse des ersten ausführbaren Programmschritts entspricht.

Die letzte Bemerkung gilt der Option [Range], mit der sich die Zahl der auszugebenden Bytes definieren läßt. Dies kann einmal mittels einer Start- und Endadresse erfolgen. Mit:

-D DS:0 0F

werden z.B. 16 Byte des Datensegments ab dem Offset 0000 bis zur Adresse 000F ausgegeben. Alternativ läßt sich der Befehl auch mit einer Längenangabe formulieren:

-D DS:0 L 10

Dabei gibt der Buchstabe L an, daß der folgende Parameter nicht als Adresse, sondern als Länge zu interpretieren ist.

### 7.1.3 Die ENTER-Funktion

Um Speicherbereiche zu ändern, bietet DEBUG die ENTER-Funktion. Dieser Befehl besitzt folgende Aufrufsyntax:

E [Seg:[Offs]] [Bitmuster]

Alle Adreßeingaben erfolgen in der bereits beim DUMP-Befehl beschriebenen Notation.

Geben Sie bitte folgende Kommandos unter DEBUG ein:

-E CS:0100

Auf dem Bildschirm erscheint dann zum Beispiel folgender Text:

-E 4770:0100
4770:0100 41.

ENTER zeigt nach dem Aufruf das erste Byte der angegebene Adresse (A = 41H). Der Punkt signalisiert, daß ENTER auf eine Eingabe wartet. Die Segmentadresse und der erste angezeigte Wert (hier 41H) wird von System zu System, je nach der Speicherbelegung, schwanken. Geben Sie nun bitte folgende Werte ein:

40 blank 8F blank blank F0 <RETURN>

<RETURN> steht dabei für die RETURN-Taste. Mit *blank* wird hier die Leertaste bezeichnet. In der Eingabezeile sollte in etwa folgendes Bild erscheinen:

-E CS:0100
4770:0100 41.40 07.8F 3E. 07.F0
-

Die Eingaben lassen sich mit der Anweisung:

-D CS:100

kontrollieren. In den angewählten Bytes sind neue Werte eingetragen. ENTER erlaubt im Abfragemodus verschiedene Eingaben. Nach der Anzeige des Speicherinhaltes gibt ENTER einen Punkt als PROMPT aus und wartet auf eine Eingabe. Hierbei sind folgende Alternativen möglich:

- Abbruch des Befehls mit der RETURN-Taste. Dann erscheint der DEBUG-PROMPT wieder. Der Wert der zuletzt angezeigten Zelle bleibt erhalten.

- Die Leertaste schaltet zur folgenden Adresse weiter. Ohne weitere Eingaben bleibt der Wert der zuletzt angezeigten Zelle unverändert.

- Eingegebene Hexziffern im Bereich zwischen [0 .. FF] werden in den Speicher übertragen und überschreiben die zuletzt angezeigte Zelle. Die Eingabe ist durch ein Leerzeichen abzuschließen.

- Durch Eingabe des Minuszeichens - wird die vorhergehende Adresse und deren Inhalt in einer neuen Zeile angezeigt.

Sobald mehr als 8 Byte angezeigt wurden, oder nach der Eingabe eines Minuszeichens, beginnt ENTER mit einer neuen Zeile. Mit dem Minuszeichen können also Speicherzellen in absteigender Reihenfolge modifiziert werden.

```
-E
B000:00A4 64.-
B000:00A3 F0.-
B000:00A2 3E.
-
```

Beachten Sie aber, daß immer nur der Offset der Adresse dekrementiert wird, d.h. ENTER arbeitet nur innerhalb eines 64 KByte Bereichs. Sollen gleichzeitig mehrere Bytes modifiziert werden, ohne daß der alte Wert angezeigt wird, kann dies z.B. mit folgender Eingabe erfolgen:

```
-E DS:0200 "Dies ist ein Text"
-
```

Die Bytes "Dies ist.." werden ab der Adresse DS:0200 in den Speicher geschrieben. Dieses Eingabeformat ist vor allem dann nützlich, wenn Texte als ASCII-Zeichen in der Form ".." eingegeben werden müssen. Dann kann die umständliche Umrechnung der Zeichen in Hexwerte entfallen. Alle ASCII-Zeichen sind durch Hochkommas oder Anführungsstriche einzuschließen.

### 7.1.4 Der FILL-Befehl

FILL füllt größere Speicherbereiche mit einem bestimmten Bytemuster. Das Kommando besitzt folgende Aufrufsyntax:

```
F Seg:Offs Lxx Bytemuster
```

Die Adresse spezifiziert den Bereich, ab dem das angegebene Bytemuster abgespeichert wird. Die Eingabe muß gemäß den bereits besprochenen Konventionen erfolgen. Die Länge des zu füllenden Bereiches ist auf jeden Fall im Parameter *Lxx* anzugeben. Die Zeichen xx stehen dabei für eine maximal 16 Bit breite Zahl (0-FFFFH). Weiterhin sind im Feld *Bytemuster* die Werte für das zu speichernde Muster anzugeben. Ist der zu füllende Bereich größer als das angegebene Füllmuster, dann wiederholt FILL die Operation so oft, bis der spezifizierte Bereich gefüllt ist. Sollen z.B. 16 Byte ab Adresse CS:100 mit dem Wert FF belegt werden, lautet das Kommando:

```
-F CS:100 L 0F FF
-
```

Das Bytemuster (FF) enthält weniger Bytes als in Lxx spezifiziert, deshalb wird die Operation so lange wiederholt, bis alle Bytes überschrieben wurden.

### 7.1.5 Die MOVE-Funktion

MOVE erlaubt ganze Bereiche innerhalb des Speichers zu verschieben. Das Aufrufformat für den Befehl lautet:

M Startadr Endadr Zieladr

Die Adressen sind wie gewohnt in der Segment:Offset-Notation anzugeben. Mit Startadresse wird der Beginn des zu verschiebenden Bereiches spezifiziert. Im Parameter *Endadr* ist die Adresse des letzten zu verschiebenden Bytes definiert. Steht hier der Text:

M XXXX L YYYY ZZZZ

wird der Wert YYYY nicht als Endadresse sondern als Längenangabe interpretiert. Der Parameter *Zieladresse* bestimmt, wohin die Bytes verschoben werden. Überlappende Bereiche werden so kopiert, daß keine Informationen verloren gehen. Nur bei der Start- und Zieladresse darf ein Segment angegeben werden. Dies bedeutet, daß sich maximal der Inhalt eines 64 KByte großen Segments in einem Stück verschieben läßt. Ohne Angabe der Segmentadresse wird das DS-Register benutzt. Die Segmentangabe der Startadresse gilt auch für die Endadresse. Die Differenz zwischen Startadresse und Endadresse gibt die Zahl der zu verschiebenden Bytes an. Geben Sie bitte folgende Zeichen ein:

```
-M CS:100 20 CS:200
-
```

Dann wird ein Bereich von 32 Byte im Codesegment zwischen 100 und 200 verschoben. Die Eingabe:

```
-M CS:0 100 DS:0
-
```

kopiert z.B. 256 Byte des Codebereichs in das Datensegment. Die Quelldaten bleiben dabei im jeweiligen Adressbereich erhalten.

### 7.1.6 Die INPUT-/OUTPUT-Befehle

Diese Befehle erlauben es, beliebige Ein- / Ausgabeadressen der 8086/8088 Prozessoren anzusprechen. Dabei gilt folgende Aufrufsyntax:

O Port Byte
I Port

Mit dem Kommando:

O Port Byte

kann ein Bytewert [0..FF] in den Ausgabeport geschrieben werden. Der Adressbereich eines Ports liegt im Bereich zwischen 0 - 64 KByte. Der Befehl:

```
-O 2F8 FF
-
```

schreibt den Wert FF in den Port mit der Adresse 2F8. Mit dem Befehl:

I Port

läßt sich der Inhalt der Portadresse [Port] einlesen. Die Eingabe:

```
-I 2F8
6B
-
```

liest das Port mit der Adresse 2F8H aus und gibt dann den Wert (z.B. 6B) zurück.

### 7.1.7 Die HEX-Funktion

Das Kommando gibt Hilfestellung bei der Berechnung der Summe und der Differenz zweier Hexdezimalzahlen. Das Kommando besitzt das Format:

H Zahl1 Zahl2
Summe Differenz

Mit *Zahl1* und *Zahl2* sind zwei maximal 16 Bit große Hexadezimalzahlen einzugeben, deren Summe und Differenz in der folgenden Zeile angezeigt wird. Versuchen Sie folgende Eingabe:

```
-H 0F 02
11 0D
-
```

Die Summe beider Zahlen ist 11H, während ihre Differenz 0DH beträgt.

### 7.1.8 Die COMPARE-Funktion

Mit Hilfe dieser Funktion lassen sich zwei Speicherbereiche auf übereinstimmende Speicherinhalte vergleichen. Die Aufrufsyntax lautet:

C [Seg]:Offs Len [Seg]:Offs

Die Adressen spezifizieren den Beginn der zwei zu vergleichenden Bereiche. Ohne Segmentangaben wird das DS-Register zur Adressberechnung benutzt. *Len* gibt an, wieviele Bytes zu vergleichen sind. Werden ungleiche Bytes gefunden, erfolgt eine Anzeige in der Form:

Adr1 Byte1 Byte2 Adr2

Die Anzeige gibt also die Lage und die Werte der unterschiedlichen Speicherzellen an. Ein Eingabebeispiel könnte folgendermaßen aussehen:

```
-C CS:0 L3 DS:0
0000:0000 FF FE 0100:0000
0000:0001 FE CD 0100:0001
0000:0002 CD 55 0100:0002
-
```

Hier wird ein Ausschnitt des Codebereiches mit dem Datenbereich verglichen. In obigem Fall sind alle Bytes unterschiedlich. L3 gibt an, daß 3 Byte zu vergleichen sind. Wird keine Abweichung gefunden, meldet sich DEBUG nach dem Vergleich nur mit dem PROMPT "-" zurück.

### 7.1.9 Die SEARCH-Funktion

Um bestimmte Werte im Speicherbereich aufzufinden kann der SEARCH-Befehl benutzt werden. Ihr Eingabeformat lautet:

S [Seg]:Offs Len Bytemuster

Ohne Angabe der Segmentadresse erfolgt die Suche im Datensegment (DS). *Len* gibt die Zahl der zu durchsuchenden Bytes an. Wird in diesem Bereich das Bytemuster gefunden, zeigt SEARCH anschließend die Anfangsadresse an. Falls das Muster mehrfach im Suchbereich auftritt, werden auch die entsprechenden Adressen angezeigt. Erscheint nur der PROMPT "-" nach Aufruf des Befehles, dann wurde das Muster nicht im angegeben Bereich gefunden. Eine Eingabe könnte folgendermaßen aussehen:

```
-S CS:100 200 55
1000:0101
1000:01FF
-
```

In diesem Fall taucht das Byte 55 zweimal im Codesegment im Bereich 100 .. 300 auf. Es besteht auch die Möglichkeit, nach ganzen Zeichenketten zu suchen:

```
-S DS:100 L100 "Test"
```

Hier wird ein Bereich von 256 Byte nach dem Muster "Test" abgesucht.

### 7.1.10 Der REGISTER-Befehl

Eine weitere Funktion ermöglicht es, den Inhalt der Prozessorregister anzuzeigen und zu verändern. Dies erfolgt mittels einer Eingabe im Format:

```
 R [Register]
```

Wird kein Register angegeben, erscheint die folgende Anzeige.

```
-R
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1007  ES=1007  SS=1007  CS=1007  IP=0100   NV UP EI PL NZ NA PO NC
1007:0100 0000   ADD   [BX+SI],AL                      DS:0000=CD
-
```

Die erste Zeile enthält die Anzeige der Arbeitsregister, während in der zweiten Zeile die Segmentregister, der Instruktionszähler (IP), sowie der Zustand der Flags dargestellt werden. In der dritten Zeile wird der aktuell im Codesegment per Instruktionszähler angesprochene Befehl decodiert. Findet ein Zugriff auf Daten statt, werden Adresse und Inhalt des entsprechenden Datums angezeigt. Soll nur der Inhalt eines Registers angezeigt oder verändert werden, erfolgt dies durch die Eingabe des Registernamens. Nachfolgendes Beispiel setzt das BX-Register.

```
-R BX
BX 0000
:3456
-
```

Nach Selektion des BX-Registers in der Eingabezeile wird dessen Inhalt in der zweiten Zeile dargestellt. In der dritten Zeile erscheint die Abfrage des neuen Wertes. Dies ist am PROMPT in Form des Doppelpunktes erkennbar (:). Durch die RETURN-Taste wird die Eingabe abgebrochen ohne den Registerwert zu verändern. In obigem Beispiel wird der Registerinhalt auf den Wert 3456 gesetzt und durch die RETURN-Taste abgeschlossen.

Der Zustand der Flags kann durch:

```
-R F
OV DN EI NG ZR AC PE CY-
```

abgerufen werden. Der Bindestrich am Ende der zweiten Zeile signalisiert, daß DEBUG auf eine Eingabe wartet. Mit RETURN wird die Anzeige ohne Änderungen verlassen. Sollen bestimmte Flags modifiziert werden, ist der jeweils komplementäre Wert in der aktuellen Anzeigezeile einzugeben:

```
-R F
OV DN EI NG ZR AC PE CY-NCNZ
-
```

Mit NC wird das Carry-Flag gelöscht, während NZ das Zero-Flag beeinflußt. Die folgende Tabelle beinhaltet die Bezeichnung der Flags und deren Zustände.

| Flag | Set | Clear |
|---|---|---|
| Overflow | OV | NV |
| Direction | DN | UP |
| Interupt | EI | DI |
| Sign | NG | PL |
| Zero | ZR | NZ |
| Auxillary Carry | AC | NA |
| Parity | PE | PO |
| Carry | CY | NC |

*Tabelle 7.2: Bezeichnung der Flags*

Mit Hilfe der REGISTER-Funktion lassen sich insbesondere die Segmentregister für andere DEBUG-Befehle voreinstellen. Weiterhin wird der Befehl im Zusammenhang mit der weiter unten vorgestellten WRITE-Funktion benötigt.

Beim Start von DEBUG werden die Segmentregister auf den untersten freien Speicherbereich gesetzt. Das Register IP enthält den Wert 100. Wurde ein Programm durch DEBUG geladen, steht die Zahl der Bytes in den Registern BX:CX.

### 7.1.11 Die File I/O-Befehle

DEBUG enthält einige Funktionen zum Einlesen und Speichern von Dateien und Platten-/Diskettensegmenten. Dies Funktionen werden nachfolgend beschrieben.

### 7.1.12 Der NAME-Befehl

Um ein Programm zu Speichern oder zu Laden muß zuerst ein Dateiname angegeben werden. Hierfür existiert die NAME-Funktion mit dem Format:

N [file1] [file2] [Param]

DEBUG speichert Dateinamen zum Laden der Dateien ab. Im Feld definierte Parameter werden im Programm-Segment-Prefix (PSP) des Programmes ab CS:81 gespeichert. Die LOAD- und WRITE-Kommandos greifen auf diese Informationen zu. Der erste Name wird als Filename der zu ladenden oder zu beschreibenden Datei interpretiert. Mit der Anweisung:

N C:COMMAND.COM

wird die nächste WRITE- oder READ-Anweisung auf die Datei COMMAND.COM zugreifen. Wichtig ist, daß beim Aufruf die Extension des Dateinamens mit angegeben wird.

### 7.1.13 Der WRITE-Befehl

Nach der Definition des Dateinamens per NAME-Befehl ist DEBUG bereit ein WRITE-Kommando mit dem Format:

W [adress [drive sector sector]]

auszuführen. Ab der angegebenen Adresse werden nun die Bytes in einer Datei abgelegt. Es gelten die bereits besprochenen Konventionen bezüglich der Segmentangabe. Standardmäßig wird das CS-Register (CS:100) benutzt. Durch Angabe des Laufwerkes A=0, B=1, etc. und der Disksektoren können auch absolute Sektoren einer Disk beschrieben werden. Da dies aber eine sehr unsichere Sache ist, bei Angabe von falschen Werten wird der Inhalt der Diskette zerstört, sollte diese Möglichkeit nur mit großer Vorsicht benutzt werden. WRITE setzt voraus, daß vorher mit NAME ein korrekter Dateiname spezifiziert wurde. Nun stellt sich noch die Frage, wieviele Byte abzuspeichern sind. Diese Information muß mit dem R-Befehl in die BX- und CX-Register (BX:CX) geschrieben werden. Wurde keine Startadresse angegeben, übernimmt WRITE automatisch alle Bytes ab CS:100. Bei der Eingabe des Dateinamens dürfen die Attribute .HEX und .EXE nicht verwendet werden, da DEBUG diese Dateiformate nicht unterstützt. Ein COM-Programm wird mit folgenden Eingaben:

```
-N A:ASK.COM
-R CX
CX 0000
:40
-R BX
```

```
BX 0000
:
-W 100
-
```

in der Datei ASK.COM abgespeichert. Obige Sequenz speichert dabei 40H Byte. Wird kein NAME-Kommando ausgeführt, benutzt DEBUG die zufällig im File-Control-Block stehenden Parameter. Dies kann z.B. der beim Aufruf des Debuggers angegebene Filename des zu ladenden Programmes sein. Um Fehler zu vermeiden, sollte vor jedem WRITE-Kommando der Dateiname mit NAME definiert werden.

### 7.1.14 Der LOAD-Befehl

Das Laden eines Programmes erfolgt mit der LOAD-Funktion:

L [adress] [drive sector sector]

Das Format entspricht der WRITE-Anweisung und erlaubt auch absolute Disksektoren zu lesen. Bei Dateizugriffen wird die NAME-Funktion zur Eingabe des Dateinamens benutzt. Mit der Eingabe:

```
-N A:ASK.COM
-L
-
```

wird der in der Datei ASK.COM abgespeicherte Code geladen. Da keine Adresse angegeben wurde, legt LOAD den Code ab CS:100 ab. Die Zahl der Bytes wird an Hand der Dateiinformationen automatisch bestimmt. Diese Information findet sich nach dem Ladevorgang in den BX- und CX-Registern (BX:CX). Wurde beim Dateinamen die Extension .EXE oder .HEX spezifiziert, kann die Adresse nicht mehr angegeben werden, da die Adreßeinstellung aus der jeweiligen Datei entnommen wird. Ausführbare Programme werden durch DOS standardmäßig auf die Adresse CS:100 gesetzt. Lediglich bei HEX-Files addiert DEBUG die angegebene Adresse als Offset zu der Ladeadresse in der Datei.

Neben dem expliziten LOAD-Befehl gibt es natürlich noch die Möglichkeit, die Datei bereits beim Aufruf des Debuggers zu laden (z.B. DEBUG ASK.COM).

### 7.1.15 Programmentwicklung mit DEBUG

Alle bisher beschriebenen Befehle dienen mehr oder weniger zur Manipulation der Speicherinhalte. Mit DEBUG lassen sich aber auch Programme erstellen und testen. Hierfür stehen die Funktionen:

- ASSEMBLE
- LOAD
- NAME
- GO
- PROCEED
- TRACE
- UNASSEMBLE
- WRITE

zur Verfügung. NAME, LOAD und WRITE wurden bereits vorgestellt. Nachfolgend möchte ich auf die Technik zur Assemblierung kleinerer Programme eingehen.

### 7.1.16 Der ASSEMBLE-Befehl

Der Debugger besitzt einen zeilenorientierten Assembler, mit dem sich eingegebene Anweisungen direkt in Maschinencode umsetzen und im Speicher ablegen lassen. Die Funktion besitzt folgende Aufrufsyntax:

A [Seg:[Offs]]

Die Adresse gibt an wo der erste Befehl abgelegt werden soll. Fehlt die Adreßangaben, wird CS:0100 eingestellt. Alle Zahlen sind als Hexziffern einzugeben. Bei Kommandos zur Stringmanipulation ist der Datentyp explizit anzugeben (Byte oder Word). Ein Rücksprung (Return) über Segmentgrenzen ist als RETF (Return Far) einzugeben. Relative Sprünge werden automatisch mit dem korrekten Displacement assembliert, nachdem die absolute Zieladresse eingegeben wurde. Es ist jedoch auch möglich, einen entsprechenden Präfix (NEAR, FAR) einzugeben.

```
0100:0400 JMP 402          ; short jump
0100:0402 JMP NEAR 505     ; near jump
0100:0405 JMP FAR 50A      ; far jump
```

Der Präfix NEAR darf dabei mit NE abgekürzt werden. Bei Operationen auf Daten ist deren Typ (WORD, BYTE) explizit anzugeben.

```
MOVSB                ; Byte String Move
MOVSW                ; Word String Move
DEC WORD PTR [SI]
DEC BYTE PTR [SI]
```

Indirekte Adressierungen werden durch eckige Klammern markiert.

```
MOV AX,3F      ; lade AX direkt mit 3F
MOV AX,[0FF]   ; lade AX indirekt mit dem
               ; Wert der Adresse 0FF
MOV AX,[SI]    ; lade indirekt mit dem
```

```
                                ; Wert der Adresse in SI
ADD BX,34[BP+2][SI+2]           ; addiere indirekt
POP [BP+DI]                     ; Pop Memory Wert
PUSH [SI]                       ;  Push Memory Wert
```

Die Pseudoinstruktionen DB (Byte) und DW (Wort) zur Eingabe von Konstanten werden unterstützt.

```
DB 1, 2, 3, "Hallo"
DW 3FF, 2000, 'Testtext'
DB "Hallo""Test"
```

Damit ist es möglich kleine Assemblerprogramme im Speicher zu erstellen. Mit ES:, DS: und CS: läßt sich die standardmäßige Segmenteinstellung für den nächsten Befehl überschreiben. Mit:

```
ES: MOV AX,[SI]
```

bezieht sich der Zeiger SI nicht auf das Daten- sondern auf das Extrasegment. Die Eingabe der Assembleranweisungen wird mittels der RETURN-Taste abgebrochen. Alle korrekt angegebenen Anweisungen werden Zeile für Zeile in den Speicher assembliert. Bei Fehleingaben zeigt DEBUG dies durch eine Fehlermeldung an:

```
-A CS:100
1400:0100 MOV AD,100
                  ^Error
```

Dann wird die Adresse angezeigt und auf eine neue Eingabe gewartet. Auf diese Weise lassen sich beliebig lange Programme erstellen. Die einzige Einschränkung bei der Assemblierung betrifft die Verarbeitung von Labels. Labels werden von DEBUG nicht unterstützt, so daß Sprünge oder CALL-Befehle mit den direkten Adressen einzugeben sind. Falls das Sprungziel zur Eingabezeit noch nicht bekannt ist, kann eine Pseudokonstante (z.B. 0000) eingetragen werden.

```
JMP NEAR 0000
```

Der Assembler reserviert dann die korrekte Anzahl Opcodes im Speicher. Nachdem die Sprungadresse feststeht, lassen sich die entsprechenden Adressen entweder durch ENTER oder durch erneute Eingabe eines ASSEMBLE-Befehls an der jeweiligen Stelle nachtragen.

```
-A CS:100
1400:100 JMP NEAR 0122
```

### 7.1.17 Assemblierung aus einer Datei

Die direkte Eingabe von Assemblerbefehlen ist nur für kurze Programmtests handhabbar. Komfortabler ist es die Quelltexte aus einer Datei in einem Durchlauf zu übersetzen. Dies läßt sich in MS-DOS über einen Trick erreichen. Zuerst wird die Quelldatei mittels eines Texteditors erstellt. Diese darf beliebige (aber gültige) DEBUG-Anweisungen enthalten. Dann wird der Debugger unter Verwendung der DOS-I/O-Umleitung gestartet. Nachfolgend ist ein solcher Aufruf dargestellt:

DEBUG < ASK.ASM > ASK.LST

ASK.ASM steht für die Quelldatei, während in ASK.LST alle Meldungen während der Assemblierung gespeichert werden. Damit liest DEBUG die Eingaben nicht mehr von der Tastatur, sondern aus der Datei ASK.ASM. Die Textausgaben lassen sich weiter in die zweite Ausgabedatei (ASK.LST) umleiten. Nachfolgendes Beispiel erzeugt eine lauffähige COM-Datei zur Abfrage der DOS-Versionsnummer.

```
A CS:100
;-------------------------------------------------
; Beispiel zur Assemblierung mit dem DOS-Debugger
; aus einer Textdatei.
;
; Aufruf:  DEBUG < VERSION.ASM > VERSION.LST
;
; In der Datei VERSION.LST werden alle Meldungen
; (auch Fehlermeldungen) des Debuggers abgelegt.
; Die lauffähige COM-Datei findet sich in
; VERSION.COM
;-------------------------------------------------
; Assembliere den Programmcode ab CS:100
;
; ORG 100
;
MOV DX,0200             ; lade Adr. String 1
MOV AH,09               ; Ausgabe des Textes
INT 21                  ; per INT 21
MOV AH,30               ; Abfrage der DOS-
INT 21                  ; Version
MOV DL,30               ; Convert Main Nr.
ADD DL,AL               ;     "
MOV AH,02               ; Ausgabe Character
INT 21                  ;     "
MOV DL,2E               ; write "."
INT 21                  ;     "
MOV DL,30               ; Convert Second Nr.
ADD DL,AH               ;     "
INT 21                  ; write char.
MOV DX,0210             ; lade Adr. String 2
MOV AH,09               ; Ausgabe CR,LF
INT 21                  ; per INT 21
MOV AX,4C00             ; Terminate Process
INT 21                  ; normal
NOP                     ; Ende des Programmcodes
```

```
; hier muß eine Leerzeile folgen !!!

A CS:200
;
; ORG 200
;
; Assembliere die Text-Konstanten ab CS:200
; statischer Text
DB "MS-DOS Version $"
; CR,LF
DB 0A,0D,"$"
;-------------------------------------------
; END     Leerzeile beendet Assemble Mode
;-------------------------------------------
; speichere den Code in der Datei VERSION.COM
;-------------------------------------------
 hier muß eine Leerzeile folgen  !!

N VERSION.COM
R BX
0
R CX
200
W CS:100
Q
```

*Listing 7.1: VERSION.ASM*

Interessant ist in diesem Zusammenhang noch eine undokumentierte Eigenschaft von DEBUG. Normalerweise lassen sich in DEBUG keine Kommentare eingeben. Ist jedoch der ASSEMBLE-Befehl aktiv, akzeptiert DEBUG Kommentarzeilen. Alle Texte mit einem vorangestellten Semikolon werden als Kommentar betrachtet und überlesen. Lediglich bei DB- und DW-Anweisungen erfolgt eine Fehlermeldung, so daß DEBUG hier keine Kommentare akzeptiert. In obigem Programm bewirkt die erste "A CS:100" Anweisung, daß DEBUG den Kommentarkopf überliest. Die Daten werden in den Bereich ab CS:200 assembliert. Vor der Anweisung "A CS:200" muß eine Leerzeile stehen, um mit dem RETURN-Zeichen der Leerzeile das noch aktive A-Kommando abzubrechen. Wichtig ist auch, daß in einem Block mit Assembleranweisungen keine Leerzeilen auftreten, da diese auch den ASSEMBLE-Mode beenden.

Mit der W-Anweisung lassen sich 200H Byte ab CS:100 speichern. Die Sequenz:

```
N VERSION.COM
R BX
0
R CX
200
W
Q
```

ist deshalb in jedem Quellprogramm, getrennt durch eine Leerzeile, an den Assemblercode anzuhängen. Ohne den Q-Befehl hängt sich DEBUG auf, da das Programm ja alle Anweisungen aus der Quelldatei erwartet. Hier muß folglich als letztes auch die Abbruchanweisung stehen. Die restlichen Befehle legen eine COM-Datei an. Ist die Länge des Programmes noch nicht bekannt, kann in den Registern CX und BX ein Pseudowert abgelegt werden. Nach der Übersetzung liegt die Programmlänge vor und kann in den Quelltext eingesetzt werden. Dann ist das Programm erneut zu übersetzen.

Nach der Assemblierung stehen alle DEBUG-Meldungen (auch Fehlertexte) in der Datei VERSION.LST. Der Inhalt dieser Datei wird nachfolgend angezeigt.

```
-A CS:100
29E9:0100 ;-----------------------------------------------
29E9:0100 ; Beispiel zur Assemblierung mit dem DOS-Debugger
29E9:0100 ; aus einer Textdatei.
29E9:0100 ;
29E9:0100 ; Aufruf:  DEBUG < VERSION.ASM > VERSION.LST
29E9:0100 ;
29E9:0100 ; In der Datei VERSION.LST werden alle Meldungen
29E9:0100 ; (auch Fehlermeldungen) des Debuggers abgelegt.
29E9:0100 ; Die lauffähige COM-Datei findet sich in
29E9:0100 ; VERSION.COM
29E9:0100 ;-----------------------------------------------
29E9:0100
-A CS:200
29E9:0200 ;
29E9:0200 ; ORG 200
29E9:0200 ;
29E9:0200 ; Assembliere die Text-Konstanten ab CS:200
29E9:0200 ; statischer Text
29E9:0200 DB "MS-DOS Version $"
29E9:0210 ; CR,LF
29E9:0210 DB 0A,0D,"$"
29E9:0213 ;
29E9:0213 ; Assembliere den Programmcode ab CS:100
29E9:0213 ;
29E9:0213
-A CS:100
29E9:0100 ;
29E9:0100 ; ORG 100
29E9:0100 ;
29E9:0100 MOV DX,0200             ; lade Adr. String 1
29E9:0103 MOV AH,09               ; Ausgabe des Textes
29E9:0105 INT 21                  ; per INT 21
29E9:0107 MOV AH,30               ; Abfrage der DOS-
29E9:0109 INT 21                  ; Version
29E9:010B MOV DL,30               ; Convert Main Nr.
29E9:010D ADD DL,AL               ;     "
29E9:010F MOV AH,02               ; Ausgabe Character
29E9:0111 INT 21                  ;     "
29E9:0113 MOV DL,2E               ; write "."
29E9:0115 INT 21                  ;     "
29E9:0117 MOV DL,30               ; Convert Second Nr.
29E9:0119 ADD DL,AH               ;     "
29E9:011B INT 21                  ; write char.
```

```
29E9:011D MOV DX,0210             ; lade Adr. String 2
29E9:0120 MOV AH,09               ; Ausgabe CR,LF
29E9:0122 INT 21                  ; per INT 21
29E9:0124 MOV AX,4C00             ; Terminate Process
29E9:0127 INT 21                  ; normal
29E9:0129 NOP                     ; Ende des Programmcodes
29E9:012A ;-------------------------------------------
29E9:012A ; END     Leerzeile beendet Assemble Mode
29E9:012A ;-------------------------------------------
29E9:012A ; speichere den Code in der Datei VERSION.COM
29E9:012A ;-------------------------------------------
29E9:012A
-N VERSION.COM
-R BX
BX 0000
:0
-R CX
CX 0000
:200
-W
Schreibe 0200 Bytes
-Q
```

*Listing 7.2: VERSION.LST*

Die Umleitung der DEBUG-Ausgaben läßt sich mit:

DEBUG < VERSION.ASM

auf den Bildschirm leiten. Wird als Ausgabeeinheit PRN: angegeben, erzeugt DEBUG sofort ein Listing auf dem Printer.

### 7.1.18 Der UNASSEMBLE-Befehl

Als weitere Funktion bietet DEBUG einen eingebauten Disassembler. Diese Funktion besitzt folgende Aufrufsyntax:

U [adresse] [range]

Als Segmentadresse wird normalerweise CS angenommen. Die Startadresse liegt bei CS:100, wenn nichts anderes spezifiziert wurde. Sofern Sie den Code aus dem ersten Beispiel (Versionsabfrage im Assemblemodus) ab Adresse CS:100 assembliert haben, geben sie den Befehl ein:

-U 100 127

Auf dem Bildschirm wird in etwa folgende Ausgabe erscheinen:

```
1007:0100 BA0001      MOV   DX,0100
1007:0103 B409        MOV   AH,09
```

```
1007:0105 CD21          INT     21
1007:0107 B430          MOV     AH,30
 .....
```

Obiger Ausdruck enthält die rückübersetzten Maschinencodebefehle des Programmes im angegebenen Adressbereich.

Wird eine Länge eingegeben, versucht UNASSEMBLE mindestens die Zahl der Bytes zu bearbeiten. Ohne Längenangaben werden ca. 20 Bytes bearbeitet. Die genaue Länge richtet sich aber nach dem Befehlstyp, da nur vollständige Befehle decodiert werden. Die Startadresse muß auf einer gültigen Befehlsadresse liegen, da sonst ungültige Ausgaben erzeugt werden. Fehlt die Startadresse, beginnt die Disassemblierung mit der durch das Register IP spezifizierten Adresse. Es soll aber noch eine weitere Besonderheit erwähnt werden. Geben Sie bitte den folgenden Text ein:

```
-E CS:100 80 00 55 82 00 55
-A 106
1007:0106 JC  100
1007:0108 JNE 100
1007:010A JNA 100
1007:010C
```

Der UNASSEMBLE-Befehl:

```
-U 100 10C
1007:0100 800055        ADD   BYTE PTR [BX + SI],55
1007:0103 820055        ADD   BYTE PTR [BX + SI],55
1007:0106 72F8          JB    0100
1007:0108 75F6          JNZ   0100
1007:010A 76F4          JBE   0100
-
```

bringt ein merkwürdiges Ergebnis. Obwohl zwei verschiedene Opcodes 80 und 82 eingegeben wurden (siehe Codes vor den Befehlen), zeigt der Disassembler in beiden Fällen einen identischen (ADD) Befehl an. Die assemblierten Sprungbefehle werden auch anders wiedergegeben. Die Entwickler des 8086 haben den gleichen Befehl unter mehreren Opcodes implementiert. Daher taucht der ADD-Befehl zweifach auf. Die falschen Sprungbefehle finden ebenfalls eine Erklärung. INTEL erlaubt für eine Abfrage mehrere Mnemonics (symbolische Assemblerbefehle). So sind z.B. JB, JNAE, JC drei gültige Sprungbefehle mit der gleichen Bedingung, die nur unterschiedlich formuliert wurde. Der Assembler akzeptiert alle drei Anweisungen, setzt sie aber in einen Opcode um. Der Disassembler kann nun nicht mehr erkennen, welche Bedingung ursprünglich formuliert wurde. Er wird daher immer nur eine Bedingung "JC" zurückgeben. Bei den SHIFT-Befehlen sind in DEBUG nicht alle Varianten implementiert. Der Disassembler ist trotzdem sehr hilfreich um Programmbereiche in die entsprechenden Assemblerbefehle zurückzuwandeln.

### 7.1.19 Programmtests mit DEBUG

Nachdem nun alle Techniken zur Programmentwicklung, zum Laden und Speichern von Daten, sowie die Befehle zur Speicherbearbeitung besprochen sind, kommen wir zum letzten Punkt. Was noch fehlt ist die Kenntnis, wie der Debugger zum Programmtest einzusetzen ist. Hierzu bietet DEBUG durchaus einige Funktionen.

### 7.1.20 Der GO-Befehl

Normalerweise ist es notwendig, die Programme zum Test unter der Kontrolle von DEBUG ablaufen zu lassen. Nur dann ist ein Test möglich. Für diesen Zweck stellt DEBUG die GO-Funktion zur Verfügung. Die allgemeine Aufrufsyntax lautet:

G [=adress] [adress] [adress] ..

Wird keine Adresse eingegeben, übernimmt GO den Inhalt der CS und IP-Register und startet den Programmablauf. Damit verliert DEBUG die Kontrolle über die Programmausführung. Durch Eingabe der optionalen Startadresse (markiert durch ein vorangestelltes Gleichheitszeichen):

G = 100

benutzt der GO-Befehl diese als Startadresse. Dabei gelten die üblichen Eingabekonventionen. Fehlt die Segmentangabe, übernimmt DEBUG den Inhalt des CS-Registers. Wichtig ist die Eingabe des Gleichheitszeichens vor der ersten Adresse. Damit läßt sich der Programmablauf in jedem beliebigen Schritt starten. Es ist darauf zu achten, daß an der spezifizierten Adresse ein gültiger Befehl beginnt, da sonst undefinierte Effekte auftreten. Mit:

-G =100 127

wird ein Teil des Programmes ausgeführt und ab der Adresse 127 zu DEBUG zurückgekehrt. Offenbar setzt DEBUG im Programm Unterbrechungspunkte. Wird kein Haltepunkt erreicht und DEBUG meldet:

Programm terminated normally

dann muß das Programm vor der nächsten Ausführung erneut geladen werden. Ohne Segmentangabe übernimmt der GO-Befehl den Inhalt des CS-Register. Mit der Eingabe:

-G 113 126

werden zwei Haltepunkte im Codesegment gesetzt. Die Programmausführung beginnt mit den aktuell gesetzten Werten im CS:IP Register, da die Option =*adress* nicht verwendet wurde.

### 7.1.21 Der TRACE-Befehl

Als weitere Funktion stellt DEBUG noch den TRACE-Befehl zur Verfügung. Das Aufrufformat lautet:

T [= adress] [value]

TRACE beginnt mit der Abarbeitung des Programmcodes ab der angegebenen Adresse in Form von Einzelschritten, wobei nach jedem Schritt die Registerinhalte und der nächste Befehl angezeigt werden. Wird eine Adresse angegeben, beginnt der Programmablauf bei dieser Adresse. Bei fehlender Segmentangabe übernimmt DEBUG den Inhalt des CS-Registers. *Value* gibt an, wieviele Einzelschritte durchzuführen sind. Mit:

-T = 100

wird z.B. das erste Beispielprogramm zur Versionsabfrage gestartet und der Inhalt der Register wird nach Ausführung des ersten Befehls gezeigt. Mit:

-T 10

werden weitere 16 Schritte (10H) ausgeführt. Da TRACE alle Hardware Interrupts vor Ausführung eines Befehls sperrt, darf das Benutzerprogramm die Interruptmaske des 8295 Controllers nicht verändern. Andernfalls sind unkontrollierbare Effekte möglich. Wird im Anwenderprogramm der INT 3 (Trap) benutzt, setzt TRACE auf den INT 3 Code einen Unterbrechungspunkt.

### 7.1.22 Der PROCEED-Befehl

Enthält ein Programm viele Unterprogrammaufrufe, ist es störend, daß mit TRACE jedesmal auch der Code des Unterprogrammes mit angezeigt wird. Beim Aufruf der DOS-INT 21-Funktionen mit TRACE kann es sogar zu Systemabstürzen kommen. Hier bietet die PROCEED-Funktion Hilfestellung. Der Aufruf:

P [Adresse] [Wert]

führt dazu, daß bei Unterprogrammaufrufen, Interrupts oder String-Move-Befehlen der Programmablauf erst nach der Ausführung unterbrochen wird. Bei anderen Befehlen verhält sich PROCEED wie die TRACE-Option. Nach Ausführung der Instruktion oder des Unterprogrammes läßt sich dann der Registerinhalt untersuchen. Außerdem wird eine Wiederholung des Befehls möglich. Die Adreßkonventionen entsprechen

den bereits behandelten Regeln. Wert gibt an, wie oft eine Funktion wiederholt wird, bevor eine Unterbrechung auftritt. Allerdings ist dieser Befehl nicht bei allen DEBUG-Versionen implementiert.

### 7.1.23 Die Expanded-Memory-Befehle

DEBUG besitzt ab DOS 4.0 einige Befehle um auf das Expanded-Memory zuzugreifen. Diese Befehle möchte ich aber hier nicht vorstellen. Gegebenenfalls ist die Originalliteratur oder /2/ zu konsultieren.

### 7.1.24 Anmerkungen zu DR-DOS 5.0/6.0

DR-DOS besitzt das Programm DEBUG.COM nicht, vielmehr wird der Debugger SID.EXE mitgeliefert. Dieses Programm weicht in der Kommandosyntax bei den Assembleranweisungen etwas von DEBUG ab. Die wesentlichen Änderungen beim Debuggen betreffen die Kommandosyntax: SID ist eine bildschirmorientierte Version. Der Aufruf erfolgt mit:

SID <Parameter>

Als Parameter lassen sich Dateinamen und weitere Optionen angeben. Die DOS-Ein-/Ausgabeumleitung kann wie bei DEBUG verwendet werden. Nach dem Start meldet sich SID mit:

```
------------------------------------------------
*** Symbolic Instruction Debugger ***  Release 3.1
     Copyright (c) 1983,1984,1985,1988,1990
    Digital Research, Inc. All Rights Reserved
------------------------------------------------
#
```

Als PROMPT wird das Zeichen # benutzt; beendet wird SID mit dem Kommando Q. Die Befehle des Debuggers lassen sich mit:

#?

abrufen. Auf dem Bildschirm erscheint eine kurze Meldung mit einer Auflistung der möglichen Eingabebefehle. Neben dem Fragezeichen zur Aktivierung den Online-Hilfe stehen verschiedene Befehle zur Verfügung. Für eine genaue Beschreibung sind die DR-DOS Unterlagen zu konsultieren.

Mit der Eingabe ?? läßt sich die genaue Syntax der einzelnen Befehle am Bildschirm abfragen. Leider hat SID aus Sicht des Assemblerprogrammiers einen weiteren Nachteil, die Kommandos sind nicht ganz kompatibel zu DEBUG.COM. Deshalb müssen die Listings für DR-DOS in einigen Punkten angepaßt werden. Nachfolgend möchte ich die wichtigsten Änderungen stichpunktartig vorstellen:

- Alle Kommentare sind aus dem Listing zu entfernen.
- Alle impliziten JMP-Anweisungen (z.B. JMP 100) müssen in SID mit einem Prefix (Short) versehen werden (z.B. JMP SHORT 100 wird zu JMPS 100).
- Die Assemblierung muß mit Axxx beginnen (z.B. A100).
- Die Länge des Programmcodes ist in SID mit der Anweisung: *Filename.COM Start Länge* abzuspeichern.

Mit Filename ist der Dateiname der zu erzeugenden COM-Datei gemeint. Der Parameter *Start* enthält die Offsetadresse ab der der Programmcode beginnt. In *Länge* wird die Zahl der zu speichernden Bytes definiert. Nach diesen Modifikationen sollten sich obige Programme übersetzen lassen. Falls bei weiteren Befehlen doch Probleme auftreten konsultieren Sie bitte die DR-DOS Handbücher.

***Anmerkung:*** In Novell DOS 7.0 besitzt der Debugger die gleiche Syntax wie das DOS-Pendant.