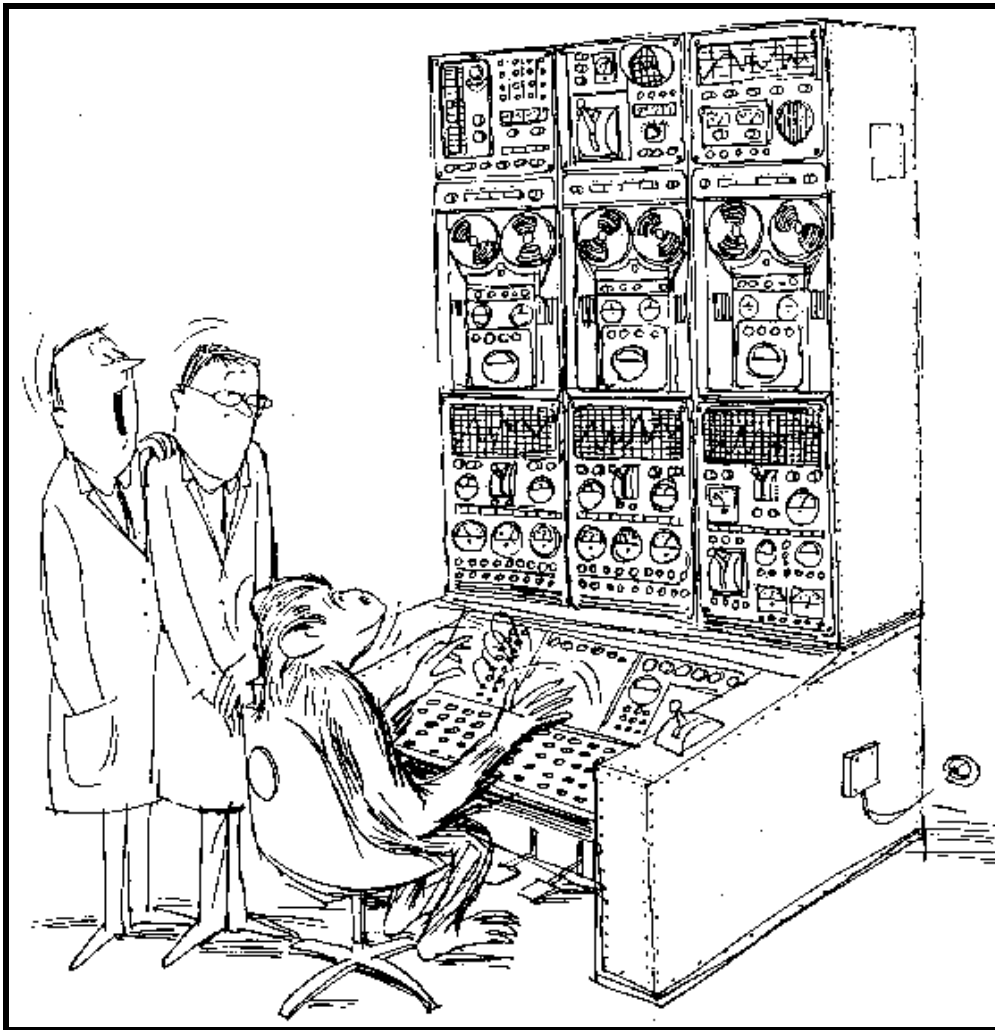


Programmierung

(nach einer Vorlesung von Herrn Prof. Patzelt*,
Sommersemester 1996, FH Dortmund)



* „Natürlich könnte ich es Ihnen erklären. Aber das müssten Sie schon alles wissen.
Wenn Sie es noch nicht wissen sollten, lernen Sie es eh besser aus Büchern.
Was man sich selbst erarbeitet hat, behält man doch auch viel besser.
Im übrigen: Wenn Sie noch nicht das Programmieren beherrschen,
warum studieren Sie eigentlich Informatik?“

Inhaltsverzeichnis

1. Das Computer-Zahlensystem	1
1.1 Das binäre Zahlensystem	1
1.2 Das hexadezimale Zahlensystem	2
2. Grundlagen der INTEL-Prozessoren	3
2.1 Geschichte der PC-Prozessoren	3
2.2.1 Arbeitsregister (auch Datenregister):	3
2.2.2 Segmentregister:	4
2.2.4 Indexregister:	4
2.2.5 Befehlszeiger (Instruction pointer):	5
2.2.6 Kennzeichen (Flags):	5
2.2.6.1 Carry-Flag:	6
2.2.6.2 Overflow-Flag:	6
2.2.6.3 Auxiliary-Flag:	6
2.2.6.4 Sign-Flag:	6
2.2.6.5 Zero-Flag:	6
2.2.6.6 Parity-Flag:	6
2.3 Die Adressierung:	7
2.3.1 Berechnung der physikalischen Adresse:	7
2.3.2 Adressierung mittels Register:	7
2.3.2.1 Operand im Register	7
2.3.2.3 Inhärente Adressierung	8
2.3.2.3 Unmittelbare Adressierung	8
2.3.3 Adressierung mittels Speicheroperand:	8
2.3.3.1 Direkte Adressierung	8
2.3.3.2 Indirekt indizierte Adressierung	8
2.4 INTEL-Konventionen:	9
3. Grundelemente der Assemblersprache (TASM = Turbo-Assembler)	10
3.1 Datentypen in Assembler	10
3.2 Der Aufbau einer Assemblerzeile	10
3.2.1 Das Namensfeld („label“)	10
3.2.2 Trennzeichen	11
3.2.3 Operationsfeld	11
3.2.4 Operandenfeld	11
3.2.5 Kommentare	12
3.3 Pseudooperationen	12
3.3.1 ORG-Anweisung	12
3.3.2 EQU-Anweisung	12
3.3.3 '='-Anweisung	13
3.3.4 DUP-Anweisung	13
4. Struktur eines Assemblerprogramms	14
4.1 Segmentierung	14
4.1.1 Codesegment	15
4.1.2 Datensegment	15
4.1.3 Stapelsegment	15
4.1.4 Vereinfachte Segmentzuweisungen (Turbo-Assembler)	15
4.1.5 Zugriff auf Segmente	15
4.1.5.1 Zugriff auf das Stapelsegment	16
4.1.5.2 Zugriff auf das Datensegment	16

4.2 'COM'- und 'EXE-Programme	17
4.2.1 Die 'COM'-Struktur	17
4.2.2 Die 'EXE'-Struktur	18
5. Grundtechniken der TASM-Programmierung	19
5.1 Elementare Befehle	19
5.1.1 Datentransport	19
5.1.1.1 MOV	19
5.1.1.2 PUSH und POP	19
5.1.1.3 IN und OUT	19
5.1.2 Arithmetische Operationen	20
5.1.2.1 Addition/Subtraktion	20
5.1.2.2 Multiplikation	20
5.1.2.3 Division	20
5.1.2.4 Inkrementierung und Dekrementierung	20
5.1.2.5 Negierung	20
5.1.3 Boolesche Operationen	20
5.1.3.1 AND	20
5.1.3.2 OR	21
5.1.3.3 NOT	21
5.1.3.4 XOR	21
5.2 Sprungbefehle	21
5.2.1 Unbedingter Sprung	21
5.2.2 Bedingter Sprung	21
5.2.2.1 Carry-Flag	23
5.2.2.2 Overflow-Flag	23
5.2.2.3 Sign-Flag	24
5.2.2.4 Zero-Flag	24
5.2.2.5 Parity-Flag	24
5.2.2.6 JCXZ und LOP	24
5.2.2.7 Fehlerbehandlung, ein Anwendungsbeispiel: Der CMP-Befehl	25
5.3 Unterprogramme	26
5.3.1 Aufruf von Unterprogrammen	26
5.3.1.2 Deklaration eines Unterprogramms	26
5.3.1.3 Versorgung mit Parametern	26
5.3.1.4 Entsorgung des Unterprogramms	27
5.3.1.5 Externe Unterprogramme	28
5.3.2 Aufruf von Makros	28
5.3.3 Einbindung von INCLUDE-Dateien	30
5.3.4 Aufruf von MS-DOS-Funktionen	30
5.4 Operatoren	31
5.4.1 Arithmetische Operatoren	31
5.4.2 Logische Operatoren	31
5.4.3 Vergleichoperatoren	32
5.4.4 Wertbestimmende Operatoren	32
5.4.5 Attributoperatoren	33
5.5 Dateitypen	34
5.6 Programme aus mehreren Modulen	34
5.6.1 Anweisung GLOBAL	35
5.6.2 PUBLIC	35
5.6.3 EXTRN	36

6. Die Schnittstelle zu Borland C++	37
6.1 Die Speichermodelle	37
6.2 C++ ruft ein Assembler-Unterprogramm auf	37
6.2.1 Parameter-Übergabe und Ergebnis-Wert	38
6.2.2 Retten von Registern	39
6.3 Assembler ruft C++-Funktion auf	40
6.4 Verwendung gemeinsamer Daten	40
7. Der numerische Koprozessor 8087	41
7.1 Interne Register	41
7.1.1 Der Stapel des 8087	41
7.1.2 Fließkommaformate	42
7.2 Datentypen	42
7.3 Befehlssatz	42
7.4 Der interne Aufbau	45
7.5 Datenaustausch zwischen 8088/8086 und 8087	47
8. Fehlersuche und -beseitigung	48
8.1 Syntaxfehler	48
8.2 Logische Fehler	48
8.3 Einsatz eines Debuggers	48
8.4 Debug-Strategie	48
8.5 Checkliste	49
Anhang A: Beispiele	50
Einbinden von Unterprogrammen	50
Equipment-Check	52
Drucker-Ansteuerung	53
Anhang B: Die BIOS-Funktionen	54
Anhang C: Übersicht der Funktionen des Interrupts 21h	57
Anhang D: Der Multiplexer-Interrupt 2Fh	61
Anhang E: Die Funktionen des XMS	62
Anhang F: Die Funktionen des EMM	63
Anhang G: Die Hardware-Interrupts	64
Anhang H: Maustreiber-Funktionen	67
Literaturhinweise	69

1. Das Computer-Zahlensystem

1.1 Das binäre Zahlensystem

Die Speicher eines Computers kennen nur zwei Zustände: „Ein“ oder „Aus“. Damit diese Speicherbausteine Zahlen beliebiger Größe darstellen können, wurde das binäre Zahlensystem eingeführt.

Die beiden Zustände „Ein“ und „Aus“ entsprechen den beiden Zahlen des binären Zahlensystems 1 und 0, daher basiert das binäre Zahlensystem auf die Basis 2.

Die schalterähnlichen Speicherbausteine werden „bits“ (binary digits) genannt. Ein Bit, welches auf „Ein“ geschaltet ist, hat den Wert 1, bei „Aus“ 0.

Auch Binärzahlen können kombiniert werden, um Zahlen größer 1 darzustellen.

Beim binären System ist jede Ziffer um den Faktor zwei größer als die vorhergehende. Die am weitesten rechts stehende Ziffer hat den Zahlenwert 2^0 (d.i. dezimal = 1), das darauf links folgende Bit den Wert $2^1 (=2)$, dann $2^2 =4$ etc.

Die binäre Zahl 10101 berechnet sich dezimal wie folgt:

$$\begin{aligned} 10101 &= (1 \cdot 2^4) + (0 \cdot 2^3) + (1 \cdot 2^2) + (0 \cdot 2^1) + (1 \cdot 2^0) \\ &= (1 \cdot 16) + (0 \cdot 8) + (1 \cdot 4) + (0 \cdot 2) + (1 \cdot 1) = 16 + 4 + 1 = 21. \end{aligned}$$

Der Stellenwert der ersten 8 Bits des Speichers lautet also:

	7	6	5	4	3	2	1	0	Bitposition
	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	binär
	128	64	32	16	8	4	2	1	dezimal

Möchte man jetzt einen dezimalen Wert in einen binärem umformen, so sind eine Reihe von Subtraktionen notwendig.

Zuerst wird der nächstkleinere binäre Wert von der Dezimalzahl subtrahiert. An der binären Speicher-Position dieser Zahl (Bitposition) wird eine 1 geschrieben.

Dann wird diese Subtraktion nach rechts zur Bitposition 0 fortgesetzt. An jeder Bitposition, wo deren Stellenwert (binärer Wert) nicht von der Dezimalzahl subtrahierbar ist, wird eine 0 eingetragen.

50		7	6	5	4	3	2	1	0
-32 (Bitposition = 5)	—			1					
18									
-16 (Bitposition = 4)	—			1	1				
2									
-2 (Bitposition = 1)	—							1	
0									
(restliche Positionen = 0)				1	1	0	0	1	0

Beispiel: dezimal 50 → binär ?

Das Endergebnis ergibt die binäre Zahl 110010.

Die Überprüfung der Rechnung geschieht durch Addition der „1er“ Bitpositionen:

$$2^5 + 2^4 + 2^1 = 32 + 16 + 2 = 50.$$

Acht Informationsbit werden ein **Byte** genannt. Diese acht Bits können dezimale Werte von 0 (binär = 00000000) bis 255 (binär = 11111111) darstellen.

Der Speicher von Computern wird in Blöcken zu 1024 Byte konstruiert, das entspricht 2^{10} Byte.

Für den Wert 1024 gibt es die beräuchliche Abkürzung Kbyte (KiloByte).

Binäre Zahlen werden ebenso wie Dezimalzahlen addiert: der Übertrag einer Spalte geht auf die nächste Spalte über.

Die Addition der binären Zahlen 1 + 1 gibt als Ergebnis 10 (Übertrag von 1 in die nächste Zweierspalte).

Als allgemeine Regel zum Rechnen mit binären Zahlen gilt:

Eingabe			Ergebnis	
Operand 1	Operand 2	Übertrag	Summe	Übertrag
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Zahlen mit Vorzeichen werden dargestellt, indem das höchstwertigste Bit (Bitposition = 7) das Vorzeichen der Zahl darstellt: ist die Zahl positiv, so gilt 2^7 ist auf 0 gesetzt, bei negativen Zahlen steht in der Bitposition 7 eine 1. Bytes mit positiven Zahlen können daher Werte von 0 (binär = 00 00 00 00) bis +127 (binär = 01 11 11 11) annehmen, Bytes mit negativen Zahlen können Werte von -1 (binär = 11 11 11 11) bis -128 (binär = 10 00 00 00) enthalten. Negative Zahlen werden durch ihr **Zweierkomplement** dargestellt. Um die binäre Darstellung einer negativen Zahl (d.i. das Zweierkomplement) herauszufinden wird von der positiven Darstellung jedes einzelne Bit invertiert (0 → 1, 1 → 0) und zu dem Ergebnis wird noch eine 1 addiert.

Beispiel: Dezimalzahl -32 → binär ?

- 1) +32 in binärer Schreibweise

32	
<u>-32</u>	(Bitposition = 5 auf 1 setzen)
0	(d.h. restliche Positionen mit 0 auffüllen).

 → binäre Zahl = 00 10 00 00
- 2) alle Bits der binären Zahl invertieren

00 10 00 00
11 01 11 11
- 3) zu dem Ergebnis eine 1 addieren

11 01 11 11
<u> + 1</u>
11 10 00 00 = Zweierkomplement

Ergebnis: dezimal -32 = binär 11 10 00 00.

Ebenso wird aus einer negativen Zahl die positive herausgefiltert: alle Bits invertieren, 1 addieren.

1.2 Das hexadezimale Zahlensystem

Da selten mit einzelnen Bits sondern fast immer mit Bitgruppen gearbeitet wird entwickelte sich alternativ zum binären Zahlensystem das hexadezimale Zahlensystem, welches auf der Basis 16 (für eine vierer-Bitgruppe) basiert. Mit vier Bits können die binären Werte 00 00 (dezimal 0) bis 11 11 (dezimal 15) dargestellt werden. Dies sind insgesamt 16 Kombinationsmöglichkeiten, jede Ziffer in einem Zahlensystem stellt eine dieser 16 Kombinationen dar. Von den 16 Zahlen des hexadezimalen Zahlensystems werden die ersten 10 durch die Zahlen 0 bis 9 (das entspricht auch den dezimalen Zahlenwerten 0 bis 9) und die letzten sechs Zahlen durch die Buchstaben A bis F (die Dezimalwerte 10 bis 15) dargestellt:

hexadezimale Ziffer	binärer Wert	dezimaler Wert
0	00 00	0
1	00 01	1
2	00 10	2
3	00 11	3
4	01 00	4
5	01 01	5
6	01 10	6
7	01 11	7
8	10 00	8
9	10 01	9
A	10 10	10
B	10 11	11
C	11 00	12
D	11 01	13
E	11 10	14
F	11 11	15

Auch jede hexadezimale Ziffer hat einen Stellenwert, der einer Potenz seiner Basis entspricht.

Das das hexadezimale Zahlensystem mit der Basis 16 arbeitet, hat jede hexadezimale Ziffer einen Stellenwert, der 16mal größer ist als die rechts gelegene Ziffer:

7	6	5	4	3	2	1	0	Bitposition
16^7	16^6	16^5	16^4	16^3	16^2	16^1	16^0	hexadezimal
268 435 456	16 777 216	1 048 576	65 536	4 096	256	16	1	dezimal

Beispiel: hexadezimal 3AF → dezimal ?

$$(3 * 16^2) + (A * (16^1)) + (F * 16^0) = (3 * 256) + (10 * 16) + (15 * 1) = 768 + 160 + 15 = 943.$$

2. Grundlagen der INTEL-Prozessoren

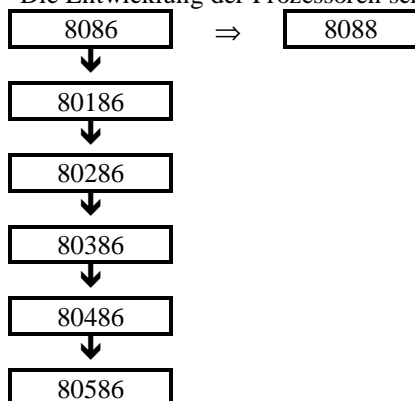
2.1 Geschichte der PC-Prozessoren

Mit der Entwicklung des 8086-Prozessors ist es Intel erstmalig gelungen, Daten mit einer Breite von 16 Bit parallel zu verarbeiten.

Dabei wurde das Konzept verfolgt, daß Programme zu dem älteren 8-Bit-Microprozessor 8080A aufwärtskompatibel sind, d.h. Programme älterer Prozessoren laufen auch auf den neuen Prozessoren.

Bei der Kombination 8086-8080A ist die Kompatibilität auf die Maschinensprache beschränkt, die Prozessoren der 8086er Reihe sind jedoch softwarekompatibel.

Die Entwicklung der Prozessoren scheint kein Ende zu nehmen.



Seit dem 80486 ist sogar eine 32-Bit-Datenbreite erreicht worden.

Der 8088-Prozessor unterscheidet sich jedoch ein bißchen von den anderen Prozessoren der großen Intel-Familie: er verarbeitet intern 16-Bit, überträgt extern jedoch nur 8 Bit.

Bei der Programmierung sollte man jedoch bedenken das eine Aufwärtskompatibilität zwar gegeben ist, moderne Prozessoren einen erweiterten und komplexeren Befehlssatz haben als ihre Vorgängerversionen.

Ein Programm, welches 80486-Befehle benutzt, läuft nicht zwangsläufig auch auf einen 80386 oder 8086/8088.

2.2 Prozessor-Register

2.2.1 Arbeitsregister (auch Datenregister):

Der Prozessor verfügt über vier Arbeitsregister.

Diese werden in der Regel explizit mit Werten versorgt und müssen auch ausdrücklich angesprochen werden.

	31 16	15 0
AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL

Jedes dieser Register hat bei Rechnern bis zum 80286 eine Größe von 16 Bit, ab dem 80386 sind dies 32-Bit-Register. Die Bits werden immer von Null an gezählt.

Diese Arbeitsregister können aber auch als 8-Bit-Register angesprochen werden (z.B. bei der Bearbeitung von ASCII-Zeichen, die 8 Bit lang sind).

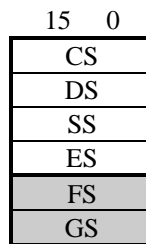
Die Unterteilung der Arbeitsregister ist wie folgt:

Register	Bezeichnung	Verwendungsbeispiel
AX	Akkumulator	- Daten für Ein-/Ausgaberroutinen - Unmittelbare Adressierung - arithmetische Operationen
BX	Basisregister	Bildung von Speicheradressen
CX	Zählregister (count register)	- Schleifenzähler
DX	Richtungsregister (Datenregister)	- Adressierung von Ein-/Ausgabekanälen - arithmetische Operationen

Die Buchstaben „L“ und „H“ kennzeichnen hierbei das niederwertige (L = low order byte) oder höherwertige (H = high order byte) des 16-Bit-“X“-Registers.

2.2.2 Segmentregister:

Computer mit dem 8086 oder 8088 halten Programme und Daten in vier verschiedenen Bereichen des Speichers. Die Programm- und Datenbereiche können bis zu 64 Kbyte groß sein und werden Segmente genannt. Der 8088 kann mit bis zu vier Segmenten gleichzeitig arbeiten. Er hält die Startadressen dieser Segmente in seinen vier Segmentregistern:



Im einzelnen haben diese vier Segmentregister folgende Funktionen:

Register	Bezeichnung	Bedeutung
CS	Codesegment	zeigt auf das Segment, welches das momentan ausgeführte Programm enthält, der 8088 kombiniert den Inhalt von CS (multipliziert mit 16) mit dem Inhalt des Befehlszeigers (IP), um die Speicheradresse des nächsten auszuführenden Befehls zu erhalten
DS	Datensegment	zeigt auf das aktuelle Datensegment, in dem sich in der Regel die Variablen befinden
SS	Stapelsegment	zeigt auf das aktuelle Stapelsegment; ein Stapel ist eine Datenstruktur im Speicher, die als zeitweiser Ablageplatz für Daten und Adressen verwendet wird; der 8088 benutzt den Stapel, um sich die Rückkehradresse zu merken, wenn eine Unteroutine ausgeführt wird oder um die Werte von Registern zu sichern, die eine Unteroutine verändert
ES	Extrasegment	zeigt auf das aktuelle Extrasegment, welches bei Zeichenkettenoperationen verwendet wird
FS	Extrasegment	ab 80386: frei verfügbar
GS	Extrasegment	ab 80386: frei verfügbar

In Systemen, welche nicht mehr als 64 Kbyte Speicher haben, überlappen sich diese Segmente häufig.

2.2.3 Zeigerregister:

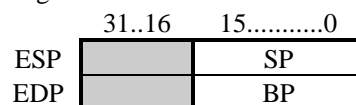
Der 8088 greift auf Daten in anderen Segmenten zu, indem er die Basisadresse in einem Segmentregister mit dem Offset in einem anderen Register kombiniert.

Wenn er auf das Stapelsegment zugreift, holt der 8088 sich die Basisadresse aus dem Stapelsegmentregister (SS) und den Offset aus einem Zeigerregister (SP oder BP).

Um auf das Datensegment zuzugreifen, verwendet er die Basisadresse aus dem Datensegmentregister (DS) und den Offset aus einem Basisregister (BX) oder aus einem der Indexregister (SI oder DI).

Die Zeigerregister dienen dem Zugriff auf Daten, die auf dem sogenannten Stapel (= Stack) zwischengespeichert werden.

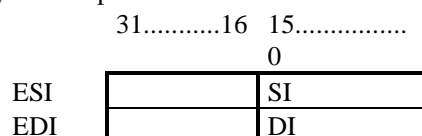
Der Stackpointer (SP) zeigt normalerweise auf das zuletzt auf dem Stapel gespeicherte Datum. Sein Inhalt wird bei PUSH- bzw. POP-Anweisungen impliziert, d.h. automatisch verändert. Der Basepointer (BP), der ebenfalls auf den Stapel zeigt, wird vom Programmierer gesetzt.



2.2.4 Indexregister:

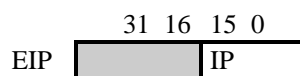
N mit den beiden Indexregistern SI (=Source Index, Quellindex) und DI (=Destination Index, Zielindex) werden Speicherplätze adressiert, wenn zum Beispiel Zeichenketten (Strings) bearbeitet werden sollen.

Bei einigen arithmetischen und logischen Operationen können sie aber auch Operanden enthalten:



2.2.5 Befehlszeiger (Instruction pointer):

Der 8088 ordnet die zwei Aufgaben „Befehle lesen“ und „Befehle ausführen“ zwei separaten Funktionseinheiten im Chip zu. Eine der Busschnittstelleneinheiten (bus interface unit -BIU) ist allein verantwortlich für das Lesen von Befehlen aus dem Speicher und die Übergabe von Daten zwischen der Ausführungshardware und der Außenwelt. Die andere Ausführungseinheit (execution unit - EU) führt nur Befehle aus. Beide Einheiten arbeiten unabhängig voneinander, d.h. die BIU kann Befehle aus dem Speicher lesen, während gleichzeitig die EU einen vorher gelesenen Befehl ausführt. Jedesmal, wenn die BIU einen Befehl liest, fügt sie diesen Befehl in eine Befehlswarteschlange im Mikroprozessor ein. So findet die EU nach Beendigung eines Befehls den nächsten in der Regel bereits in der Warteschlange. Da die BIU jedoch nicht weiß, in welcher Reihenfolge die Befehle eines Programms ausgeführt werden sollen, liest es Befehle immer aus aufeinanderfolgenden Speicherstellen. Die EU braucht also nur auf das Lesen eines Befehls zu warten, wenn die Programmausführung an einer anderen, nicht direkt auf den aktuellen Befehl folgenden Stelle fortgesetzt wird. Zu diesem Zeitpunkt muß die EU darauf warten, daß die BIU die Warteschlange löscht und einen neuen Befehl liest. Dieses „Nächste-Ausführungs-Adresse“-Register wurde von den Intel-Entwicklern Befehlszeiger (=instruction pointer, IP) genannt. Der Befehlszeiger enthält immer den Offset des nächsten Befehls, der von der EU des 8088 ausgeführt wird. Das das IP-Register ein Register mit besonderer Bedeutung ist, können damit keine Berechnungen angestellt werden. Der 8088 hat aber Befehle, welche den IP verändern oder auf den Stapel übertragen können.



2.2.6 Kennzeichen (Flags):

Für den Ablauf eines Programms ist der Inhalt des Prozessor-Status-Registers (=Processor Status Word, PSW) von großer Bedeutung.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PSW		N	IOPL	O	D	I	T	S	Z		A		P			C

Dabei unterscheidet man die einzelnen Bitpositionen (Flags):

zum einen gibt es die Steuer-Flags, die der Programmierer setzt (Inhalt: 1) bzw. löscht (Inhalt: 0),

zum anderen die Ergebnis-Flags, die nach der Ausführung verschiedener Befehle vom Prozessor automatisch gesetzt oder gelöscht werden.

Durch eine Abfrage des Zustands einzelner Flags kann der Programmierer auf die unterschiedlichen Situationen gezielt reagieren. So deutet ein gesetztes Carry-Flag (Bitposition: 0, Inhalt: 1) nach dem Aufruf einer MSDOS-Funktion in der Regel auf einen aufgetretenen Fehler hin.

Ab dem 80386 existiert ein weiteres Flag-Register:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PSW															V	R

Die Bedeutung der Steuer-Flags sind wie folgt:

Flag	Bezeichnung	ab	Wert	Bedeutung
D	Direction	8086	1	Stringbefehle werden mit fallender Adresse bearbeitet
			0	Die Bearbeitung erfolgt mit steigender Adressierung
I	Interrupt Enable	8086	1	Externe maskierbare Interrupts können das gerade laufende Programm unterbrechen
			0	derartige Interruptanforderungen werden ignoriert
T	Trap	8086	1	Einzelschrittmodus, d.h. nach jedem Befehl eines Programms wird der Prozessor durch einen internen Interrupt angehalten
C	Carry	8086	1	ein Übertrag aus dem höchstwertigen Bit des Ergebnisses heraus ist erfolgt
O	Overflow	8086	1	Überschreitung des zulässigen Wertebereiches beim Rechnen mit vorzeichenbehafteten Zahlen (Zweier-Komplement)
A	Auxiliary	8086	1	Interner Übertrag von Bit 3 nach Bit 4 ist aufgetreten
S	Sign	8086	1	negatives Ergebnis
			0	positives Ergebnis
Z	Zero	8086	1	Das Ergebnis einer Rechenwerksoperation ist gleich Null
P	Parity	8086	1	Im Low-Byte des Ergebnisses ist die Zahl der auf „1“ gesetzten Bits gerade

Ab dem 80286 kamen folgende Flags hinzu:

Flag	Bezeichnung	ab	Wert	Bedeutung
IOPL	I/O Privilege Level	80286		Ist dieses Bit gesetzt, arbeitet der Prozessor im Protected Mode, d.h. eine Unterstützung durch MS-DOS besteht dann nicht mehr
			1	Ein-/Ausgabe nur für Tasks mit Priorität 1 oder 0
			0	Ein-/Ausgabe nur für Tasks mit höchster Priorität
N	Nested Task	80286	1	wird bei Prozeßverschachtelung im Protected Mode gesetzt

Die 80386-Generation fügte noch folgende Steuerflags hinzu:

Flag	Bezeichnung	ab	Wert	Bedeutung
R	Resume	80386	1	diese Bit legt fest, ab welcher Ebene ein Task (Programm im sogenannten 'Protected Mode') Ein- bzw. Ausgabeoperationen durchführen kann
V	Virtual 86 Mode	80386	1	es werden vier 8086-Prozessoren simuliert

2.2.6.1 Carry-Flag:

In das Übertrags- oder Carry-Flag wird nach arithmetischen Operationen ein Übertrag aus der höchstwertigen Stelle einer Zahl heraus eingetragen.

Es wird bei manchen „Schiebebefehlen“ benutzt und kennzeichnet bei vielen MS-DOS-Funktionen das Auftreten eines Fehlers.

2.2.6.2 Overflow-Flag:

Das Overflow-Flag zeigt an, ob der Gültigkeitsbereich für vorzeichenbehaftete Zahlen überschritten wurde. Bei diesen Zahlen gilt das höchste Bit als Vorzeichen: eine 0 an dieser Stelle kennzeichnet positive Zahlen, eine 1 dementsprechend negative Zahlen.

Dieses Flag wird gebildet durch eine EXOR-Verknüpfung des internen Übertrags in die Vorzeichenstelle hinein mit dem externen Übertrag aus der Vorzeichenstelle heraus.

2.2.6.3 Auxiliary-Flag:

Die gleiche Funktion wie das Übertragsflag nimmt das Auxiliary-Flag bei der Dezimalarithmetik mit BCD-Zahlen wahr, bei der ein Nibble (4 Bit) für das Speichern einer Ziffer verwendet wird.

Es wird immer dann gesetzt, wenn ein Übertrag von Bit 3 nach Bit 4 auftritt.

Beispiel:

dual	hexadezimal	dezimal	
1010 0101b	A5h	165	
+ 0001 1101b	+ 1Dh	+ 29	
+ 111 1 1 b	+ 1 h	1	Übertrag
1100 0010b	C2h	194	

2.2.6.4 Sign-Flag:

Für die Generierung des Vorzeichen- oder Sign-Flags wird einfach nur eine Kopie des Vorzeichenbits des Ergebnisregisters angelegt. Über dieses Flag kann daher recht einfach ein positives oder negatives Ergebnis einer arithmetischen Operation ermittelt werden. Diese Kopie des höchstwertigen Bit wird jedoch immer angelegt, d.h. auch, wenn nicht mit vorzeichenbehafteten Zahlen operiert wird.

2.2.6.5 Zero-Flag:

Das Zero-Flag wird immer dann gesetzt, wenn das Ergebnis einer Operation gleich Null ist.

2.2.6.6 Parity-Flag:

Mit dem Begriff „Parity“ wird die Möglichkeit der Kontrolle der übertragenen Bits bezeichnet. Die Zahl der gesendeten Bits mit dem Wert „1“ wird erfaßt. Ist das Ergebnis gerade, spricht man von gerader Parität.

Wenn bei der seriellen Kommunikation eine ungerade Paritätsprüfung verlangt wird, wird in diesem Fall zusätzlich ein „1“-Bit übertragen, damit die Zahl der gesetzten Bits wieder ungerade ist. Wäre die Zahl der gesendeten „1“-Bits ungerade, wird ein „0“-Bit nachgesendet.

In ähnlicher Weise verhält sich das Parity-Flag der 8086-CPU. Hier werden jedoch die niederwertigen 8 Bit des Ergebnisregisters untersucht.

Das Parity-Flag wird dann gesetzt, wenn sich eine gerade Anzahl von „1“-Bits ergibt.

2.3 Die Adressierung:

2.3.1 Berechnung der physikalischen Adresse:

Die 16 bzw. 32-Bit breiten Segmentregister enthalten die Adresse, an welcher ein Segment beginnt. Diese Adresse wird als Segmentbasisadresse bezeichnet.

In Kombination mit einem weiteren 16-Bit-Register, welches die effektive Adresse

Segmentbasisadresse	XXXX XXXX XXXX XXXX 0000	(b, binär)
effektive Adresse	+ 0000 YYYY YYYY YYYY YYYY	(b, binär)
absolute Adresse	ZZZZ ZZZZ ZZZZ ZZZZ YYYY	(b, binär)

innerhalb eines Segments enthält, wird die Adressierung vervollständigt.

Im Real Mode der 8086-Prozessoren wird dabei folgende Methode angewendet:

Durch diesen Trick erhält man eine 20-Bit-Zahl. Die Breite entspricht genau der Anzahl von Adreßleitungen des 8086/8088-Prozessors.

Mit der absoluten 20-Bit-Adresse kann nun ein Speicher mit einer Größe von

$$2^{20} \text{ Byte} = 1\,048\,576 \text{ Byte} = 1024 \text{ Kbyte} = \text{Megabyte}$$

verwaltet werden.

Im Maschinenprogramm wird die absolute Adresse durch die Schreibweise

Segmentadresse : effektive Adresse

angegeben, z.B.

```
mov ax, DS:SI      ; DS: Segmentbasisadresse
                   ; SI: effektive Adresse.
```

Wird ein Speicherplatz in dieser Form angesprochen, bezeichnet man die effektive Adresse auch als **Offset** oder **Displacement**, d.h. als Abstand vom Segmentanfang.

Einige Kombinationen von Registern zur Adressbildung sind bei 8086-Prozessoren fest vorgegeben.

Hierzu gehört vor allem das Zahlenpaar CS:IP, welches innerhalb vom Codesegment auf den nächsten auszuführenden Befehl zeigt. Dies wird z.B. bei der „fetch instruction“ (=„befehl holen“) benötigt.

Eine andere feste Zuordnung bildet das Zahlenpaar SS.SP, welches bei Stack-Operationen Verwendung findet. Hier als Beispiel „push“, „call“, „pop“ oder auch „ret“.

Daneben existiert noch die variable Zuordnung, welche der effektiven Adresse (im folgenden mit EA abgekürzt) ein Segmentregister zuordnet:

EA	DS
EA	ES
EA	SS
EA	CS

Die Default-Vorgabe ist hier „DS“, wenn nicht „BP“ in der effektiven Adresse steht. Steht jedoch der Basepointer (BP) in der effektiven Adresse ist die Default-Vorgabe das „SS“-Register.

Beispiele für die Adressierung:

```
mov AX, [BX]      ; Default DS
mov AX, [BP]      ; Default SS
mov AX, [ES:BS]   ; Default ES
```

2.3.2 Adressierung mittels Register:

2.3.2.1 Operand im Register

Anzahl der Operanden	Beispiel	Funktion
1	inc ax neg bl	AX := AX + 1 BL := -BL
2	add ax, bx	AX := AX + BX

Befinden sich die für die Ausführung eines Befehles notwendigen Operanden bereits in den Registern, so erübrigt sich ein relativ langsamer Zugriff auf den Arbeitsspeicher.

Es gibt hier zwei Variationen:

Wenn in einem Befehl zwei Operanden angegeben werden, stellt der linke Operand sowohl Quelle als auch Ziel dar. Der rechte Operand beinhaltet immer eine Quellenangabe.

2.3.2.3 Inhärente Adressierung

Eine weitere Adressierungsart, bei welcher mindestens ein an der Operation beteiligter Operand in einem Register zu finden ist, ist die inhärente Adressierung.

Inhärent bedeutet soviel wie „innewohnend“ oder „anhaftent“.

Für Programmierer bedeutet dies, daß nur der zweite Zahlenwert in Form einer Register- oder Arbeitsspeicherangabe angegeben werden muß.

Beispiel: `mul bl ; AX := AL * BL`
`lahf ; Hole Inhalt der 8 niederwertigsten Bit des PSW nach AH.`

2.3.2.3 Unmittelbare Adressierung

Hierbei handelt es sich um die wohl einfachste Art einen Befehl zu formulieren.

Der zweite für den Befehl notwendige Operand wird direkt in der Befehlszeile angegeben.

Es ist zu beachten, daß der direkt angegebene Wert nur als Quelle, nicht jedoch als Ziel für eine arithmetische Operation angegeben werden darf.

Beispiel: `sub cx, 123 ; CX := CX - 123.`

2.3.3 Adressierung mittels Speicheroperand:

2.3.3.1 Direkte Adressierung

Die direkte Adressierung sollte nicht verwechselt werden mit der unmittelbaren Adressierung. Wird dort ein absoluter Zahlenwert angegeben, welcher in ein Register geladen wird, so wird der Inhalt einer Speicherzelle im Arbeitsspeicher, auf die der zweite Operand zeigt, in ein Register geholt.

Tabelle:

24h
86h
34h
0FH

Beispiel:

Mit dem Befehl `mov ax, Tabelle` soll der Inhalt der Speicherstelle, auf die der Name `Tabelle` zeigt, nach AX geholt werden.

Der logische Name `Tabelle` steht für den Abstand zwischen dem Anfang des Datenssegments und der gesuchten

Datensegment DS	0001 0010 0011 0100 0000	(binär)
Offset auf Tabelle	+ 0000 0000 0101 0001 0010	(binär)
absolute Adresse	0001 0010 1000 0101 0010	(binär)
	1 2 8 5 2	(hexadezimal)
	75858	(dezimal)

Speicherzelle. In Verbindung mit dem Datenssegment wird nun die physikalische Adresse gebildet.

Angenommen in DS befindet sich der Wert 1234h und das Label `Tabelle` zeigt auf eine Speicherzelle, welche 512h Byte vom Anfang des Datenssegments entfernt ist.

Dann ergibt sich für die physikalische Adresse:

An dieser Stelle befindet sich die 16-Bit-Zahl 1234h, welche nun in das Register AX kopiert wird.

2.3.3.2 Indirekt indizierte Adressierung

Ein Beispiel soll den Vorgang dieser Adressierungsart verdeutlichen:

Sucht man in einem Buch nach einem Stichwort, schaut man zunächst in den Index. Dort wird die Seitenzahl gefunden, auf welcher die gesuchte Information zu finden ist.

Ganz analog geschieht dies im PC.

Der Prozessor findet im „Index“-Register einen Offset auf eine Speicherzelle im Datenssegment.

Die effektive Adresse im Arbeitsspeicher wird aus der Segmentadresse, die hier im DS zu finden ist, und dem Offset im SI-Register gebildet.

Beide Informationen zusammen zeigen zusammen auf die Adresse.

Als Indexregister kann das SI-, DI-, BX- oder das BP-Register verwendet werden.

Zusätzlich können diese Register auch noch miteinander addiert werden:

BX	BX + SI	BX + Displacement	BX + SI + Displacement
BP	BX + DI	BP + Displacement	BX + DI + Displacement
SI	BP + SI	SI + Displacement	BP + SI + Displacement
DI	BP + DI	DI + Displacement	BP + DI + Displacement

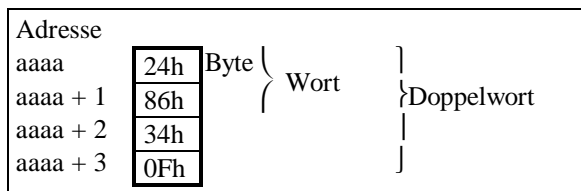
Um diese Adressierungsart abzurunden, lassen die INTEL-Ingenieure auch noch das addieren von absoluten, konstanten Zahlenwerten, den sogenannten *Displacement*, zu, z.B.

```
mov ax, [SI + 1234h]
mov bx, [BP + DI + 9F8Eh].
```

Wenn man diese Art der Adressierung in seinem Programm sinnvoll einsetzt, ist es möglich, sehr flexibel auf Daten im Arbeitsspeicher zuzugreifen, da mit Ausnahme des Displacement alle Adressangaben während des Programmlaufs berechnet werden können.

2.4 INTEL-Konventionen:

Der Arbeitsspeicher eines Computers, welcher mit einem 8086-Prozessor ausgerüstet ist, wird in Bytes verwaltet. Da aber beispielsweise ein 80286-Rechner 16-Bit-Daten, also 2 Byte auf einmal verwalten kann, wurde eine Regelung bezüglich der Reihenfolge der einzelnen Bytes bei der Datenspeicherung notwendig. INTEL legte fest, daß das niederwertigste Byte (Low-Byte, oder kurz Lo-Byte) in der Speicherstelle mit der niedrigeren Adresse geschrieben wird. Dementsprechend sind die höherwertigen Bytes (High-Byte, oder Hi-Byte) an den höheren Adressen zu finden. Dies hat zur Folge, daß die Überprüfung des übersetzten Programmcodes erschwert wird.



Nach dieser Festlegung führt ein Zugriff auf ein Byte an Adresse aaaa (Beispiel-Bild) zu Ergebnis 24h, ein Zugriff auf ein Wort zum Wert 8624h und ein Doppelwortzugriff zum Wert 0F348624h.

Vor Hexadezimalzahlen, welche mit einem Buchstaben anfangen, wird immer eine Null geschrieben. Diese Festlegung ist dadurch begründet, daß der Assembler die Zahl ansonsten als Variable oder Label interpretieren würde. Dies wird durch die führende Null verhindert.

In Befehlen, die zwei Operanden benötigt, gibt der erste das Ziel der Operation an. Der zweite Operand (und eventuell auch der erste, Beispiel Addition) stellt die Quellenangabe dar.

3. Grundelemente der Assemblersprache (TASM = Turbo-Assembler)

3.1 Datentypen in Assembler

Datentyp	ab Prozessor	Größe in Byte
BYTE	8086	1
WORD	8086	2
DWORD	8086	4
FWORD	80386	6
PWORD	80386	6
QWORD	8086	8
TBYTE	80386	10

Auf der Ebene des Prozessors hat ein Programmierer es nur noch mit Bytes, Wörtern oder dem Vielfachen von Wörtern als Datentypen zu tun:

In der Praxis arbeitet man vorrangig mit den ersten drei Formaten: Byte, Word und Double-Word.

Ein ASCII-Zeichen wird Rechnerintern in einem 8-Bit-Format definiert. Demzufolge paßt ein ASCII-Zeichen exakt in ein Byte.

Eine Zeichenkette wiederum besteht aus mehreren ASCII-Zeichen und läßt sich also als Folge einzelner Bytes darstellen.

3.2 Der Aufbau einer Assemblerzeile

Eine Datei, welche von einem Assembler-Compiler übersetzt werden soll, kann mit jedem beliebigen Texteditor erstellt werden.

Wichtig ist nur, daß die Datei keinerlei Steuerzeichen enthalten darf.

Sie besteht aus einer Ansammlung einzelner Anweisungen an den Compiler und mnemonischen Befehlen, die als Programmcode übersetzt werden sollen.

Pro Zeile ist dabei jeweils nur eine Anweisung, bzw. Befehl erlaubt.

Der Aufbau einer solchen Zeile sieht aus wie folgt:

Marke:		mov		ax		,		bx
Namensfeld („label“)	Feld-Trenn- Zeichen	Operationsfeld („Mnemonicfeld“)	Feld-Trenn- Zeichen	Operand 1	optionales Trennzeichen	Trennung von Operanden	optionales Trennzeichen	Operand 2

3.2.1 Das Namensfeld („label“)

Im Namensfeld stehen symbolische Bezeichner (Label), welche der Programmierer festlegt. Mit ihrer Hilfe können Speicheradressen, Variablen, Konstanten, Pseudooperanden oder Sprungziele (Marken) für eine Programmverzweigung angesprochen werden. Ein Label muß eindeutig sein, d.h. ein Label darf nur einmal im Programm definiert werden. Zulässig zur Definition eines Labels sind die Zeichen A - Z a - z _ @ \$? 0 - 9, wobei Ziffern nicht als erstes Zeichen, @ und \$ nur eingeschränkt eingesetzt werden können. Mit der Option **/m1** ist der TASM-Compiler in der Lage, in allen Symbolnamen nach Groß- und Kleinschreibung zu unterscheiden. Für ein Label, mit dem eine Speicheradresse im Datensegment bezeichnet wurde, fügt der Assembler beim Übersetzen die Segmentadresse und den Abstand vom Segmentanfang bis zum bezeichneten Speicherplatz, den **Offset**, in den Programmcode ein. Ein solcher Bezeichner kann im Programm wie eine Variable eingesetzt werden, d.h. man kann den Wert dieser Variablen einem Register zuweisen oder auch den Wert verändern. Wird eine Variable zur Adressierung eines Speicherplatzes eingesetzt, so kann diese Variable bis zu vier Attribute besitzen:

Segment:Offset	Typ	Länge	Wert
Adresse innerhalb des Arbeitsspeichers	Byte, Word, Dword, etc.	Anzahl der definierten Typen	Inhalt bei Programmstart

Eine Variablen-Deklaration kann z.B. wie folgt aussehen:

Winni	dw	1234h
symbolischer Name	Typ: Word (2 Byte)	Startwert bei Programmbeginn

Pseudooperationen sind solche, die eine Wertzuweisung nur für den Compiler beinhalten. An jeder Stelle des Programmes, an der das Label eines solchen Operanden eingesetzt wird, fügt der Compiler den entsprechenden Wert ein, der nicht als Wert im Daten- oder Codesegment wiederzufinden ist.

Wird ein Label für ein Sprungziel verwendet, muß dem Namen ein Doppelpunkt folgen. Trifft der Compiler auf eine solche Anweisung, wird hinter dem Befehlsbyte für den Sprungbefehl der Abstand zu diesem Label eingetragen.

3.2.2 Trennzeichen

Als Trennzeichen zum nächsten Element einer Zeile können ein oder mehrere Leerzeichen oder Tabulatoren verwendet werden. Für den Assembler ist an dieser Stelle die Erkennbarkeit eines neuen Elements in der Zeile wesentlich. Bei mehreren Leerzeichen ist also nur das erste relevant. Leerzeichen bzw. Tabulatoren können dem Programm daher ein lesbares Äußeres geben.

3.2.3 Operationsfeld

Im Operationsfeld stehen die eigentlichen Assembler-Befehle. Hier wird angegeben, welche Operation vom Prozessor durchgeführt werden soll. Die Wirkung des Befehls wird in Zusammenhang mit den Operanden deutlich, welche durch ein Komma voneinander getrennt werden. Es gibt Befehle, die keinen, einen oder zwei Operanden benötigen.

Hier ein paar Beispiele: `clc` ; kein Operand
`pop ax` ; ein Operand
`mov ax, bx` ; zwei Operanden
`mov [bx+di], [ax+si+12]` ; zwei Operanden mit arithmetischen Operationen

3.2.4 Operandenfeld

Im Operandenfeld wird angegeben, woher die Werte kommen, mit denen der Prozessor arbeiten soll.

Die Datenquelle kann entweder ein Register, ein Speicherplatz oder ein absoluter Wert, welcher direkt in der Befehlszeile steht, sein.

Ferner wird hier das Ziel angegeben, an welches der Prozessor das Ergebnis der Operation schreiben soll.

Wenn das Operandenfeld leer ist, handelt es sich in der Regel um einen Befehl, welcher den Prozessor direkt steuert.

Hier eine Liste der Steuerbefehle:

Befehl	Wirkung
CLC	löscht das Carry-Flag
CMC	Komplementiere (invertiere) das Carry-Flag
CLD	löscht das Direction-Flag
CLI	sperrt Hardware-Interrupts
STC	setzt das Carry-Flag
STD	setzt das Direction-Flag
STI	gibt Hardware-Interrupts frei
HLT	Prozessor wartet auf den nächsten Hardware-Interrupt
WAIT	Prozessor wartet auf ein Hardware-Signal am Prozessoreingang TEST

Es existieren aber auch zahlreiche Ein-Byte-Befehle, die zur Korrektur von arithmetischen Operationen oder der Konvertierung von einem Datentyp in den nächsthöheren gebraucht werden, so z. B.:

Befehl	Wirkung
DB	reserviert ein Byte im Datensegment
DW	reserviert ein Word im Datensegment (2 Byte)
DD	reserviert ein Double-Word im Datensegment (4 Byte)
DP	reserviert sechs Byte im Datensegment
DQ	reserviert ein Quad-Word im Datensegment (8 Byte)
DT	reserviert ten Byte im Datensegment
AAA	Korrektur nach BCD-Addition
AAD	Korrektur nach BCD-Division
AAM	Korrektur nach BCD-Multiplikation
AAS	Korrektur nach BCD-Subtraktion
CBW	wandelt ein Byte in ein Word um, indem das Bit 7 von AL in die Bits 0 bis 7 von AH kopiert wird
CWD	wandelt ein Word in ein Double-Word um, indem das Bit 15 von AX in die Bits 0 bis 7 von DH kopiert wird

Befehle mit zwei Operanden dienen dem Datentransport oder der Manipulation von Daten. Für diese Aktionen müssen natürlich Quelle und Ziel angegeben werden. In der Maschinensprache gilt dabei die Festlegung, daß zuerst das Ziel und als zweites die Quelle angegeben wird, so z. B.

`mov ax, bx` ; entspricht $Variable a := Variable b$.

Das Ziel kann aber gleichzeitig auch Datenquelle sein. Als Beispiel diene hier die Addition:

`add ax, bx` ; entspricht: Variable a := Variable a + Variable b.

Einige häufig benutzte Befehle sind:

Befehl	Wirkung
ADD	Addition zweier 8- oder 16-Bit-Operanden
AND	logische UND-Verknüpfung zweier 8- oder 16-Bit-Operanden
MOV	Übertragung von Daten in der Richtung Register → Arbeitsspeicher Arbeitsspeicher → Register absolutes Datum → Register absolutes Datum → Arbeitsspeicher
OUT	gibt einen 8- oder 16-Bit-Operanden auf eine Hardwareadresse aus

3.2.5 Kommentare

Innerhalb einer Assemblerzeile kann durch das Einfügen eines Semikolons (;) der Beginn eines Kommentares markiert werden.

Alles, was rechts von einem Semikolon steht, wird vom Compiler ignoriert. So können ganze Anweisungszeilen durch ein Semikolon als Kommentar deklariert und so von der Compilierung ausgeschlossen werden.

Da der Sinn eines Assemblerbefehls auf den ersten Blick meist nicht erkennbar ist, sind Kommentare wichtige Elemente in jedem Assemblerprogramm; mit ihnen sollte nicht geizt werden.

Wird auch der Umfang des Quellcodes durch die Verwendung von Kommentaren größer, sorgt doch der Compiler dafür, daß das lauffähige Programm nur noch reinen Quellcode enthält.

3.3 Pseudooperationen

3.3.1 ORG-Anweisung

Die ORG-Anweisung legt die Position des nächsten Befehlscodes innerhalb des Codesegmentes fest.

Besonders bei „COM“-Programmen ist das wichtig, da bei diesen der erste ausführbare Befehl an der Adresse 100h beginnen muß.

Beispiel:

`ORG 100h` ; Der nächste Befehl steht an Offset 100h

3.3.2 EQU-Anweisung

Häufig werden in einem Programm numerische Konstanten und konstante Strings benötigt.

Diese können mit `[LABEL] EQU [Ausdruck]` definiert werden.

An jeder Stelle des Programms, an welcher das 'Label' angegeben wird, setzt der Assembler das Ergebnis des 'Ausdrucks' ein. Genausogut - wenn auch nicht ganz so komfortabel - könnte überall der Term 'Ausdruck' von Hand eingesetzt werden. Das Ergebnis wäre das gleiche.

Im Gegensatz zur Definition einzelner Datentypen im Datensegment (z.B. mit „db“) wird an der Stelle, an welcher die EQU-Anweisung steht, kein Maschinencode erzeugt.

Zur Verdeutlichung ein Beispiel. Die Befehlszeilen

`mov ax, 1000` und `Anzahl EQU 1000`
`mov ax, Anzahl`

bewirken das Gleiche, jedoch wird das Programm durch die Verwendung des Labels leichter lesbar und es kann mehrfach auf denselben Wert zurückgegriffen werden.

Solange die einzelnen Operanden bekannt sind, dürfen innerhalb einer EQU-Anweisung auch arithmetische Berechnungen durchgeführt werden. Besonders zu beachten ist hier, daß die richtige Reihenfolge der Deklarationen eingehalten wird. Der Compiler kann nur auf Werte zurückgreifen, die er zum Zeitpunkt der Compilierung zur Verfügung hat. Die Zeilenfolge

`Kilometer EQU Meter + 1000`
`Meter EQU 1`

kann vom Compiler also nicht berechnet werden.

Strings können einen Ausgabertext terminieren oder man durchsucht mit ihrer Hilfe z.B. eine Tabelle nach diesem Zeichen. Die Strings müssen mit einfachen (') oder doppelten (") Hochkommas begrenzt werden.

Da TASM dort, wo der Name einer EQU-Anweisung steht, den Wert dieser Zuweisung einsetzt, folgt daraus, daß der String nur maximal zwei ASCII-Zeichen (= 2*1 Byte) lang sein darf.

Dies entspricht der maximalen Größe eines CPU-Registers bei 80286-Prozessoren.

3.3.3 '='-Anweisung

Die =-Anweisung weist einem Label numerische Werte zu.

Der Befehl hat die Form *[Label] = [Ausdruck]*.

Ein sinnvoller Einsatz für diese Variablenart ergibt sich in Programmen, in welchen eine Variable an unterschiedliche Umgebungsparameter angepaßt werden muß und kein Maschinencode hierfür erzeugt werden soll.

Als Beispiel:

```
Bildschirm = 0B000h ; Anfang des Video-RAM für Monokarten
IF Farbe
    Bildschirm = 0B800h ;Anfang des VIDEO-RAM für Farb-Bildschirmkarten
ENDIF
```

3.3.4 DUP-Anweisung

Durch die Syntax *Anzahl dup [Wert]* wird der Compiler angewiesen, entsprechend der 'Anzahl' und dem Datentyp den Speicherplatz im Datensegment zu reservieren und mit dem 'Wert' zu initialisieren.

Beispiel:

```
Zeile db 80 dup (?) ; 80 Byte reservieren
```

Später im Programm kann dann das erste reservierte Datum unter dem Namen *Zeile* angesprochen werden.

Für einen weiteren Zugriff auf die weiteren Elemente dieses Bereiches muß nur der Abstand zum Label gezählt in Byte, der sogenannte **Offset** Wert hinzuaddiert werden.

Beispiel:

```
mov ah, [Zeile] ; holt das erste Byte nach AH
mov ah, [Zeile + 4] ; holt das fünfte Byte nach AH
```

4. Struktur eines Assemblerprogramms

4.1 Segmentierung

Während der Compiler einer höheren Programmiersprache den Befehlen automatisch ein Codesegment und den Daten ein Datensegment zuweist, sowie ein Stapelsegment anlegt, muß der Assemblerprogrammierer die notwendigen Segmentdeklarationen selber vornehmen.

Für die Kennzeichnung eines Segmentes genügen ein paar Zeilen:

```

; Segmentanfang
<SegName> Segment [Anordnung] [Verbindung] [Verwendung] ['Klasse']
                ASSUME <Segmentregister>:<SegName>
...
<SegName> ENDS ; Segmentende
    
```

Der angegebene Segmentname muß mit der zugehörigen Segment-Anweisung übereinstimmen. Zwischen den Marken für den Beginn und dem Ende eines Segments stehen die Assembler-Befehle. Hier nun mögliche Segment-Anweisungen:

Sement	Erklärung der Parameter		
SegName	Symbolischer Name des Segments		
Anordnung	Festlegung, wo das Segment im Arbeitsspeicher beginnt		
	Kennung	Anfang	Adresse
	BYTE	nächste Adresse	Segment:XXXXh
	WORD	nächste geradzahlige Adresse	Segment:XXXXh
	DWORD	nächste Doppelwordadresse	Segment:XXXXh
	PAGE	nächste 256-Byte-Adresse	Segment:XX00h
PARA	nächste Paragraphenadresse	Segment:0000h	
Verbindung	Anordnung des Segments im Arbeitsspeicher		
	Kennung	Anordnung	
	AT	feste Adresse	
	COMMON	Segmente gleichen Namens überlagern sich	
	MEMORY	Segmente gleichen Namens werden hintereinander angeordnet (Gesamtgröße ≤ 64 KB)	
	PUBLIC	wie MEMORY	
	PRIVATE	Alle Segmente werden separat angelegt, d.h. Label sind nur den einzelnen Segmenten bekannt	
STACK	Segmente gleichen Namens werden zusammengefaßt Doch: Nur für Stapelsegment zulässig !!!		

Alle Parameter der Segment-Anweisung, mit Ausnahme des Namens, sind optional. Erfolgen keine Zuweisungen durch den Programmierer, so gilt die Default-Einstellung:

```
<SegName> Segment PARA PRIVATE
```

Durch die ASSUME-Anweisung wird der Compiler angewiesen, bei allen Befehlen, welche implizit definierte Segmentregister verwenden, die korrekten Segmentreferenzen einzusetzen.

Jedes einzelne Segment eines Programms kann bis zu 64 KB groß werden. Ist der Bedarf für größere Segmente gegeben, müssen den entsprechenden Segmentregistern und dem Compiler neue Referenzen mitgeteilt werden. In der ASSUME-Anweisung können einzelne Zuweisungen durch ein Komma getrennt in eine Zeile geschrieben werden. Es ist auch möglich, einem Register „nichts“ zuzuweisen:

```
ASSUME CS:CSEG, DS:DSEG, SS:SSEG, ES:NOTHING
```

Hierdurch können Segmentreferenzen wieder entfernt werden, wenn ein Register auf kein Segment zeigen soll. Durch ein ASSUME werden die Segmentregister jedoch nicht mit den zugehörigen Adressen versorgt.

Besteht ein Programm aus mehreren gleichartigen Segmenten, muß vor dem Zugriff auf dieses Segment eine ASSUME-Zeile eingefügt werden.

Eine vollständige Segmentzuweisung muß also so aussehen:

```

<SegName> SEGMENT [Anordnung] [Verbindung] [Verwendung] ['Klasse']
                ASSUME <Segmentregister>:<SegName>
                mov ax, <SegName>
                mov <Segmentregister>, ax
...
<SegName> ENDS
    
```

4.1.1 Codesegment

Alle Befehle eines Programms müssen nach der Übersetzung durch den Compiler von der CPU in einem Segment, welches über das CS-Register angesprochen wird, zu finden sein. Dies ist das sogenannte Codesegment.

Die einzelnen Befehle werden durch das Registerpaar CS:IP adressiert.

Innerhalb des Codesegments können zusätzlich zu den Befehlen auch Daten gespeichert werden.

4.1.2 Datensegment

Die gesamten Daten eines Programms (hierzu zählen Konstanten, Variablen, feste Texte, Datenpuffer etc.) werden in dem Datensegment abgelegt, welches über das DS-Register oder das ES-Register adressiert wird.

Werden die Daten im Codesegment abgelegt, beinhaltet das DS-Register den gleichen Wert wie das CS-Register.

Dieses Verfahren wird bei 'COM'-Programmen angewendet.

In 'EXE'-Dateien werden hingegen vom Codesegment getrennte Datensegmente angelegt. Somit hat man für die Daten bis zu 64 KB zur Verfügung.

Innerhalb eines Programmes können mehrere Datensegmente existieren. Ein zweites Datensegment kann z.B. über das ES-Register angesprochen werden, nachdem das erste Datensegment über DS adressiert wurde.

4.1.3 Stapelsegment

Das Stapelsegment ist neben dem Codesegment das einzige, das in jedem Programm enthalten sein muß. Es dient generell zur Zwischenspeicherung von Registerinhalten, die immer als WORD, d.h. mit 2 Byte dort abgelegt werden.

Eine Zwischenspeicherung ist z.B. immer dann notwendig, wenn ein Unterprogramm aufgerufen werden soll.

Die Größe des Stapels wird durch folgende Syntax festgelegt:

```
STACK SEGMENT 'STACK'  
    db <Größe> dup (?)  
STACK ENDS
```

Für 'Größe' muß eine entsprechende Anzahl von Bytes eingetragen werden.

Bei der Erstellung einer 'COM'-Datei kann auf die Deklaration eines Stapelsegments verzichtet werden. Da dieser Dateityp mit nur einem 64 KB-Segment auskommt, werden die letzten 100 Byte des Segments automatisch als Stapel angelegt.

4.1.4 Vereinfachte Segmentzuweisungen (Turbo-Assembler)

Für die Bildung des Codesegments genügt alternativ zum obigen Verfahren die Zeile

```
.CODE
```

für das Datensegment

```
.DATA
```

für das Stapelsegment

```
.STACK [Größe]
```

Der TASM sorgt dann automatisch für die korrekte Segmentbildung einschließlich der ASSUME-Anweisung.

4.1.5 Zugriff auf Segmente

Für den Zugriff auf Befehle und Daten eines Programmes ist die Kenntnis des Zusammenwirkens der CPU-Register notwendig. Für die 8086-Prozessoren gelten hierbei folgende Festlegungen:

Segment	Register
Codesegment	CS:IP
Stapelsegment	SS:SP SS:BP
Datensegment	DS:AX DS:BX DS:CX DS:DX DS:SI DS:DI
Extrasegment	ES:AX ES:BX ES:CX ES:DX ES:SI ES:DI

4.1.5.1 Zugriff auf das Stapelsegment

Der Stapel ist nach dem Schema „Last in - First out“ aufgebaut. In Zusammenarbeit mit dem SS-Register zeigt das SP-Register (Stackpointer) auf die Adresse, an der das letzte auf dem Stapel gespeicherte Wort abgelegt ist. Der Buffer-Pointer (BP) erlaubt durch seine implizite Verknüpfung mit dem SS-Register einen schnellen Zugriff auf Werte innerhalb des Stacksegments. Auf den Stapel werden Registerinhalte in der Regel durch den Befehl ‘PUSH’ abgelegt, bzw. durch ‘POP’ von dort wieder abgeholt. Dabei wird jeweils der Wert des Stapelzeigers aktualisiert.

Dies soll an einem Beispiel verdeutlicht werden:

Durch den Befehl ‘push ax’ wird nun zuerst der Inhalt von SP um zwei vermindert und anschließend der Inhalt von AX auf die nun adressierte Speicherstelle kopiert. Entsprechend der INTEL-Konvention befindet sich dabei das Low-Byte auf der niedrigeren Adresse. Ein anschließender Befehl ‘push cx’ hinterläßt das folgende Bild:

Stapel vor Ausführung von PUSH				Stapel nach dem Ausführen von PUSH AX			
AX	0A01h	→	96h	?	?	97h	
CX	0150h		98h	?	?	99h	
SP	0100h	→	100h	?	?	101h	
AX	0A01h		96h	?	?	97h	
CX	0150h		→98h	01	0A	99h	
SP	0098h	↗	100h	?	?	101h	

Wird jetzt der Befehl ‘pop ax’ verwendet, wird zuerst der Inhalt der Speicherstelle, auf die der SP zeigt, nach AX kopiert, und anschließend der SP um 2 erhöht.

Dies liefert folgendes Bild:

Ein Aufruf von ‘pop cx’ liefert dann eine Vertauschung der Inhalte von AX und CX.

Stapel nach Ausführen von PUSH CX						
AX	0A01h	→	96h	50	01	97h
CX	0150h	↗	98h	01	0A	99h
SP	0096h	↑	100h	?	?	101h

Generell muß in jedem Programm dafür Sorge getragen werden, daß sich für jedes ‘PUSH’ auch ein ‘POP’ im Programm findet.

Werden zuviele oder zuwenige Werte vom Stapel zurückgeholt, ist nicht nur ein Fehlverhalten des Programms die

Stapel nach Ausführen von POP AX						
AX	0150h	→	96h	50	01	97h
CX	0150h	↗	→98h	01	0A	99h
SP	0098h	↗	100h	?	?	101h

Stapel nach Ausführen von POP CX						
AX	0150h	→	96h	50	01	97h
CX	0A01h		98h	01	0A	99h
SP	0100h	→	100h	?	?	101h

Folge, sondern der Rechnerabsturz ist, sozusagen, vorprogrammiert.

Obwohl das SP-Register für arithmetische Operationen zur Verfügung steht, sollte der Registerinhalt nicht verändert werden. Ein Stapelüberlauf und ein damit verbundener Systemabsturz könnte die Folge einer fehlerhaften Stackprogrammierung sein.

Es gilt also:

- Alle Daten, die auf einen Stapel abgelegt werden, müssen von dort auch wieder abgeholt werden.
- Möglichst keine Veränderung des SP-Registers.

4.1.5.2 Zugriff auf das Datensegment

Der Zugriff auf variable Daten eines Programms erfolgt entweder mittels der direkten Adressierung oder durch indirekt indizierte Adressierung des Datensegments.

Beispiele:

Im Datensegment ist die Deklaration *Variable db 0815h* zu finden. Mit der Anweisung *mov ax, Variable*; holt den Inhalt der Speicherstelle nach AX wird die *Variable* direkt adressiert und der Wert 0815h in das AX-Register geschrieben.

Befindet sich beispielsweise der Offset des Datums *Variable* im SI-Register, kann dessen Inhalt auch indirekt indiziert in den Akkumulator kopiert werden: *mov ax, [si]* ;holt den Inhalt der Speicherstelle nach AX, auf die der Inhalt des SI-Registers zeigt.

Bei beiden Adressierungsmethoden wird die effektive Arbeitsspeicheradresse, an der der Wert der Variablen physikalisch gespeichert ist, unter Zuhilfenahme des DS-Registers gebildet.

Zeigt hingegen das ES-Register auf das gewünschte Datensegment, muß dies explizit programmiert werden.

Die Programmzeilen könnten dann vielleicht so aussehen:

```
mov ax, es:Variable
mov ax, [es:si]
```

Durch die Anweisung PTR ist es möglich, Daten in ein Register einzulesen, wenn der Datentyp nicht übereinstimmt. Wichtig ist dies dann, wenn beispielsweise im Datensegment ein Datenwort definiert wurde, hiervon aber nur ein Byte in ein 8-Bit-Register eingelesen werden soll.

Beispiel:

```
.DATA
WortWert      dw 4711h
...
.CODE
...
mov al, BYTE PTR [WortWert] ; Low-Byte nach AL
```

4.2 'COM'- und 'EXE-Programme

Unter dem Betriebssystem CP/M und den zur Verfügung stehenden 8-Bit-Prozessoren war es nur möglich auf 2^{16} Byte = 65535 Byte = 64 Kbyte zuzugreifen. Daher mußte sowohl der Programmcode als auch die Daten innerhalb dieses Platzes untergebracht werden.

Diese ausführbaren Programme hatten das Suffix 'COM'. Einige 'COM'-Programme haben bis auf den heutigen Tag überlebt, so z.B. 'command.com'.

Später waren größere Programme von neuen Betriebssystemen mit leistungsfähigeren Prozessoren zu verwalten. Diese Programme erhielten das Suffix 'EXE'.

4.2.1 Die 'COM'-Struktur

Ein 'COM'-Programm belegt immer ein Segment mit einer Maximalgröße von 64 KB. Innerhalb dieses Bereiches sind Programmcode, Daten und der Stapel untergebracht.

Zwangsläufig werden alle Segmente mit der gleichen Anfangsadresse geladen.

Die gesamte Struktur dieses Programmtyps ist fest definiert:

Segmentanfang	Offset		
	0000h		Program-Segment-Prefix (PSP)
	007Eh		
	0080h	Disk-Transfere-Area	Daten und Programmcode
	00ffh		
	JMP	0100h	Erster Befehl
		FEFEh	
		FEFFh	Stapel
		FFDh	
		FFFEh	
		FFFFh	Rücksprung-Adresse

Beim Aufruf jedes Programmes legt MS-DOS das sogenannte *Program Segment-Prefix* an. In ihm sind alle wichtigen Systemaufrufe und die Aufrufparameter des Programms abgelegt.

Hierdurch sind die ersten 255 Byte des Segments vergeben.

Folglich kann der erste ausführbare Befehl erst an Adresse 0100h stehen. Üblicherweise ist hier ein Sprung zu einer höheren Adresse zu finden, um einen Bereich zur Datenspeicherung zu schaffen.

An der Adresse 0FEFFh beginnt der Stack, für den 255 Byte durch MS-DOS reserviert werden.

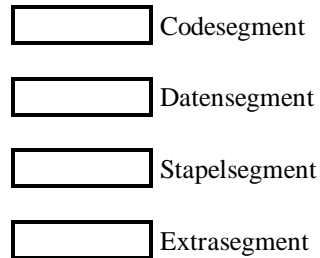
Am Anfang des Stacks (Offset 0FFFEh - 0FFFFh) speichert MS-DOS 0000h als Rücksprungadresse, um das Programmende mittels dem RET-Befehl (return from subroutine) zu ermöglichen.

Durch diesen Rücksprung wird das Programm innerhalb des gleichen Segments am Adreßoffset 0000h fortgesetzt, wo von MS-DOS der Befehl INT 20h abgelegt wurde. Dieser Befehl führt dann zum Programmende.

4.2.2 Die 'EXE-Struktur

Ein 'EXE'-Programm unterscheidet sich von einem 'COM'-Programm bezüglich der Anzahl der möglichen Segmente, des Daten- und des Stapelsegmentes.

Die Grundstruktur eines 'EXE'-Programms sieht wie folgt aus:



Es ist möglich, mit mehreren Code- und mehreren Datensegmenten zu arbeiten. Damit ist die Größe eines 'EXE'-Programmes grundsätzlich nur durch die vom System vorgegebenen physikalischen Grenzen in seiner Größe beschränkt.

Als Mindestvoraussetzung müssen jedoch sowohl das Code- wie auch das Stapelsegment vorhanden sein.

Über die Reihenfolge, wie die Segmente im Arbeitsspeicher angeordnet werden, braucht man sich nicht zu kümmern. Da die einzelnen Segmente an verschiedenen physikalischen Adressen innerhalb des Arbeitsspeichers liegen können, ist es bezüglich der Datensegmente notwendig, die dazugehörigen Segmentregister zu Beginn des Programms mit der richtigen Segmentadresse zu laden.

TASM unterstützt hierbei den Programmierer, sofern er die vereinfachte Segmentanweisung verwendet. Es ist dann möglich, mit einem der vordefinierten Symbole @data, @fardata oder @fardata? dem DS- bzw. dem ES-Register die zugehörige Segmentadresse zuzuweisen:

```

.DATA
; Datendefinition
...
.CODE
mov ax, @data      ; Datensegmentadresse nach AX
mov ds, ax        ; und anschließend nach DS laden
ASSUME ds:@data   ; Compileranweisung
...
  
```

Da die Prozessoren der 8086-Familie keinen Befehl kennen, der es ermöglicht, dem DS-Register eine 16-Bit-Zahl direkt zuzuweisen, wird hier die Segmentadresse über den Umweg des AX-Registers dem DS-Register zugewiesen.

Um die Funktionalität von EXE-Dateien zu erreichen, besitzen EXE-Dateien zusätzlich vor dem Programm-Segment-Prefix einen „Programmkopf“, den sogenannten program header oder auch kurz **header**.

Der Header für EXE-Dateien sieht wie folgt aus:

Offset hex	Inhalt
00 - 01	4DH („M“), 5AH („Z“), Kennzeichnung für gültige EXE-Datei
02 - 03	Byte-Anzahl in letzter durch Programm belegter Speicher-Page (Page = 512 Bytes)
04 - 05	Datei-Größe in Pages inkl. Header
06 - 07	Anzahl der Header-Einträge, die zur endgültigen Speicherbereichs-Anpassung erforderlich sind
08 - 09	Header-Größe in 16-Bytes-Inkrementen; dient der Kennzeichnung des Anfangs des eigentlichen Programms (load modul)
0A - 0B	Minimal-Anzahl benötigter 16-Bytes-Blöcke, die über das Ende des geladenen Programms hinausreichen
0C - 0D	Maximal-Anzahl benötigter 16-Bytes-Blöcke, die über das Ende des geladenen Programms hinausreichen
0E - 0F	Offset des Stacksegments nach dem Laden
10 - 11	Offset des Stackpointers nach dem Laden
12 - 13	negative Word-Prüfsumme aller Wörter in der Datei; Überlauf wird ignoriert
14 - 15	Offset des Instruction-Pointer (IP) nach dem Laden
16 - 17	Offset des Codesegment-Registers nach dem Laden
18 - 19	Offset des ersten Relokalisations-Eintrags in der Datei
1A - 1B	Overlay-Nummer (vom Linker erzeugt; 0 für residenten Programmteil)

5. Grundtechniken der TASM-Programmierung

5.1 Elementare Befehle

5.1.1 Datentransport

Zu den elementaren Befehlen gehören diejenigen, mit denen Daten aus dem Arbeitsspeicher in die CPU und umgekehrt kopiert werden.

5.1.1.1 MOV

Mit MOV werden Daten in Register geschrieben oder von Registern in den Arbeitsspeicher kopiert. Der Zustand der Quelle bleibt hierbei unverändert.

Auch die Statusbits des Prozessor Status Word (PSW) wird durch MOV nicht verändert.

Beispiel:

von	nach	mögliche Register	Beispiel
Register	Register	AX, BX, CX, DX SP, BP, SI, DI	mov ax, cx mov bh, dl
Register	Speicher	AX, BX, CX, DX, SP, BP, SI, DI	mov [1234h], ax mov anna, cl
Speicher	Register	AX, BX, CX, DX SP, BP, SI, DI	mov ah, [47] mov bh, berta
Datum	Register	AX, BX, CX, DX SP, BP, SI, DI	mov cx, 15h mov si, 8150
Datum	Speicher	AX, BX, CX, DX SP, BP, SI, DI	mov anna, Fh mov [bl], 15h
Segmentregister	Register	alle	mov ax, es
Register	Segmentregister	alle	mov ds, bx

Die Segmentregister können nicht direkt durch ein Datum oder indirekt über eine Speicheradresse gesetzt werden. Hier muß immer die Adressierung über eines der anderen Register erfolgen. Eine Änderung des Stack-Pointers (SP) ohne genaue Kenntnisse des Systems ist mit Vorsicht zu genießen, da dies schnell zum Absturz des Rechners führen kann.

5.1.1.2 PUSH und POP

Der Befehl PUSH bietet die Möglichkeit, Registerinhalte auf dem Stapel abzulegen und mit POP diese wieder zurückzuholen.

Zu beachten ist, das nur der Inhalt eines 16-Bit-Registers zwischengespeichert werden kann.

Die Daten, welche als letzte auf dem Stapel abgelegt wurden, werden als erste wieder von dort abgeholt.

5.1.1.3 IN und OUT

Mit IN und OUT werden Daten in einen Ausgabekanal einer zusätzlich in den PC eingebauten PC-Karte gelesen. Zum Beispiel kann die Grafikkarte über definierte Ein- bzw. Ausgabekanäle gesteuert werden.

von	nach	mögliche Register	Beispiel
Peripherie	Register	I/O:Kanal: DX oder direkt Register: AX, AL	in ax, dx in al, 300h
Register	Peripherie	I/O:Kanal: DX oder direkt Register: AX, AL	out dx, al out 3Bh, ax

5.1.2 Arithmetische Operationen

5.1.2.1 Addition/Subtraktion

Da nur 8- oder 16-Bit-Register bei einem AT-kompatiblen Rechner für Operationen zur Verfügung stehen, können die Ergebnisse einer Addition/Subtraktion maximal 16 Bit lang sein.

Das Ergebnis kann entweder vorzeichenlos in einem Bereich von
0 bis 4.294.967.295 (hexadezimal: FFFF FFFF)
oder vorzeichenbehaftet innerhalb des Bereichs von
+2.147.483.647 bis -2.147.483.648

liegen.

Befehl Beschreibung

adc Addition mit Übertragsbit
add Addition
sbb Subtraktion mit Übertragsbit
sub Subtraktion

5.1.2.2 Multiplikation

Bei der Multiplikation können Ergebnisse bis zu 32 Bit lang sein, wenn zwei 16-Bit-Operanden miteinander verknüpft werden. Das Ergebnis steht in dem Registerpaar DX (höherwertige 16 Bit) und AX (niederwertige 16 Bit).

Befehl Beschreibung

imul vorzeichenbehaftete Multiplikation
mul vorzeichenlose Multiplikation von AL oder AX

5.1.2.3 Division

Der Divisionsbefehl läßt einen 32-Bit-Divisor und einen 16-Bit-Dividenden zu. Das Ergebnis setzt sich zusammen aus dem 16-Bit-Quotienten in AX und dem 16-Bit langen ganzzahligen Rest in DX.

Es können nur ganzzahlige Ergebnisse geliefert werden, d.h. Dezimalbrüche können nicht auftreten.

Befehl Beschreibung

idiv vorzeichenbehaftete Division
div vorzeichenlose Division von AL oder AX

5.1.2.4 Inkrementierung und Dekrementierung

Als Inkrementierung bezeichnet man die Erhöhung eines Register- oder Speicherinhaltes um 1, die Dekrementierung verringert den Wert um 1.

Die In- bzw. Dekrementierung arbeitet bis zu 2mal so schnell wie die Addition/Subtraktion.

Befehl Beschreibung

inc um 1 erhöhen
dec um 1 vermindern

5.1.2.5 Negierung

Unter Negierung wird der Vorzeichenwechsel einer Zahl verstanden.

Dabei wird das Zweier-Komplement gebildet (siehe Seite 1/2).

Befehl Beschreibung

neg Zweier-Komplement bilden

5.1.3 Boolesche Operationen

5.1.3.1 AND

Die logische Verknüpfung AND zweier Dualzahlen bewirkt ein Ergebnis nach folgendem Schema:

AND				
A	0	1	0	1
B	0	0	1	1
X	0	0	0	1

5.1.3.2 OR

Das Ergebnis einer OR-Verknüpfung ist dann gleich Eins, wenn mindestens eine der beiden Größen gleich Eins ist:

OR				
A	0	1	0	1
B	0	0	1	1
X	0	1	1	1

5.1.3.3 NOT

Bei der Funktion NOT wird der Wert der einzelnen Bits urngedreht ("Komplementbildung"):

NOT		
A	0	1
X	1	0

5.1.3.4 XOR

XOR setzt im Gegensatz zur OR-Funktion nur dann ein Bit, wenn das betreffende Bit der Vergleichszahlen unterschiedlich ist:

XOR				
A	0	1	0	1
B	0	0	1	1
X	0	1	1	0

5.2 Sprungbefehle

Innerhalb eines Programms besteht nur die Möglichkeit entsprechend dem Zustand der einzelnen Bits (Flags) des Prozessor-Status-Register (PSW) Verzweigungen vorzusehen.

5.2.1 Unbedingter Sprung

Ein Sprung, der ohne jegliche Bedingung durchgeführt werden soll, nennt man unbedingte.

Er hat die Form:

JMP <Label>	Jump
-------------	------

Beispiel:

```
...
    jmp anna      ; Sprung zu Label
    mov ax, 8FFFh
    mov bx, 1
```

...
anna: ...

Hier werden die MOV-Befehle nicht ausgeführt, da unmittelbar vorher die unbedingte Sprunganweisung zum Bezeichner 'anna' steht.

5.2.2 Bedingter Sprung

Bedingte Sprünge werden auf Grund des Zustandes einzelner Bits im Prozessor-Status-Word (PSW) ausgeführt, d. h. sie sind in der Regel durch Flags bestimmt.

Bedingte Sprünge zwingen den Assembler-Programmierer zur Beachtung des Integer- oder Absolut-Wertes, da der Assembler abhängig von der Interpretation des Wertes unterschiedlich arbeitet.

Beispiel:

		absoluter Wert	integer Wert
0000	0001b	+1d	+1d
1111	1111d	+255d	-1d

Will der Programmierer jetzt einen bedingten Sprung durchführen, muß er eigentlich folgende Fälle beachten, um eine Sprunganweisung mit „jump“ (j..., wobei „...“ in der nachfolgenden Tabelle näher erklärt wird) durchführen zu können:

a > b	a < b	a ≥ b	a ≤ b	a = b	a ≠ b	Beschränkung auf 2stellige Dualzahlen																														
ja jnbe	jb jnae	jae jnb	jbe jna	je	jne	a, b werden absolut betrachtet; es gilt: a-b																														
$(CF) \vee (ZF) = 0$ $\overline{CF} \wedge ZF$	$(CF) = 1$ CF	$(CF) = 0$ \overline{CF}	$(CF) \vee (ZF) = 1$ $CF \vee ZF$	$(ZF) = 1$ ZF	$(ZF) = 0$ \overline{ZF}	Zahlenbereich: <table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>b→</td><td>3</td><td>2</td><td>1</td><td>0</td></tr> <tr><td>a↓</td><td>0</td><td>1</td><td>2</td><td>3</td></tr> <tr><td>3</td><td>0</td><td>1</td><td>2</td><td>3</td></tr> <tr><td>2</td><td>-1</td><td>0</td><td>1</td><td>2</td></tr> <tr><td>1</td><td>-2</td><td>-1</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>-3</td><td>-2</td><td>-1</td><td>0</td></tr> </table>	b→	3	2	1	0	a↓	0	1	2	3	3	0	1	2	3	2	-1	0	1	2	1	-2	-1	0	1	0	-3	-2	-1	0
b→	3	2	1	0																																
a↓	0	1	2	3																																
3	0	1	2	3																																
2	-1	0	1	2																																
1	-2	-1	0	1																																
0	-3	-2	-1	0																																
jg jnle	jl jnge	jge jnl	jle jng	je	jne	a, b werden als Integer-Werte betrachtet; es gilt wieder: a - b																														
$[(SF) \oplus (OF)] \vee (ZF) = 0$ $(SF \oplus OF) \vee ZF$	$(SF) \oplus (OF) = 1$ $SF \oplus OF$	$(SF) \oplus (OF) = 0$ $\overline{SF \oplus OF}$	$[(SF) \oplus (OF)] \vee (ZF) = 1$ $(SF \oplus OF) \vee ZF$	$(ZF) = 1$ ZF	$(ZF) = 0$ \overline{ZF}	Zahlenbereich: <table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>b→</td><td>+1</td><td>±0</td><td>-1</td><td>-2</td></tr> <tr><td>a↓</td><td>±0</td><td>±0</td><td>±0</td><td>±0</td></tr> <tr><td>+1</td><td>±0</td><td>+1</td><td>+2</td><td>+3</td></tr> <tr><td>±0</td><td>-1</td><td>±0</td><td>+1</td><td>+2</td></tr> <tr><td>-1</td><td>-2</td><td>-1</td><td>±0</td><td>+1</td></tr> <tr><td>-2</td><td>-3</td><td>-2</td><td>-1</td><td>±0</td></tr> </table> Underflow +3 11 Overflow	b→	+1	±0	-1	-2	a↓	±0	±0	±0	±0	+1	±0	+1	+2	+3	±0	-1	±0	+1	+2	-1	-2	-1	±0	+1	-2	-3	-2	-1	±0
b→	+1	±0	-1	-2																																
a↓	±0	±0	±0	±0																																
+1	±0	+1	+2	+3																																
±0	-1	±0	+1	+2																																
-1	-2	-1	±0	+1																																
-2	-3	-2	-1	±0																																

dabei bedeuten:

bei Absolut-Wert-Betrachtung

- ja = jump above
- jnbe = jump not below or equal
- jb = jump below
- jnae = jump not above or equal
- jae = jump above or equal
- jnb = jump not below or equal
- jbe = jump below or equal
- jna = jump not above
- je = jump equal
- jne = jump not equal

bei Integer-Wert-Betrachtung

- jg = jump greater
- jnle = jump not less or equal
- jl = jump less
- jnge = jump not greater or equal
- jge = jump greater or equal
- jnl = jump not less
- jle = jump less or equal
- jng = jump not greater
- je = jump equal
- jne = jump not equal

 = Zero-Flag gesetzt (= 1)

 = Carry-Flag gesetzt (= 1)

 = Zero-Flag gesetzt (= 1)

 = Sign-Flag gesetzt (= 1)

 = Overflow-Flag gesetzt (= 1)

weiterhin gilt:

- ⊕ = XOR (logische Exklusiv-Oder-Verknüpfung)
- ∨ = Oder-Verknüpfung ∧ = Und-Verknüpfung

und

- (CF) = Carry-Flag (OF) = Overflow-Flag
- (ZF) = Zero-Flag (SF) = Sign-Flag

Auch der Compare-Befehl (cmp), siehe auch 5.2.2.7) verlangt die Aufmerksamkeit des Programmierers hinsichtlich der von ihm verwendeten Werte.

Ein bedingter Sprung ist relativ zum Instruction-Pointer (IP) und short.

Ein Vergleich zwischen Assembler-Befehlssatz und einer höheren Programmiersprache, die beide Gleiches bewirken, verdeutlichen vielleicht das Gesagte:

Assembler	„höhere Sprache“
cmp ... ; setzt Flags durch „Subtraktion ohne Ergebnis“	if „Bedingung erfüllt“
j... marke ; wertet gesetzte Flags aus und führt Sprungbefehl jump aus	then (IP) ← (IP)+D8 // -128d ≤ Marke-(IP) ≤ 127d

Die Sprungbefehle müssen unterscheiden zwischen Absolut- und Integer-Wert:

<u>Absolut-Wert</u>	<u>Integer-Wert</u>
jb = jump below	jl = jump less
ja = jump above	jg = jump greater

Doch jetzt eine frohe Nachricht: **Es geht auch einfacher ...**

5.2.2.1 Carry-Flag

Das Carry-Flag C wird gesetzt, wenn bei einer arithmetischen Operation ein Übertrag aus der höchstwertigen Bitposition heraus erfolgt, so zum Beispiel:

	dual	hexadezimal	dezimal
Summand 1	10100101b	A5h	165
Summand 2	+ 10010001b	+ 91h	+ 145
Übertrag	+1	+1h	+11
Summe	1 00110110b	136h	310

Bei einer 8-Bit-Addition kann es zum Beispiel vorkommen, daß die Summe nicht mehr mit einem Byte beschrieben werden kann. Dann kommt es zu einem Übertrag, welcher in das Carry-Flag geschrieben wird.

Bei vielen MS-DOS-Funktionsaufrufen zeigt das Carry-Flag, daß innerhalb der Funktion bei der Ausführung ein Fehler aufgetreten ist.

Es kann ein bedingter Carry-Flag durchgeführt Der Befehl hierzu lautet:

JC<LABEL>	Jump on Carry
-----------	---------------

Sprung in Abhängigkeit von einem gesetzten werden.

Ein Beispiel soll dies verdeutlichen:

```

mov ax, 8FFFh    ; initialisiere AX
mov bx, 1        ; und BX
anna: add ax, bx  ; AX := AX + BX
jc berta        ; Jump on Carry = bedingter Sprung zu berta
jmp anna        ; unbedingter Sprung zu anna
berta: ...

```

Hier wird eine Addition solange ausgeführt, bis das Carry-Flag gesetzt wird. Ist dies der Fall, wird das Programm ab dem Label 'berta' fortgesetzt.

Es besteht auch die Möglichkeit den Sprung durchzuführen, wenn das Carry-Flag nicht gesetzt ist. Dazu gehört der Befehl:

JNC<LABEL>	Jump on not Carry<
------------	--------------------

5.2.2.2 Overflow-Flag

Das Overflow-Flag zeigt an, daß der Gültigkeitsbereich der Binärzahlen überschritten wurde. Es wird durch eine Exklusive-Oder-Verknüpfung des Übertrags von Bit 6 nach Bit 7, bzw. von Bit 14 nach Bit 15 gebildet.

	dual	hexadezimal	dezimal
Summand 1	0101 1010b	5Ah	90
Summand 2	+ 0111 0110b	+ 76h	+ 118
Übertrag	+ 0 1111 11b	+ 1h	
Summe	1101 0000b	D0h	208

Beispiel:

Übertrag von Bit 6 nach 7 : 1
 Carry-Flag: 0
 EXOR-Verknüpfung: 1

Das Overflow-Flag wird daher auf 1 gesetzt

Da in der Computertechnik in der Regel mit Zweier-Komplementzahlen gerechnet wird, müßte das Ergebnis des Beispiels negativ sein. Durch das gesetzte Overflow-Flag wird aber angezeigt, daß der Gültigkeitsbereich der 2K-Zahlen überschritten wurde und das Ergebnis als positive Zahl zu werten ist.

Die Befehle, um einen bedingten Sprung mittels des Overflow-Flag durchzuführen, sind:

JO<LABEL>	Jump on Overflow
JNO<LABEL>	Jump on not Overflow

5.2.2.3 Sign-Flag

Das Sign-Flag gibt an, welches Vorzeichen das Ergebnis einer Operation hat: ist das Flag gelöscht (0), ist das Ergebnis positiv, andernfalls negativ.

Der Inhalt dieses Flags wird durch eine Kopie des höchstwertigsten Bits des Ergebnisregisters generiert.

Der Sprung-Befehl zur Sign-Flag-Prüfung lautet:

JS<LABEL>	Jump on Sign
JNS<LABEL>	Jump on not Sign

5.2.2.4 Zero-Flag

Ist das Ergebnis einer arithmetischen Operation gleich Null, wird das Zero-Flag auf eins gesetzt.

Der Sprungbefehl zur Überprüfung dieses Ergebnisses lautet:

JZ<LABEL>	Jump on Zero
JE<LABEL>	Jump on Equal Zero
JNZ<LABEL>	Jump on not Zero
JNE<LABEL>	Jump on not Equal Zero

Mit dieser Funktion läßt sich leicht eine Schleife mit abwärts zählendem Schleifenzähler erstellen:

```
    mov ax, 8FFFh ; initialisiere ax
anna: dec ax      ; AX := AX - 1
      jnz anna
      ...
```

5.2.2.5 Parity-Flag

Das Parity-Flag wird auf eins gesetzt, wenn bei einer Operation die Anzahl der auf eins gesetzten Bit des niederwertigsten Byte gerade ist.

Man spricht daher dann auch von der „geraden Parität“.

```
Beispiel 1001 1010 → P = 1
         1101 0110 → P = 0
```

Bei seriellen Datenübertragungen (z.B. RS232) wird die Parität des übertragenen Bytes für die Fehlererkennung ausgenutzt.

Die Befehle zur Überprüfung des Parity-Flag lauten:

JP<LABEL>	Jump on Parity
JPE<LABEL>	Jump on Parity even
JPO<LABEL>	Jump on Parity odd

5.2.2.6 JCXZ und LOP

Diese beiden Sprünge unterscheiden sich von den anderen bedingten Sprüngen, da die Programmierverzweigung nicht in Abhängigkeit vom Zustand des PSW ausgeführt wird, sondern vom Inhalt des CX-Registers abhängt.

Der Befehl JCXZ ist eigentlich selbsterklärend:

JCXZ<LABEL>	Jump if CX-Register Zero
-------------	--------------------------

Der Befehl wird immer dann ausgeführt, wenn das CX-Register gleich Null ist.

Hinter dem Befehl LOOP verbergen sich mehrere Aktionen:

LOOP<LABEL>	Jump if CX-Register not Zero
-------------	------------------------------

- durch diese Anweisung wird der Inhalt des CX-Registers um Eins vermindert
- ist anschließend CX ungleich Null, wird zum angegebenen Label verzweigt, andernfalls wird mit dem nächsten Befehl fortgefahren.

5.2.2.7 Fehlerbehandlung, ein Anwendungsbeispiel: Der CMP-Befehl

Bei vielen Aktionen werden nicht unbedingt die für einen Sprungbefehl gewünschten Flags geändert. Wurde z.B. eine DOS-Funktion fehlerhaft beendet, kann dies mit dem Term

je Fehler ; Carry gesetzt: Weiter mit Fehlerbehandlung

über das gesetzte Carry-Flag abgefragt werden.

Die Information über die Art des aufgetretenen Fehlers steckt im Fehlercode, der im AL-Register zu finden ist. Zu einer dem Fehler entsprechenden Fehlermeldung ist es daher erforderlich, den AL-Inhalt genau zu untersuchen.

Eine Möglichkeit der Analyse bietet eine „Sprungleiste“: der zu überprüfende Register- oder Speicherinhalt wird sukzessive mit den erwarteten Inhalten verglichen.

; Fehlermeldungen

Fehler_0 db 'Diskette/Festplatte ist voll', 7, CR, LF, EDB

Fehler_3 db 'Pfad nicht gefunden', 7, CR, LF, EDB

Fehler_4 db 'Zu viele Dateien offen', 7, CR, LF, EDB

Fehler_5 db 'zugriff verweigert', 7, CR, LF, EDB

Fehler_6 db 'Ungültiges Handle', 7, CR, LF, EDB

Fehler_80 db 'Datei existiert bereits', 7, CR, LF, EDB

Fehler_XX db 'Unbekannter Fehler', 7, CR, LF, EDB

; Fehlerbehandlung

*Fehler: cmp ax, 3 ; Fehler-Nr. 3
jne F_01 ; nein: weiter bei Label*

*mov dx, OFFSET Fehler_3 ; DX := Anfangsadresse Text
jmp SHORT F_AUS ; weiter bei Label*

*F_01: cmp ax, 4 ; Fehler-Nr. 4
jne F_02 ; nein: weiter bei Label*

*mov dx, OFFSET Fehler_4 ; DX := Anfangsadresse Text
jmp SHORT F_AUS ; weiter bei Label*

*F_02: cmp ax, 5 ; Fehler-Nr. 5
jne F_03 ; nein: weiter bei Label*

*mov dx, OFFSET Fehler_5 ; DX := Anfangsadresse Text
jmp SHORT F_AUS ; weiter bei Label*

*F_03: cmp ax, 6 ; Fehler-Nr. 6
jne F_04 ; nein: weiter bei Label*

*mov dx, OFFSET Fehler_6 ; DX := Anfangsadresse Text
jmp SHORT F_AUS ; weiter bei Label*

*F_04: cmp ax, 80 ; Fehler-Nr. 80
jne F_05 ; nein: weiter bei Label*

*mov dx, OFFSET Fehler_80 ; DX := Anfangsadresse Text
jmp SHORT F_AUS ; weiter bei Label*

F_05: mov dx, OFFSET Fehler_XX ; DX := Anfangsadresse Text

*F_AUS: mov ah, BILDAUS ; Rückkehr zu DOS
int DOSINT*

Der Vergleich wird durchgeführt mit der Anweisung:

CMP<WERT	compare
>	

Damit wird der Zustand der folgenden Flags verändert:

Auxiliary	Carry	Overflow	Parity	Sign	Zero
-----------	-------	----------	--------	------	------

Der Prozessor führt den Vergleich durch eine Subtraktion des Vergleichsmusters vom Inhalt des Registers bzw.

Speicherplatzes durch, ohne jedoch den Register- bzw. Speicherinhalt zu verändern.
Mit einem CMP-Befehl kann man im Programm fast überallhin bedingt springen.

5.3 Unterprogramme

5.3.1 Aufruf von Unterprogrammen

Unterprogramme und Funktionen werden mit der Syntax *call <Prozedurname>* aufgerufen. Das Programm verzweigt daraufhin zu diesem Unterprogramm und die Befehle werden abgearbeitet bis zur Rücksprunganweisung *RET*. Anschließend wird das Programm mit dem der CALL-Anweisung folgendem Befehl fortgesetzt.

Der Instruction-Pointer (IP) zeigt vor der Ausführung eines Befehls immer auf den folgenden Befehl. Im Falle eines Unterprogrammaufrufes zeigt das IP-Register auf die Stelle, an der nach der Ausführung des Unterprogramms im Hauptprogramm fortgefahren wird.

Diese Rücksprungadresse wird als erstes auf den Stack gesichert. Bei 'NEAR'-Prozeduren, d.h. Prozeduren im gleichen Codesegment, ist diese Rücksprungadresse zwei Byte lang. 'FAR'-Prozeduren benötigen zwei zusätzliche Byte für die Sicherung der Segmentadresse.

Daraufhin wird das IP-Register mit der Adresse des Unterprogramms geladen. Die nächste *RET*-Anweisung, auf die der Prozessor trifft, bewirkt, daß der Wert, auf den der Stackpointer (SP) zeigt, in das IP-Register übertragen wird. Nun wird das Programm an der Stelle nach der CALL-Anweisung fortgesetzt.

Damit diese Rücksprungadresse richtig vom Stapel gelesen werden kann, gilt besonders für Unterprogramme:

Anzahl der PUSH-Befehle = Anzahl der POP-Befehle
und
Vorsicht mit dem Registerpaar SS:SP

„Unsaubere“ Manipulationen mit dem Stack-Segmentregister (SS) oder am Stack-Pointer (SP) sorgen für einen sicheren „Absturz“ des Rechners.

5.3.1.2 Deklaration eines Unterprogramms

Falls sich ein Unterprogramm nicht im gleichen Codesegment befindet, kann es nur mit einer FAR-Deklaration aufgerufen werden:

PROC <NAME> FAR.

Sinnvollerweise werden beim Aufruf einer FAR-Prozedur die Inhalte des CS- und des IP-Registers auf dem Stapel abgelegt, bevor das Unterprogramm ausgeführt wird.

Durch den *RETF*-Befehl wird dafür gesorgt, daß auch beide Werte wieder vom Stapel zurückgegeben werden.

Um eine FAR-Prozedur un eine NEAR-Prozedur umzuwandeln, muß nur der entsprechende Zusatz der PROC-Zeile geändert werden:

PROC <NAME> NEAR.

Der TASM erzeugt in Abhängigkeit von der Prozedur-Deklaration automatisch die richtige Rücksprungadresse, sobald er auf eine *RET*-Anweisung trifft.

Ist man sich nicht sicher, wo sich das Unterprogramm zum Zeitpunkt des Aufrufs befindet, kann entweder der Zusatz NEAR bzw. FAR ganz weggelassen werden oder der Programmteil wird direkt als FAR deklariert.

Im ersten Fall sorgt der Turbo-Assembler für den richtigen Aufruf entsprechend der Programmgröße, was jedoch bei Verwendung von externen Unterprogrammen mit Vorsicht zu genießen ist.

Die zweite Methode funktioniert hingegen immer, auch wenn durch die zusätzliche Speicherung des CS-Registers Zeit und Speicherplatz verloren geht.

5.3.1.3 Versorgung mit Parametern

Es gibt viele Anwendungsfälle, bei denen einem Unterprogramm verschiedene Parameter übergeben werden müssen. Auf Assemblerebene stehen hierfür drei verschiedene Wege zur Verfügung:

- über den Speicherplatz
- über die Register
- über den Stack (= Stapel).

Die erste Methode erfordert, das im Datensegment genügend Raum für den Datenaustausch reserviert wurde. Dieser Speicherbereich steht dem übrigen Programm dann nicht mehr uneingeschränkt zur Verfügung. Nachteilig ist auch, daß bei einer Änderung der Programmschnittstelle sowohl das Haupt- und Unterprogramm als auch das Datensegment geändert werden müssen.

Der Weg über einzelne Register ist der schnellste Weg, um Unterprogramme mit Daten zu versorgen. Man sollte bei der Auswahl der Register nur sehr sorgfältig arbeiten, da man ansonsten schnell für MS-DOS-Funktionsaufrufe notwendige Register blockiert. Auch können pro Register nur ein oder zwei Byte übergeben werden.

Im Stapel können nahezu beliebig viele Daten abgelegt werden. Zudem ist er immer über SS:SP bzw. SS:BP erreichbar. Zum Zeitpunkt des Prozeduraufrufes sind zudem alle Register frei und können - sofern ihr Inhalt nicht für das Hauptprogramm bewahrt werden muß - beliebig eingesetzt werden. Ein Unterprogramm mit dem Stack als Schnittstelle kann in jedem Programm eingesetzt werden. Dies scheint besonders wichtig bei Programmen, die als 'Include'-Dateien oder als externe Programm-Module eingebunden werden sollen.

Da die Verwendung des Stacks als Schnittstell bei der Einbindung von Assemblermodulen in höhere Programmiersprachen von großer Bedeutung ist, soll hier näher darauf eingegangen werden.

Bevor eine Prozedur programmiert wird, muß festgelegt werden, welche Parameter die Routine verarbeiten soll und in welcher Reihenfolge diese wo zu finden sind.

Ist dies geschehen, kann mit dem Programmieren begonnen werden, in dem vor dem eigentlichen Aufruf des Unterprogramms alle Parameter über ein oder mehrere beliebige Register auf dem Stapel abgelegt werden.

Beispiel:

```

...
; Parameter auf dem Stapel kopieren
mov bx, Para_1
mov cx, Para_2
mov dx, Para_3
push bx
push cx
push dx

call UP          ; Unterprogramm aufrufen

```

weiter: ...

Das Unterprogramm muß nun in der Lage sein, auf diese Parameter zugreifen zu können. Der Turbo-Assembler bietet alternativ zur expliziten Deklaration der einzelnen Parameter durch EQU-Zeilen eine wesentlich einfachere Anweisung:

```
ARG <Parameter:Typ> [, <Parameter:Typ>] [= <Größenvariable>].
```

Der Assembler bildet dabei automatisch die notwendigen Referenzen relativ zum BP-Register, unter der Annahme, daß dieses Register auf dem Stack gesichert wurde.

Somit könnte die Deklaration eines Unterprogramms wie folgt aussehen:

```

UP PROC NEAR
ARG @@PARA_01:WORD, @@PARA_02:WORD, @@PARA_03:WORD = AnzahlParaByte

push bp
mov bp, sp
...
pop bp
ret AnzahlParaByte

```

In der Variablen AnzahlParaByte ist die Gesamtzahl der Bytes der Parameter gespeichert. Die Bedeutung dieser Variablen soll im folgenden Abschnitt besprochen werden.

5.3.1.4 Entsorgung des Unterprogramms

Wird ein Unterprogramm mit Parametern, welche über den Stapel übergeben wurden, wieder verlassen, muß sorgsam darauf geachtet werden, daß diese Parameter vor dem Rücksprung wieder vollständig vom Stapel beseitigt werden. Die Anweisung RET erwartet nämlich die Rücksprungadresse an der Position, auf die der Stack-Pointer verweist. Es erscheint jedoch müßig, die einzelnen Parameter mittels mehrerer POP-Befehle vom Stapel zu entfernen. Als Lösung bietet sich an, den Rücksprungbefehl mit einem Parameter zu versehen:

```
ret <Bytezahl>.
```

Hierdurch werden genau die angegebene Zahl von Bytes auf dem Stapel durch eine Korrektur des Stack-Pointers beseitigt, bevor die Rücksprungadresse ausgelesen wird.

TASM bietet in Verbindung mit der ARG-Anweisung dahingehend einen gewissen Komfort: Die Zahl der Parameter, die mit dem RET-Befehl vom Stapel geholt werden sollen, kann vollständig weggelassen werden. Der entsprechende Wert wird dann automatisch eingefügt.

Im Bereich der Assemblersprache ermöglicht der Einsatz des Befehls ARG die Deklaration von Funktionen auf einfache Weise:

```

ARG <Parameter:Typ> [, <Parameter:Typ>] [= <Größenvariable>] \
...
[RETURNS <Ergebnis:Typ>].

```

Die *Größenvariable* beinhaltet die Anzahl der Parameterbytes und kann der RET-Anweisung hinzugefügt werden.

Durch den Zusatz RETURNS wird festgelegt, welchen Ergebnistyp die Funktion besitzen soll. Bezugnehmend auf das

Label 'Ergebnis' wird das Ergebnis auf dem Stapel abgelegt.

Von dort kann es nach der Rückkehr zum Hauptprogramm durch ein einfaches POP abgeholt werden. Das Zeichen „\“ am Ende der Zeile informiert den TASM, daß der Befehl in der nächsten Zeile fortgesetzt wird.

5.3.1.5 Externe Unterprogramme

Sollen in Programmen immer wieder die gleichen Prozeduren eingebunden werden, empfiehlt es sich, diese getrennt vom Hauptprogramm zu compilieren.

Durch die Zeile *PUBLIC <Name>* werden Variablen oder Unterprogramme „öffentlich“, d.h. externe Programmmodule erhalten die Möglichkeit auf diese Bezeichner zurückzugreifen. Das Hauptprogramm steht dann in einer eigenen Datei und führt die als 'Public' gekennzeichneten Namen nun mittels der Programmzeile

EXTERN <Name>:<FAR/NEAR>

aus.

Der Compiler überprüft daraufhin die deklarierten Namen nicht mehr auf ihr Vorhandensein im Quelltext. Erst der 'Linker' sorgt für den Eintrag der Segmentreferenzen in den ablauffähigen Programmcode.

Hierfür wird der Linker aufgerufen durch

TLINK <Objektdatei> + <Objektdatei>, <EXE/COM-Datei>.

Die erste Objektdatei muß hierbei die des Hauptmoduls sein, erst anschließend folgen ein oder mehrere Module mit einzelnen Unterprogrammen.

Im Laufe der Zeit können sich so eine Reihe von Objektdateien ansammeln, welche den Programmcode für externe Unterprogramme enthalten. TASM bietet mit dem Programm TLIB eine bequeme Verwaltung dieser Bibliotheken an:

TLIB <LibName> Aktion <Objektdatei>	
Aktion	Wirkung
+	Modul hinzufügen
-	Modul entfernen
*	Modul herausziehen
+- oder -+	Modul ersetzen
- oder -	Modul herausziehen und löschen

Soll z.B. die Objektdatei 'ASS_TOOL.OBJ' in die Bibliothek 'TOOLS' aufgenommen werden, erfolgt folgende Eingabe:

TLIB tools + ass_tool.

Sollen Routinen aus dieser Bibliothek von einem Programm aufgerufen werden, müssen die Prozeduren innerhalb des aufrufenden Programmes als extern (*EXTERN <Name>.<FAR/NEAR>*) deklariert werden.

Durch den DOS-Aufruf des Linkers mit der Zeile *TLINK <Objektdatei>, <EXE/COM-Datei>, <Bibliothek>* werden daraufhin die benötigten Module automatisch aus der Bibliothek eingelesen und in den Programmcode eingebunden.

5.3.2 Aufruf von Makros

Im Gegensatz zu Unterprogrammen werden durch den Aufruf von Makros die Makrobefehle während der Übersetzung durch den TASM an der Stelle eingefügt, an welcher der Makroaufruf erfolgt.

Der Vorteil des Makro-Einsatzes gegenüber dem Einsatz von Unterprogrammen liegt in dem Zeitgewinn. Ist ein Makro nur wenige Befehlsbyte lang, kann der Zeitgewinn erheblich sein, da durch das Einsetzen der Befehle, die innerhalb eines Makros programmiert sind, sich die Zugriffe auf den Stapel zum Sichern der Rücksprungadresse oder einzelner Register erübrigen.

Der offensichtliche Nachteil liegt in der Vergrößerung des Programmes um die Befehle des Makros, d.h. das Programm benötigt mehr Speicherplatz.

Doch diesen Nachteil kann man ja sehr leicht abschätzen. Wird ein Makro, welches aus 20 Befehlsbyte besteht, in einem Programm 10mal aufgerufen, so erhöht sich der Speicherbedarf des Programms um $10 \cdot 20 = 200$ Byte.

Ein anderer Nachteil der Makroprogrammierung liegt darin, daß diese nicht in kompilierter Form in ein neues Programm eingebunden werden können. Es muß also immer der Quellcode zur Verfügung stehen, um ihn in andere Programme einbinden zu können.

Da an jeder Stelle, an welcher ein Makro aufgerufen wird, in der Listing-Datei die einzelnen Befehle eingesetzt werden, vergrößert sich dementsprechend das Listing. Diese „Listing-Vergrößerung“ kann jedoch auf zwei Arten unterdrückt werden.

Die erste Möglichkeit besteht durch *%NOMACS*. Hierdurch werden in die Listing-Datei nur noch die Makrozeilen aufgenommen, die Befehle enthalten; alle Kommandozeilen und Compiler-Anweisungen tauchen nicht mehr auf. Wird am Anfang der Makrodefinition *%NOLIST* und am Ende *%LIST* eingegeben, erscheint nichts im Listing, was zwischen diesen beiden Anweisungen steht.

Beispiel für den Aufruf des Makros „DOSRUF“:

```

;*****Makrodefinition*****
; Das Makro „DOSRUF“ dient zum Aufrufen einer DOS-Funktion

MACRO DOSRUF Funktion
    mov ah, Funktion    ; lade nach AH die gewünschte Funktionsnummer
ENDM

;*****Anfang des Codesegmentes*****
    CODESEG            ; Beginn des Codesegmentes
    ORG 100h           ; Anfang für COM-File
Anfang    JM SHORT Start    ; Sprung zu Label Start

;*****Programmbereich*****
Start:    ...
          DOSRUF ENDE
          END ANFANG

```

5.3.3 Einbindung von INCLUDE-Dateien

Man kann den Quelltext von immer wieder verwendeten Unterprogrammen und Makros in Form von Include-Dateien in neue Programme einbinden. Die Anweisung hierfür, die an jeder beliebigen Stelle im Programm stehen kann, lautet: `INCLUDE ^Dateiname^`. An der Stelle im Programm, wo diese Anweisung steht, wird der Quelltext der Include-Datei eingefügt. Damit die Listing-Datei durch das Einbinden von Include-Dateien nicht endlos lang wird, kann mit `%NOINCL` die Ausgabe der Include-Datei im Listing verhindert werden. Beispiel für das Einbinden einer Include-Datei mit Namen „COMPILE.INC“:

```

; Compiler Deklaration einlesen
INCLUDE ^COMPILE.INC^
...

```

5.3.4 Aufruf von MS-DOS-Funktionen

Das Betriebssystem MS-DOS stellt viele Funktionen zur Verfügung, mit denen Dateien bearbeitet oder Zeichenketten auf den Bildschirm ausgegeben werden können. Wird eine Funktion des Betriebssystems aufgerufen, löst die Software eine Interrupt anforderung aus. Diese Interruptanforderung bewirkt, daß das laufende Programm unterbrochen wird und die Befehle innerhalb des Betriebssystems ausgeführt werden. Der am häufigsten verwendete Interrupt ist der **INT 21h**. Generell muß bei einem Aufruf einer Funktion des Interrupt 21h die Funktionsnummer im AH-Register stehen. Die Schnittstelle zum Int 21h zwischen Anwenderprogramm und MS-DOS sieht wie folgt aus:

INT 21h	
Register	Bedeutung
Vor dem Funktions-Aufruf	
AH	Funktionsnummer
AL	Unterfunktion (optional)
CX	Parameter (optional)
DL	Parameter (optional)
DS:DX	Zeiger auf den Speicherbereich (optional)
Nach dem Aufruf: Carry-Flag nicht gesetzt	
diverse (meist AL)	Funktionsergebnisse
Nach dem Aufruf: Carry-Flag gesetzt	
AX	Fehlernummer

Mit diesem Interrupt ist man in der Lage, die fehlerfreie Ausführung eines DOS-Funktion-Aufrufs zu überprüfen, d.h. man ist damit auch in der Lage, auf mögliche Fehler zu reagieren. Seit der DOS-Version 2.0 wird das Carry-Flag zur Kennung des erfolgreichen Funktionsaufrufes benutzt. Üblicherweise ist nach einem fehlerhaften Funktionsaufruf das Carry-Flag gelöscht (0). Sollte es gesetzt sein, wird in AX ein Fehlercode geliefert, den die nachfolgende Tabelle im Klartext erläutert:

Code	Fehler	Code	Fehler
01h	ungültiger Funktionscode	0Bh	ungültiges Format
02h	Datei nicht gefunden	0Ch	ungültiger Zugriffscode
03h	Pfad nicht gefunden	0Dh	ungültige Daten
04h	zu viele Dateien offen	0Eh	reserviert
05h	Zugriff verweigert	0Fh	ungültiges Laufwerk
06h	ungültiges Handle	10h	aktuelles Directory nicht löschar
07h	Speicher-Kontrollblöcke zerstört	11h	Funktion nicht auf das gleiche Laufwerk anwendbar
08h	nicht genügend Speicher frei	12h	keine weiteren Dateien vorhanden
09h	ungültiger Zugriff auf Speicher-Kontrollblock	21h	sperrern bzw. freigeben nicht möglich
0Ah	ungültiger Zeiger auf Enviroment	50h	Datei existiert bereits

Ein Beispiel soll die Vorgehensweise der Einbindung des Int 21h verdeutlichen:

```

;*****Gleichsetzungen für den Compiler*****
...
DOSFUNK equ 21h          ; MS-DOS Interrupt-Nummer
...
;*****Programmbereich*****
...
int DOSFUNK
...

```

5.4 Operatoren

5.4.1 Arithmetische Operatoren

Operator	Funktion
+	Format: Wert1 + Wert2 Addiert Wert1 und Wert2
-	Format: Wert1 - Wert2 Subtrahiert Wert2 von Wert1
*	Format: Wert1 * Wert2 Multipliziert Wert1 und Wert2
/	Format: Wert1 / Wert2 Dividiert Wert1 durch Wert2
MOD	Format: Wert1 MOD Wert2 Dividiert Wert1 durch Wert2 und gibt den Quotienten zurück
SHL	Format: Wert SHL Ausdruck Verschiebt die Bits von Wert um Ausdruck Bitpositionen nach links
SHR	Format: Wert SHR Ausdruck Verschiebt die Bits von Wert um Ausdruck Bitpositionen nach rechts

Die arithmetischen Operationen des Assemblers verknüpfen numerische Operanden und erzeugen ein numerisches Ergebnis. Die am häufigsten verwendeten Operatoren sind die Addition (+), Subtraktion (-), Multiplikation (*) und die Division (/). Zusätzlich gibt es noch den Modulo-Operator (MOD), Shift Left (SHL) und Shift Right (SHR). Die übliche Verwendung der Operatoren sieht wie folgt aus:

- Addition: TAB_PLUS_2 DW TAB+2
Die Variable TAB_PLUS_2 hat hier den Wert der Adresse des zweiten Bytes nach TAB, und *nicht* den Inhalt von TAB plus 2
- Subtraktion DIFF DW TAB1-TAB2
ergibt die Entfernung in Bytes zwischen TAB1 und TAB2
- Multiplikation MIN_PRO_TAG EQU 60*24
hier berechnet Assembler das Ergebnis und weist dieses der Konstanten zu: MIN_PRO_TAG erhält den Wert 1440
- Division PI_QUOT EQU 31416/10000
berechnet den Quotienten, der sich aus der Division ergibt; PI_QUOT wird hier der Wert 3 zugewiesen
- MOD PI_REST EQU 31416 MOD 10000
berechnet den Rest einer Division; PI_REST erhält hier den Wert 1416.

Die Operatoren SHL und SHR verschieben das Bitmuster einer numerischen Operation nach links oder rechts. Diese Möglichkeit wird meistens dann benötigt, wenn Masken definiert werden müssen, durch welche Bitmuster im Speicher verändert werden sollen.

Beispiel:

Es wird mit der Anweisung

Maske EQU 110010b

eine Maske definiert. Nun ergibt die Anweisung

MASKE_LINKS2 EQU MASKE SHL 2

eine neue Konstante mit dem Wert 1100 1000b. Ähnlich ergibt sich aus der Anweisung

MASKE_RECHTS2 EQU MASKE SHR 2

der Wert 1100b.

5.4.2 Logische Operatoren

Operator	Funktion
AND	Format: Wert1 AND Wert2 Verknüpft Wert1 und Wert2 logisch UND
OR	Format: Wert1 OR Wert2 Verknüpft Wert1 und Wert2 logisch Inklusiv-ODER
XOR	Format: Wert1 XOR Wert2 Verknüpft Wert1 und Wert2 logisch Exklusiv-ODER
NOT	Format: NOT Wert Kehrt den Status jedes einzelnen Bits in Wert um, d.h. es wird das Einerkomplement von Wert gebildet

Logische Operatoren werden vorwiegend verwendet um binäre, d.h. nicht dezimale Werte zu manipulieren. Die logischen Operationen erlauben dabei die Manipulation einzelner Bits einer Gruppe.

Die logischen Operatoren AND, OR und XOR verknüpfen zwei Operanden zu einem Ergebnis; NOT erfordert nur einen Operanden.

Der AND-Operator (logisch UND) wird hauptsächlich verwendet, um bestimmte Bits abzudecken, herauszunehmen oder zu filtern. Hierzu wird mit AND überall dort das Ergebnisbit auf 1 gesetzt, wo bei beiden Operanden in dieser Position eine 1 steht. Bei allenm anderen Bitkombinationen in den Operanden wird das Ergebnis auf Null gesetzt.

Der OR-Operator (logisch ODER) setzt jedes Ergebnisbit auf 1, wo sich bei einem dieser Operatoren eine 1 in dieser Bitposition befindet. Bitpositionen, die bei beiden Operatoren auf 0 stehen, werden auch im Ergebnis auf 0 gesetzt.

Der XOR-Operator (logisch EXKLUSIVES ODER) ergibt eine 1, wenn genau ein Operand eine 1 in der Bitposition besitzt.

Operand	Operand	Ergebnis		
1	2	AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	1

Der NOT-Operator invertiert ganz einfach jedes einzelne Bit des Operanden.

NOT 0110 1001b ergibt zum Beispiel *1001 0110b*.

5.4.3 Vergleichoperatoren

Operator	Funktion
EQ	Format: Operand1 EQ Operand2 Wahr, wenn beide Operanden identisch (= equal) sind
NE	Format: Operand1 NE Operand2 Wahr, wenn die beiden Operanden nicht identisch (= not equal) sind
LT	Format: Operand1 LT Operand2 Wahr, wenn Operand 1 kleiner (= less than) als Operand2 ist
GT	Format: Operand1 GT Operand2 Wahr, wenn Operand 1 größer (= greater than) als Operand2 ist
LE	Format: Operand1 LE Operand2 Wahr, wenn Operand 1 kleiner oder gleich (= less or equal) als Operand2 ist
GE	Format: Operand1 GE Operand2 Wahr, wenn Operand 1 größer oder gleich (= greater or equal) als Operand2 ist

Vergleichsoperatoren vergleichen zwei numerische Werte oder Adressen im selben Segment, das Ergebnis ist immer eine von zwei Zahlen: 0, wenn der Vergleich „falsch“ ist oder 0FFFFh, wenn der Vergleich „wahr“ ist.

Da die Vergleichsoperationen nur zwei Ergebnisse produzieren können, werden sie häufig so kombiniert, daß sie einen Ausdruck ergeben, der eine Entscheidung herbeiführt.

Beispiel:

AX soll den Wert '5' bekommen, wenn WAHL kleiner als 20 ist, ansonsten den Wert '6'. Dies kann z.B. so erreicht werden:

MOV AX, ((WAHL LT 20) AND 5) OR ((WAHL GE 20) AND 6).

5.4.4 Wertbestimmende Operatoren

Operator	Funktion
SEG	Funktion: SEG Variable oder SEG Label Ergibt den Segmentwert der Variablen oder des Labels
OFFSET	Funktion: OFFSET Variable oder OFFSET Label Ergibt den Offsetwert der Variablen oder des Labels
TYPE	Format: TYPE Variable oder TYPE LABEL Wenn der Operand eine Variable ist, ergibt TYPE 1 (Byte), 2 (Word) oder 4 (Double-Word); wenn der Operand ein Label ist, ergibt Type -1 (Near) oder -2 (Far)
SIZE	Format: SIZE Variable Ergibt die Anzahl von Bytes, die dieser Variablen zugewiesen sind
LENGHT	Format: LENGHT Variable Ergibt die Anzahl von Einheiten (Bytes oder Worte), die dieser Variablen zugewiesen sind

Diese Gruppe besteht aus passiven Operatoren, die Informationen über Variablen oder Label in einem Programm zurückgeben.

Die Operatoren SEG und OFFSET liefern die Segment - und Offset-Adresse einer Variablen oder eines Labels.

Ein Beispiel

MOV AX, SEG TABELLE

MOV BX, OFFSET TABELLE

Hier werden die Register AX und BX mit der Segmentadresse und dem Offset von Tabelle geladen. Da beide Werte 16-Bit-Werte sind, können sie natürlich auch nur in 16-Bit-werte geladen werden.

Der Operator TYPE übergibt den numerischen Wert, welcher das Typeattribut einer Variablen oder das Entfernungsattribut eines Labels darstellt.

Auf Variablen angewendet ergibt Type eine 1 bei Bytevariablen und eine 2 für Wortvariable. Auf ein Label angewendet liefert Type -1, wenn das Label Near ist und -2 wenn das Label Far ist.

Die Operatoren LENGHT und SIZE (Länge und Größe) sind nur im Zusammenhang mit Variablen sinnvoll, die mit DUP (ein besonderer Operator, mit dem sich wiederholende Werte ausgegeben werden können) definiert wurden. Der erste Operator, LENGHT, ergibt die Anzahl von Einheiten (Bytes oder Words), die eine Variable im Speicher belegt.

Beispiel

TAB DW 100 DUP(1)

MOV CX, LENGHT TAB ; Anzahl Worte von TAB

Diese Anweisungen belegen CX mit 100. Wird LENGHT mit anderen (einfachen) Variablen verwendet, liefert es immer 1 zurück.

Der Operator SIZE übergibt die Anzahl von Bytes in einer Variablen, d.h. SIZE errechnet sich aus LENGHT * TYPE. Auf die Variable 'TAB' aus obigen Beispiel angewendet führt die Anweisung

MOV CX, SIZE TAB ; Anzahl Bytes nach CX

dazu, das CX mit dem Wert 200 geladen wird.

5.4.5 Attributoperatoren

Operator	Funktion
PTR	Format: Typ PTR Ausdruck Übergibt den Typ (Byte oder Word) oder die Entfernung (Near oder Far) eines Speicheroperanden. <i>Typ</i> ist das neue Attribut und <i>Ausdruck</i> ist der Bezeichner, dessen Attribut übergangen werden soll
DS: ES: SS:	Format: Segmentregister:Adressausdruck oder Segmentregister:Label oder Segmentregister:Variable Übergeht das Segmentattribut des Adressausdrucks, des Labels oder der Variablen
SHORT	Format: JMP SHORT LABEL Verändert das Attribut des Sprungziels. Gibt an, daß das Sprungziel +127 bzw. -128 Bytes vom nächsten Befehl entfernt liegt.
THIS	Format THIS Attribut oder THIS Typ Erzeugt einen Speicheroperanden mit dem Entfernungsbitt NEAR oder Far oder dem Typenattribut Byte oder Word an dem Offset, der dem aktuellen Stand des Adresszählers entspricht und das das Segmentattribut des ihn umschließenden Segment hat
HIGH	Format: HIGH Wert oder HIGH Ausdruck Ergibt das höherwertige Byte eines 16-Bit-Wertes oder Adressausdruckes
LOW	Format LOW Wert oder LOW Ausdruck Ergibt das niederwertige Byte eines 16-Bit-Wertes oder Adressausdruckes

Mit den Attributoperatoren kann man einen Operanden ein neues Attribut zuweisen und sein gegenwärtiges Attribut übergeben.

Mit den Zeigeroperator PTR kann man den Typ (Byte oder Word) oder die Entfernung (Near oder Far) eines Operanden übergeben. So kann man mit PTR zum Beispiel auch einzelne Bytes in einer Tabelle aus Worten ansprechen. Wenn die Tabelle so definiert wird

WORD_TAB DW 100 DUP (?)

weist die Anweisung

BYTE_EINS EQU BYTE PTR WORD_TAB

dem ersten Byte der Tabelle aus Worten einen Namen zu. Danach kann jedes andere Byte so einfach definiert werden, wie z.B. *BYTE_FUENF EQU BYTE_EINS+4.*

Mit PTR kann auch das Entfernungsattribut eines Labels verändert werden. Wenn in einem Programm z.B. die Anweisung *START: MOV CX, 100*

steht, erhält 'START' durch den Doppelpunkt das Attribut Near. Dadurch kann 'START' aus dem gleichen Segment

mit *JMP START* angesprungen werden. Um von einem anderen Segment aus nach 'START' zu springen, muß 'START' mit dem Far-Attribut undefiniert werden:

```
FAR_START EQU FAR PTR START
JMP FAR_START.
```

Auch die Segmentattribute eines Labels, einer Variablen oder eines Adressausdrucks können mit speziellen Operatoren (DS:, ES:, SS:) verändert werden.

Dies kann z.B. beim Umgang mit dem 8088 recht nützlich sein. Der 8088 nimmt normalerweise an, daß SS das gültige Segmentregister ist, wenn der Offset mit SP oder BP angegeben wird. Genauso wird DS als Segmentregister verwendet, wenn als Offsetoperand BX oder SI verwendet wird. Mit den genannten Operatoren kann ein anderes Segmentregister angegeben werden. Bei dem Befehl

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

verwendet der 8088 ES und nicht SS um die Speicheradresse zu berechnen.

Der Operator SHORT (= kurz) sagt dem Assembler, das ein Sprungziel in einer Entfernung von +127 oder -128 Bytes vom nächsten Befehl aus gerechnet liegt. Mit dieser Information generiert der Assembler einen JMP als 2-Byte-Befehl und nicht als 3-Byte-Befehl wie sonst. Dies spart logischerweise Speicherplatz.

Beispiel:

```
JMP SHORT DORT
```

...

...

```
DORT: ...
```

Der Operator THIS (= dies) erzeugt einen Adressoperanden vom angegebenen Typ (Byte oder Word) oder der angegebenen Entfernung (Far oder Near), und weist diesem Operanden dieselben Segment- und Offsetattribute zu, wie der darauf folgenden Adresse.

Die Anweisungen

```
BYTE_EINS EQU THIS BYTE
WORT_TAB DW 100 DUP(?)
```

erzeugen die Variable 'BYTE_EINS' mit dem Typenattribut BYTE, aber denselben Segment- und Offsetadressen wie 'WORT_TAB'.

Denselben Effekt wie die obige Anweisung liefert

```
BYTE_EINS EQU BYTE PTR WORD_TAB.
```

THIS kann aber auch benutzt werden, um Labeln das Attribut FAR zu geben, zum Beispiel:

```
START EQU THIS FAR
MOV AX, 100.
```

Hier wird diesem Label das Attribut Far zugewiesen, sodaß dieses auch aus anderen Segmenten heraus angesprungen werden kann.

Die Operatoren HIGH (= hoch) und LOW (= niedrig) haben als Operatoren einen 16-Bit-Wert und liefern das höherwertige bzw. niederwertige Byte dieses Arguments.

Nach der Konstantendefinition

```
KONST EQU 0ABCDh
```

läßt die Anweisung

```
MOV AH, HIGH KONST
```

den Wert 0abh in das AH-Register.

5.5 Dateitypen

Nach dem Editieren, Assemblieren und Linken sind einige Dateien vorhanden:

*.BAK	Sicherungsdatei des Editors
*.ASM	Quelldatei des Editord

*.OBJ	Objektdatei des Assemblers (Zwischencode)
*.LST	Listingdatei des Assemblers

*.EXE	Objektdatei des Linkers (ausführbare Datei)
*.MAP	Mappingdatei des Linkers

5.6 Programme aus mehreren Modulen

Die modulare Programmierung wird durch die Anweisungen GLOBAL, PUBLIC und EXTERN unterstützt.

5.6.1 Anweisung GLOBAL

Die Anweisung GLOBAL teilt dem Assembler mit, daß die mit GLOBAL definierten Symbole in einem anderen Modul benutzt werden sollen, bzw. dort definiert wurden; GLOBAL ist also praktisch eine Kombination aus PUBLIC und EXTRN.

Es gilt:

- die Typen der Symbole müssen in beiden Modulen übereinstimmen;
- der Typ wird im Format *GLOBAL Symbol:Typ* angegeben.

Für Typ gültige Formate:

ABS	eine absolute Konstante
BYTE	Byte
DATAPTR	vom Speichermodell abhängig: NEAR oder FAR Datenzeiger
DWORD	Doppelwort
FAR	FAR-Label CS:IP
FWORD	6 Byte
NEAR	NEAR-Label in IP geladen
PROC	Prozedur-Label NEAR oder FAR
QWORD	8 Bytes
TBYTE	10 Bytes
UNKNOWN	unbekannt
WORD	2 Bytes

Beispiel für die Anweisung *GLOBAL Symbol:Typ*:

```
; Modul Video.ASM
```

```
; in Programm Video eingebunden
```

```
.MODEL SMALL
```

```
GLOBAL VideoRam:ABS ; Symbole im Hauptmodul definiert
```

```
GLOBAL ZeichenZahl:ABS
```

```
GLOBAL Farbe:BYTE
```

```
.CODE
```

```
GLOBAL Ausgabe:PROC ; Prozedur wird in anderem Modul aufgerufen
```

```
Ausgabe PROC
```

```
mov cx, ZeichenZahl ; an anderer Stelle definiert
```

```
cld
```

```
mov ah, Farbe
```

```
mov al, 'A'
```

```
rep stow
```

```
ret
```

```
Ausgabe ENDP
```

```
END
```

Vorgehen bei der Programmentwicklung mit Modulen:

1. getrenntes Editieren der Module
2. die Module werden separat mittels TASM (Turbo-Assembler) assembliert
3. mit TLINK (Turbo-Link) werden die einzelnen Objektdateien zu einem Programm verbunden

5.6.2 PUBLIC

PUBLIC steht in dem Modul, in welchem die Symbole definiert werden. Es erlaubt externen Zugriff auf dieses Modul und die darin enthaltenen Daten, wenn in den externen Modulen die Import Schnittstellen korrekt initialisiert sind. Die 'PUBLIC'-Definition beschreibt die sogenannte Export-Schnittstelle des Moduls, in der alle Namen aufgeführt sind, die irgendein anderes Modul verwenden darf.

Die Syntax lautet:

PUBLIC Name(,Name).

Folgende Namen können als PUBLIC definiert werden:

- Variablen-Namen
- Unterprogramm-Namen
- Marken im Programm
- konstante Zahlenwerte, die mit equ definiert wurden

5.6.3 EXTRN

EXTRN wird in dem Modul definiert, in welchem EXTRN benutzt werden soll. Es bindet externe Moduln in das Programm ein. Wird eine 'EXTRN'-Anweisung innerhalb eines Segments bzw. hinter .CODE oder .DATA angegeben, dann behandelt TASM die zugehörigen Namen so, als seien die externen Bezeichner in diesem Segment definiert.

Wird eine 'EXTRN'-Anweisung außerhalb eines Segments angegeben, dann geht TASM davon aus, daß die angegebenen Variablen über das DS-Register erreicht wird.

Die 'EXTRN'_Definition beschreiben die sogenannte Import-Schnittstelle des Moduls. Sie enthält alle Namen, die in diesem Modul von außerhalb importiert werden müssen und in dem Modul nur verwendet werden.

Die Syntax lautet:

EXTRN Definition (, Definition),

mit Definition:

Name:Typ[:Anz].

Der Name ist ein Symbol, das in einem anderen Modul definiert sein muß.

Der Typ legt den Datentyp des Symbols fest. Folgende Datentypen sind möglich:

- PROC, NEAR, FAR für Unterprogramme und Marken; PROC ergibt abhängig vom verwendeten Speichermodell entweder NEAR oder FAR
- BYTE, WORD, DWORD, FWORD, QWORD, TBYTE für Variable
- ABS für Absolutkonstante, die mit equ deklariert sind.

Anzahl beschreibt, wieviele Datenobjekte des angegebenen Typs das Symbol belegt (optionale Angabe).

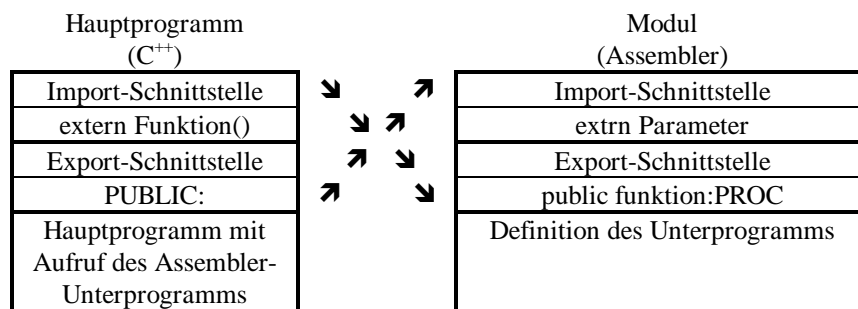
6. Die Schnittstelle zu Borland C⁺⁺

Die interessanteste Anwendung der Assembler-Programmierung ist die Einbindung von Assembler Modulen in Programm-Pakete einer höheren Programmiersprache. Denn werden besondere Anforderungen an die Ausführungsgeschwindigkeit von Programmteilen gestellt oder eine Hardware-nahe Programmierung benötigt, so können die Vorteile des Assemblers genutzt werden.

Eine Basis für solch einen gemischt-sprachigen Programmierstil bildet das Modulkonzept. Voraussetzung hierfür ist, daß der Compiler für die höhere Programmiersprache

- a) das Modulkonzept unterstützt,
- b) die einzelnen Module in Objekt-Dateien übersetzt.

C und C⁺⁺ sowie alle ihre Compiler erfüllen diese Voraussetzung. Für den Daten-Austausch benötigt man zudem Schnittstellen:



6.1 Die Speichermodelle

Der Borland C⁺⁺-Compiler ist in der Lage, Daten und Code auf verschiedene Arten im ausführbaren Programm anzuordnen. Dabei verwendet er eines von sechs Speichermodellen:

- TINY: Daten und Code zusammen maximal 64 Kbyte; Daten NEAR, Code NEAR;
- SMALL: Daten und Code jeweils maximal 64 Kbyte; Daten NEAR, Code NEAR;
- COMPACT: Daten mehr als 64 KByte, Code maximal 64 Kbyte; Daten NEAR, Code NEAR;
- MEDIUM: Daten maximal 64 Kbyte, Code mehr als 64 KByte; Daten NEAR, Code FAR;
- LARGE: Daten und Code jeweils mehr als 64 Kbyte; Daten FAR, Code FAR;
- HUGE: Daten und Code jeweils mehr als 64 Kbyte; Daten FAR, Code FAR.

Die Auswahl des Speichermodells erfolgt über eine Option des Compilers. Danach liegen sämtliche Parameter fest, die man bei der Segment-Definition wählen kann. Um einem zum C⁺⁺ passenden Turbo Assembler-Modul zu schreiben, verwendet man am besten die vereinfachten Segment-Anweisungen. Dabei muß in der .MODEL-Direktive des Assembler-Moduls dasselbe Speichermodell gewählt werden, wie im Hochsprachen-Programm.

Da in C⁺⁺ zwischen Groß- und Kleinschreibung unterschieden wird, müssen im Assembler die Segment- und Klassen-Namen groß geschrieben werden.

6.2 C⁺⁺ ruft ein Assembler-Unterprogramm auf

Soll ein Assembler-Unterprogramm von C⁺⁺ aus aufgerufen werden, muß es im Assembler-Modul in der Export-Schnittstelle aufgeführt sein. Der Bezeichner des Unterprogramms muß in das C⁺⁺-Programm importiert werden. Dazu schreibt man vor der Vereinbarung des Bezeichners das Schlüsselwort 'extern'. Da hier eine Funktion importiert (= Assembler-Unterprogramm) werden soll, entfällt natürlich der Rumpf der Funktion; man gibt in der extern-Definition nur den vollständigen Funktionskopf an.

Im Assembler-Modul muß die Namenskonvention für die Funktion berücksichtigt werden, die der C⁺⁺-Compiler verwendet. Der C⁺⁺-Compiler orientiert sich standardmäßig an der Datei-Endung, wie das Quellprogramm zu

übersetzen ist: Programme mit der Endung .C werden als C-Programme, solche mit der Endung .CPP als C++-Programme übersetzt.

Die Namenskonvention für C ist denkbar einfach: Funktionsnamen erhalten am Anfang einen Unterstrich.

In C++ wurde für Funktionen ein strenges Typkonzept eingeführt, bei welchem die Anzahl und die Datentypen der Parameter beim Funktionsaufruf genau mit den entsprechenden Angaben der Funktionsdefinition verglichen werden. Dieses Typkonzept wirkt sich auf die interne Namenskonvention der Funktionen aus. Am einfachsten erhält man die internen Namen, wenn die Funktionen mit leerem Funktionsrumpf vom C++-Compiler mit der Option -S übersetzt werden. Dies erzeugt eine Assembler-Datei, in der die internen Funktionsnamen vorkommen.

Der Borland C++-Compiler (ab Version 3.0) generiert für Funktionsnamen in C++-Programmen folgende interne Namen:

@Funktionsname\$qParameterbeschreibung[S].

Das optionale 'S' wird bei statischen Funktionen verwendet. In der Parameterbeschreibung wird für jeden Parameter eine Abkürzung angegeben, eventuell gefolgt von einer Zusatzinformation. Hier eine Tabelle mit den Standard-

Parameter-Datentyp	
v	void
zc	chsr
i	int
s	short int
l	long int
f	float
d	double
e	ellipse

Zusatzinformation	
w	const
x	volatile
p	near pointer
n	far pointer
r	near reference
m	far reference

Datentypen:

Beispiel: Für die Funktion `int SwapMax`

*int (*a, int& max, int i, int j)*

ergibt sich folgender interner Name

@SwapMax\$qpiriii.

Will man auf den typsicheren internen Namen verzichten, kann die Funktion im C++-Programm folgendermaßen vereinbart werden:

*extern ²C² int SwapMax(*a, int& max, int i, int j)*

Jetzt wird intern der einfache Name `_SwapMax` gemäß der C-Konvention verwendet. Die Funktion `SwapMax` kann jetzt allerdings nicht mehr überladen werden !

6.2.1 Parameter-Übergabe und Ergebnis-Wert

In C++ werden die Parameter auf dem Stack übergeben. Die aktuellen Parameter werden dabei in umgekehrter Reihenfolge auf dem Stack gespeichert, wie sie in der Aufrufzeile stehen, also von rechts nach links. Dabei wird auch für einen Referenz-Parameter eine Adresse übergeben.

Im Rumpf des Unterprogramms wird auf die Parameter über das Register BP zugegriffen. Dazu wird mit den beiden Befehlen

push BP
mov BP, SP

der alte Inhalt von BP gesichert und BP auf den richtigen Wert gesetzt. Für die Rückkehr aus dem Unterprogramm wird mit

pop BP

der alte Wert in bekannter Weise wieder hergestellt.

In C++ werden die aktuellen Parameter hinter der Aufrufstelle vom Stack entfernt, d.h. im Assembler-Modul muß man sich darum nicht kümmern. Es kann mit 'ret' einfach das Unterprogramm verlassen werden.

Die folgende Tabelle zeigt, wie sich die Datentypen von C++ und Assembler entsprechen:

Datentyp in C++	Datentyp in Assembler
unsigned char, char	byte
enum	word
unsigned short, short	word
unsigned int, int	word
unsigned long, long	dword
float	dword
near *	word
far *	dword

Zu beachten ist: Da auf dem Stack jeweils 16-Bit-Werte geschrieben werden, belegt ein formaler Parameter vom Typ 'char' auf dem Stack ein 16-Bit-Wort; als gewöhnliche Variable belegt 'char' hingegen nur ein Byte.

Aus dem Unterprogramm werden die Ergebnis-Werte über Register an die Aufruf-Stelle zurückgegeben, wobei je je nach Datentyp ein oder zwei Register verwendet werden:

Datentyp des Ergebnis-Wertes	Ergebnis-Wert im Register
unsigned char, char	AX
enum	AX
unsigned short, short	AX
unsigned int, int	AX
unsigned long, long	DX:AX
near *	AX
far *	DX:AX

Die Verwaltung und Benennung der formalen Parameter kann man auch von der Direktive 'arg' erledigen lassen, jedoch kann der Ergebnis-Wert nicht mit Hilfe der Komponente 'returns' der 'arg'-Direktive zurückgegeben werden. Die einzige Wirkung von 'returns' besteht darin, die angegebenen Parameter nicht vom Stack zu entfernen. Bei C++ dürfen aber überhaupt keine Parameter vom Stack entfernt werden. Außerdem werden die Ergebnisse von Funktionen, die von C++ aufgerufen wurden, immer in Registern zurückgegeben.

Beim Schreiben von Unterprogrammen fallen einige Routine-Tätigkeiten auf die Dauer als lästig auf:

- die Befehle des Standard-Vorspanns und Standard-Nachspanns werden in jedem Unterprogramm benötigt, das mit Parametern arbeitet
- der Speicherplatz von lokalen Variablen muß auf dem Stack reserviert werden; vor Rückkehr aus dem Unterprogramm muß der Stack-Pointer SP wieder entsprechend zurückgesetzt werden.

Diese Routine-Aufgaben erledigt TASM, wenn folgende Direktive verwendet wird:

```
.MODEL      SMALL, C      ; für C-Programme
.MODEL      SMALL, CPP    ; für C++-Programme.
```

6.2.2 Retten von Registern

Im Assembler-Unterprogramm werden in der Regel einige Register verwendet. Wird ein Assembler-Programm von C++ aus aufgerufen, stellt sich die Frage nach den Registerinhalten für das C++-Programm. Es gilt, einige Regeln zu beachten.

- Die Segment-Register CS, DS und SS dürfen durch den Unterprogramm-Aufruf nicht verändert werden. Muß eines von diesen Registern innerhalb des Unterprogramms geändert werden (z.B. Stringbefehle), so ist der Inhalt des Registers vorher zwischenspeichern und vor dem Rücksprung wiederhergestellt werden.
- Stack-Pointer SP und das Register BP müssen beim Verlassen des Unterprogramms denselben Wert haben wie beim Aufruf.
- C++ verwendet normalerweise die Register SI und DI für Zählvariablen und als Indizes von Vektoren. Deshalb sollten diese Register gesichert werden, falls sie im Unterprogramm verwendet werden sollen.

Die anderen Register AX, BX, CX, DX und ES dürfen im Unterprogramm frei verwendet werden. Das Status-Register wird ohnehin durch fast jeden Befehl modifiziert.

6.3 Assembler ruft C⁺⁺-Funktion auf

In diesem Fall wird die Funktion aus dem C⁺⁺-Programm exportiert und im Assembler-Modul importiert. Für letzteres wird im Assembler Quellcode einfach folgendes an den Anfang des Moduls eingefügt:

```
extrn @Funktionsname$Parameterbeschreibung[S]:PROC ; für C++  
extrn _Funktionsname:Proc ; für C.
```

In C/C⁺⁺ können alle Bezeichner, die nicht innerhalb einer Funktion deklariert sind, in ein anderes Modul exportiert werden, wobei der C⁺⁺-Compiler diese Bezeichner als „public“ betrachtet.

Im Turbo-Assembler muß der Funktionsaufruf so programmiert werden, wie ihn der C⁺⁺-Compiler generiert hätte:

- Die aktuellen Parameter der Funktion werden im Stack gespeichert, und zwar nicht in der angegebenen Reihenfolge, sondern von rechts nach links.
- Der Funktionsaufruf erfolgt über den 'CALL'-Befehl.
- Nach Rückkehr aus der Funktion werden die Parameter vom Stack entfernt.
- Der Ergebnis-Rückgabe-Wert der Funktion steht im Register AX zur Verfügung; das Assembler-Programm kann ihn verwenden.

6.4 Verwendung gemeinsamer Daten

Daten können sowohl im C⁺⁺-Programm als auch im Assembler-Modul deklariert werden. Nur an der Deklarationsschnittstelle wird ihnen Speicherplatz zugewiesen. In die Moduln, in denen diese Daten nur verwendet werden, müssen sie importiert werden. In C⁺⁺ werden externe Daten wieder einfach durch Voranstellen des Schlüsselwortes 'extern' bei der Deklaration gekennzeichnet. Dagegen können alle Daten, die außerhalb von Funktionen (einschließlich „main“) deklariert sind, ohne besondere Kennzeichnung exportiert werden.

Im Assembler-Modul werden Export-Daten mit der Direktive 'PUBLIC' gekennzeichnet, während die importierten Größen hinter der Direktive 'EXTRN' anzugeben sind.

Im Zusammenhang mit C⁺⁺ spielen die verschiedenen Segment-Anweisungen zur Daten-Definition eine wichtige Rolle:

- Initialisierte Daten werden hinter der Direktive .DATA vereinbart; sie liegen damit im Segment mit dem Namen _DATA.
- Nicht-initialisierte Daten werden hinter der Direktive .DATA? vereinbart und liegen damit im Segment _BBS.

7. Der numerische Koprozessor 8087

Der Mikroprozessor 8088 (und auch der 8086) hat den Nachteil, daß die Rechenleistung recht dürftig ist. Aus diesem Grund wurde parallel zum Mikroprozessor ein Koprozessor entwickelt, welcher durch einen festen Satz von Befehlen zum Abwickeln von rechenintensiven Aufgaben wesentlich besser geeignet ist:

Befehl	Ausführungszeit in μs	
	8088 + 8087	8088 + numerische Software
Addition, Subtraktion	17	1600
einfache Multiplikation	19	1600
erweiterte Multiplikation	27	2100
Division	39	3200
Vergleichsoperation	9	1300
erweitertes Laden	10	1700
erweitertes Speichern	21	1200
Wurzel-Funktion	36	19600
Tangens-Funktion	90	13000
Exponential-Funktion	100	17100

Der numerische Koprozessor 8087 ist ein Chip, welcher komplexe mathematische Berechnungen durchführen kann. Er ist so aufgebaut, daß er zusammen mit einem 8086- oder 8088-Mikroprozessor verwendet werden kann und dessen Befehlssatz und arithmetische Fähigkeiten erweitert. Der IBM Personal Computer bzw. IBM-kompatible PC's haben einen speziell für den 8087 vorgesehenen Stecksockel.

Der 8088 kann selbst arithmetische Berechnungen durchführen, doch sein Befehlssatz kann nur fünfstellige Ganzzahlen (das entspricht zwei Byte) verarbeiten und verfügt auch nur über die vier Grundrechenarten. Der 8087 kann im Gegensatz dazu eine Vielzahl arithmetischer und transzendentaler Operationen (logarithmische als auch trigonometrische Operationen) mit Ganzzahlen und Fließkommazahlen bis zu 18 Stellen (10 Bytes) ausführen.

Der 8087 wird als Koprozessor bezeichnet, weil er Programme in Zusammenarbeit mit dem Hauptprozessor (8088 oder 8086) ausführt. Beim Ablauf eines Programmes, welches darauf eingestellt ist den 8087 zu verwenden, führt der 8086 bzw. 8088 die Befehle aus, die er erkennt und der 8087 die, die er erkennt.

Programme in Hochsprachen verwenden in der Regel den 8087 automatisch, wenn er vorhanden ist.

7.1 Interne Register

Der 8087 verfügt über acht 80-Bit-Datenregister, ein Statuswort und ein Steuerwort, beide 16 Bit groß. Das Statuswort gleicht dem Kennzeichenregister des 8087. Das Steuerwort bestimmt, wie der 8087 Rundungen, Unendlichkeit und Genauigkeit handhabt. Mit seinen Genauigkeitsbits kann eingestellt werden, ob Ergebnisse eine Genauigkeit von 64, 53 oder 24 Bit haben sollen; 64 Bit ist die Standardeinstellung, die anderen sind vorhanden, um Ergebnisse erzeugen zu können, welche mit der älteren Fließkommasoftware kompatibel ist.

7.1.1 Der Stapel des 8087

Die Datenregister des 8087 funktionieren in der gleichen Weise wie der Stack: das zuletzt darauf abgelegte muß als erstes verarbeitet werden.

Bei der „Ablage“ heißt dies, Zahlen werden immer im obersten Register übergeben und der Rest der Register wird um eine Position nach unten verschoben.

Über den Daten-Puffer erhält der Koprozessor seine Daten und Befehle. Der Datenbus hat ein 8-Bit-Format und diese Daten werden in den Registerstapel geschrieben, welcher ein 80-Bit-Format hat. Die Steuerung für den Datentransfer zwischen 8088/8086 und dem 8087 erfolgt durch Adreß- und Statusleitungen.

Da die Datenregister wie ein Stapel arbeiten, verwenden die meisten 8087-Befehle implizit den Stapelinhalt. Wird z.B. eine Addition zweier Zahlen durchgeführt, addiert er den Inhalt von dieser zwei Zahlen auf den Stapel und speichert das Ergebnis auch wieder auf dem Stapel.

7.1.2 Fließkommaformate

Die Datenregister enthalten Zahlen im Fließkommaformat. Diese werden dargestellt, indem diese Zahlen in drei Felder aufgeteilt werden:

- 1) ein Vorzeichenbit
- 2) ein 15-Bit-Exponent
- 3) ein 64-Bit-Signifikant (auch Mantisse genannt).

Glücklicherweise braucht man sich um die interne Darstellungsweise des 8087 nur selten kümmern. Was man aber wissen muß, ist, welche Datentypen der 8087 verarbeiten kann.

7.2 Datentypen

Der 8087 kann sieben Datentypen verarbeiten:

- drei Arten von Ganzzahlen (Wort, kurz und lang)
- drei Arten von Fließkommazahlen (kurz, lang und temporär)
- gepackte Dezimalzahlen:

Datentyp	Bits	Stellen	Bereich
Wort-Ganzzahl	16	4	-32 768 bis 32 767
kurze Ganzzahl	32	9	$-2 \cdot 10^9$ bis $2 \cdot 10^9$
lange Ganzzahl	64	18	$-9 \cdot 10^{18}$ bis $9 \cdot 10^{18}$
kurze Fließkommazahl	32	6 oder 7	10^{-37} bis 10^{38}
lange Fließkommazahl	6	15 oder 16	10^{-307} bis 10^{308}
temporäre Fließkommazahl	80	19	10^{-4932} bis 10^{4932}
gepackte Dezimalzahl	80	18	18 Dezimalstellen + Vorzeichen

Die vierstellige Wort-Ganzzahl entspricht dem Datentyp *Integer* von BOSIC und wird zum Indizieren von Feldern und anderen Datenstrukturen verwendet. Da ein Wort Werte im Bereich -32 768 bis 32 767 enthalten kann, gibt es nicht viele Anwendungen, die die Typen kurze und lange Ganzzahl verwenden.

Die kurze und lange Fließkommazahl entsprechen den Datentypen *single* (einfache Genauigkeit) und *double* (doppelte Genauigkeit). Die kurzen Fließkommazahlen sind bis zu etwa sieben Dezimalstellen genau. Kurze Fließkommazahlen werden häufig bei der Dateneingabe verwendet; Berechnungen sollten jedoch mit der langen Fließkommazahl durchgeführt werden, um Rundungsfehler zu vermeiden. Lange Fließkommazahlen haben etwa eine Genauigkeit von 16 Dezimalstellen.

Das Format *temporäre Fließkommazahl* wird vom 8087 verwendet, um Zahlen in seinen internen Datenregistern zu speichern. Da dieses Format eine 64-Bit-Mantisse verwendet, kann jedes andere Format ohne Verlust an Genauigkeit in dieses umgewandelt werden. Die Größe von 80 Bit verhindert Rundungsfehler und Überlauf bei Zwischenberechnungen.

Die gepackten Dezimalzahlen schließlich werden hauptsächlich in geschäftlichen Bereichen verwendet. Sie können bis zu 18 signifikante Stellen enthalten, wobei jeweils zwei Ziffern in ein Byte „gepackt“ werden.

7.3 Befehlssatz

Der Befehlssatz des 8087 läßt sich in sechs Gruppen aufteilen:

- Datentransfer,
- Arithmetik,
- Vergleich,
- transzendente Berechnungen,
- Konstanten,

- Steuerung.

Die Datentransferbefehle schreiben oder lesen Zahlen in/aus dem Datenregisterstapel des 8087:

Name	Beschreibung
FBLD	lade gepackte Deimalzahl
FBSTP	lese und speichere gepackte Dezimalzahl
FILD	lade Ganzzahl
FIST	speichere Ganzzahl
FISTP	lese und speichere Ganzzahl
FLD	lade Fließkommazahl
FST	speichere Fließkommazahl
FSTP	lese und speichere Fließkommazahl
FXCH	Register austauschen

Die Vergleichsbefehle vergleichen die Zahl oben auf dem Stapel mit einer anderen Zahl im Stapel oder einer Speicherstelle. Vergleiche eignen sich dafür, die größte Zahl in einem Feld festzustellen oder zu ermitteln, ob eine Zahl kleiner, gleich oder größer als Null ist:

Name	Beschreibung
FCOM	Fließkommazahl vergleichen
FCOMP	Fließkommazahl vergleichen und lesen
FCOMPP	Fließkommazahl vergleichen und zweimal lesen
FICOM	Ganzzahl vergleichen
FICOMP	Ganzzahl vergleichen und lesen
FTST	Wert auf Stapel mit „0“ vergleichen
FXAM	Obersten Wert auf Stapel untersuchen

Die Steuerbefehle ermöglichen

- die Abfrage von Statusinformationen,
- die Änderung der Rundungsart,
- das Erlauben und Verboten von Unterbrechungen

und führen eine ganze Reihe andere Organisationsaufgaben aus:

Name	Beschreibung
FLDCW	LADE Steuerwort
FSTCW	speichere Steuerwort
FSTSW	speichere Statuswort
FSAVE	Status sichern
FRSTOR	Status wiederherstellen
FLDENV	lade Umgebung
FSTENV	speichere Umgebung
FWAIT	Warte (8088/8086 anhalten)
FINIT	8087 initialisieren (= zurücksetzen)
FENI	Unterbrechungen erlauben
FDISI	Unterbrechungen verbieten
FCLEX	Ausnahmen bereinigen
FINCSTP	Stapelzeiger erhöhen
FDECSTP	Stapelzeiger vermindern
FFREE	Register löschen
FNOP	keine Operation

Der Befehl 'FWAIT' erzeugt den 8088- bzw. 8086-Befehl 'WAIT', der den Prozessor davon abhält, auf eine Speicherstelle zuzugreifen, welche vom 8087 gerade verwendet wird.

Die arithmetischen Befehle erlauben die vier Grundrechenarten und andere nützliche Funktionen wie z.B. Wurzelfunktion und Absolutwert.

Die Befehle Subtrahieren und Dividieren existieren in zwei Formen. Mit der Standardform wird ein Quellwert von einem Zielwert subtrahiert, bzw. ein Zielwert durch einen Quellwert dividiert.. Mit der „umgekehrten“ Form wird ein Zielwert vom Quellwert subtrahiert, bzw. ein Quellwert durch einen Zielwert dividiert. Mit der umgekehrten Form können Ergebnisse im Speicher verbleiben:

Name	Beschreibung
FADD	Fließkommazahl addieren
FADDP	Fließkommazahl addieren und lesen
FIADD	Ganzzahl addieren
FSUB	Fließkommazahl subtrahieren
FSUBP	Fließkommazahl subtrahieren und lesen
FISUB	Ganzzahl subtrahieren
FSUBR	Fließkommazahl umgekehrt subtrahieren
FSUBRP	Fließkommazahl umgekehrt subtrahieren und lesen
FISUBR	Ganzzahl umgekehrt subtrahieren
FMUL	Fließkommazahl multiplizieren
FMULP	Fließkommazahl multiplizieren und lesen
FIMUL	Ganzzahl multiplizieren
FDIV	Fließkommazahl dividieren
FDIVP	Fließkommazahl dividieren und lesen
FIDIV	Ganzzahl dividieren
FDIVR	Fließkommazahl umgekehrt dividieren
FDIVRP	Fließkommazahl umgekehrt dividieren und lesen
FIDIVR	Ganzzahl umgekehrt dividieren und lesen
FSORT	Wurzel
FSCALE	mit Faktor 2 skalieren
FPREM	Rest
FRNDINT	auf Ganzzahl runden
FXTRACT	Exponent und Mantisse extrahieren
FABS	Absolutwert
FCHS	Vorzeichen ändern

Name	Beschreibung
F2XMI	$2^X - 1$
FYL2X	$Y * \log_2 X$
FYL2XPI	$Y * \log_2(X+1)$
FPTAN	Tangens
FPATAN	Arcustangens

Die transzendenten Befehle berechnen logarithmische und trigonometrische Funktionen:

Die Konstantenbefehle legen eine von sieben Konstanten auf den Stapel ab: Null, Eins, Pi oder vier logarithmische Werte. In allen Fällen werden sie mit der ganzen Genauigkeit (19 Dezimalstellen) des 8087 dargestellt:

7.4 Der interne Aufbau

Die numerischen Register verwalten nur „Extended-Precision“-Werte; andere Werte werden im Speicher abgelegt. Die numerischen Register werden bei der Programmierung häufig als Stackregister eingesetzt. Es gilt folgende Registeraufteilung:

Name	Beschreibung
FLDZ	lade Null
FLD1	lade Eins
FLDPI	Lade Pi
FLDL2T	Lade $\log_2 10$
FLDL2E	lade $\log_2 e$
FLDLG2	lade Logarithmus von 2
FLDLN2	lade natürlichen Logarithmus von 2

80-Bit-Register		16-Bit-Register	
R0	Extended-Precision-Floating-Point	Kontroll-Register	Rundung auf Datentyp
R1	19 signifikante Stellen	Status-Register	Vergleichsoperation BUSY-Flag, 8088 wartet
R2	Grenzen 10^{-4931} bis 10^{+4931}	TAF-Register	Statuskontrolle der numerischen Register auf gültige Zahlenformate
R3			
R4			
R5			
R6			
R7			

Am Beispiel für den Stapelzeiger soll der interne Aufbau des Registerstapels und des Statusregisters aufgezeigt werden.

Registerstapel (Basis für Stapelzeiger =5):

	4F	40	3F	0
R7	VZ	Exponent	Mantisse	ST2
R6				ST1
R5				ST(0)
R4				ST(7)
R3				ST(6)
R2				ST(5)
R1				ST(4)
R0				ST(3)

Kontrollregister:

Das Kontrollregister erlaubt zwischen zwei möglichen Behandlungsarten von unendlichen Zahlen zu unterscheiden. Die Standardmethode ist dabei der positive Abschluß des Zahlensystems.. Dabei werden vom 8087 sowohl positive als auch negative unendliche Zahlen als eine einzige vorzeichenlose Unendlichkeit behandelt.

Die andere Methode, der angepaßte Abschluß des Zahlensystems, unterscheidet zwischen positiven und negativwun unendlichen Zahlen.

Der 8087 gestattet außerdem unter vier mögliche Arten des Rundens von Zahlen auszuwählen. Gerundet werden Zahlen immer, wenn ein Ergebnis eine größere Genauigkeit erfordern würde, als das Zahlensystem zuläßt. Die Methode des Rundens entscheidet dabei, welche Zahl als Ergebnis ausgewählt wird. Es wird unterschieden in:

- das Runden auf die nächstgrößere Zahl des Systems,
- das Runden auf die nächstkleinere Zahl des Systems,
- dem Abrunden auf Null,
- dem Runden auf die nächste gerade Zahl.

Der 8087 erlaubt zudem die 64-Bit-Genauigkeit des temporären Gleitpunktformats einzuschränken.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
			IC		RC		PC		IEM		PM	UM	OM	ZM	DM	IM

mit (Zustandsmaske: 1 = Zustand nicht erlaubt):

- Bit 0: ungültige Operation (invalid operation)
- Bit 1: denormalisierter Operand (denormalized operand)
- Bit 2: Nulldivisions (Zero divide)
- Bit 3: Überlauf-Status(overflow)
- Bit 4: Unterlauf-Status(underflow)
- Bit 5: Genauigkeits-Status (precision)
- Bit 6: reserviert
- Bit 7: Interrupt-Status (interrupt request)
- Bit 8,9: Genauigkeitskontrolle
- Bit 10,11: Rundungskontrolle
- Bit 12: Unendlichkeitskontrolle
- Bit 13, 14, 15: reserviert

Statusregister:

Das Statusregister des 8087 teilt den aktuellen Zustand des Koprozessors mit. Das Statusregister enthält Bits für jeden Ausnahmezustand, so daß die Interrupt-Routine in jedem Fall den Grund der Störung erkennen kann. Das Statusregister verfügt außerdem über ein Bit, welches angibt, ob der Prozessor gerade arbeitet oder nicht. Dieses Bit wird auch nach außen weitergegeben, um eine Synchronisation mit dem 8088/8086 zu ermöglichen. Das Statusregister enthält zusätzlich den Pointer auf die aktuelle Stackspitze innerhalb der Register des 8087.

Der vermutlich am meisten benutzte Teil des Statusregisters ist das Bedingungscode-Register. Das Statusregister enthält dazu vier Bits, die von den Operationen des 8087 gesetzt werden. Zwei dieser Bedingungscode-Bits entsprechen dabei direkt dem Carry- und dem Null-Flag des 8088/8086. Diese Anordnung ist von Vorteil, wenn das Statusregister im Speicher abgelegt wird, da hier das höherwertige Byte des Statusregisters in das AH-Register abgelegt wird. Der Befehl SAHF setzt dann Carry- und Null-Flag in Abhängigkeit von den Ergebnissen des Vergleichs im 8087. Da alle Zahlen innerhalb des 8087 vorzeichenbehaftete Gleitpunktzahlen sind, genügen die beiden Flags immer, um die Reihenfolge von zwei Zahlen festzustellen.

Die restlichen beiden Bits des Bedingungscode-Registers werden in Zusammenhang mit einem speziellen Befehl des 8087 verwendet, welcher es erlaubt, einen Test auf alle vom 8087 unterstützten besonderen Zahlen durchzuführen. Da viele dieser Zahlen spezielle Verarbeitungsweisen erfordern, verfügt man mit dem Bedingungscode-Register über einen Mechanismus, diese besonderen Zahlen herauszufiltern.

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
B	C3		ST		C2	C1	C0	IR		PE	UE	OE	ZE	DE	IE

mit:

- Bit 0: Ungültigkeits-Status IE (invalid operation exception)
- Bit 1: Unmaskiert-Status DE (denormalized operand exception)
- Bit 2: Nulldivisions-Status ZE (Zero divide exception)
- Bit 3: Überlauf-Status OE (overflow exception)
- Bit 4: Unterlauf-Status UE (underflow exception)
- Bit 5: Genauigkeits-Status PE (precision exception)
- Bit 6: reserviert
- Bit 7: Interrupt-Status IR (interrupt request)
- Bit 8: Bedingungscode-Bit 0
- Bit 9: Bedingungscode-Bit 1
- Bit A: Bedingungscode-Bit 2
- Bit B: Stapelzeiger-Bit 0
- Bit C: Stapelzeiger-Bit 1
- Bit D: Stapelzeiger-Bit 2
- Bit E: Bedingungscode
- Bit F: Busy-Status B (busy)

Alle numerischen Werte werden in dem Registerstatus als temporäre Gleitkommazahl abgelegt. Der 8087 verarbeitet auch andere Zahlenformate, diese werden dann automatisch in das Temporary-real-Format konvertiert. Dazu sind die acht Stack-ähnlichen 80-Bit-Register vorhanden. Das logische Register „0“ ist das physikalische Register, auf welches der Stackpointer zeigt. Das folgende Register hat die logische Nummer „1“ usw.

Das Statusregister hat ein 16-Bit-Format und setzt sich aus acht Statusbits, einem 4-Bit-Bedingungscode und einem 3-Bit-Stapelzeiger zusammen.

7.5 Datenaustausch zwischen 8088/8086 und 8087

Der Austausch der Daten erfolgt über den gemeinsamen Adreß-Datenbus AD0 bis AD6. Der Adreßbus hat einen Umfang von 20 Leitungen, wobei man die Besonderheiten von A16 bis A19 beachten muß, denn diese Leitungen lassen sich noch als Statusleitungen einsetzen. BHE/S7 wird nur in Verbindung mit dem 8086 benötigt, wenn die Daten von dem niederwertigen auf das höherwertige Byte umgeschaltet werden müssen.

Die Instruktionen des 8087 benötigen zur Ausführung bis zu 885-Taktzyklen. In dieser Zeit kann der 8088/8086 weiterarbeiten. Bei Auftreten einer zweiten Instruktion für den 8087 muß der 8088/8086 allerdings auf die Beendigung der ersten Funktion warten.

Die drei Statusleitungen S0, S1 und S2 sind mit dem Mikroprozessor und dem Bus-Controller 8288 verbunden und damit erhalten der Koprozessor und der 8288 die globalen Steuersignale, die intern noch decodiert werden. Der 8288

S2	S1	S0	Funktion
0	X	X	ohne Funktion
1	0	0	ohne Funktion
1	0	1	Lesebetrieb, Datenspeicher lesen
1	1	0	Schreibbetrieb, Datenspeicher schreiben
1	1	1	passiv, kein Buszyklus

kann diese Steuersignale nur aufnehmen, aber der 8087 erzeugt eigene Signale. Es gelten folgende Statussignale: Durch den Schreib- und Lese-Betrieb hat der 8087 die Möglichkeit, einen Operanden in den Datenspeicher zu schreiben oder eine gespeicherte Information auszulesen.

Der 8087 ist direkt an dem Daten-/Adreßbus des Mikroprozessors angeschlossen und hat daher den gleichen Datenfluß mit Programmanweisungen wie der Mikroprozessor.

Der Mikroprozessor ist verantwortlich für die Erzeugung aller Adressen, sowohl für Programmcode als auch für die Daten. Operanden (in der Regel Daten) und Op-Codes (in der Regel Anweisungen) werden über den Daten-/Adreßbus an den Koprozessor übergeben.

Sobald der Mikroprozessor einen Befehl für den 8087 entdeckt, wartet er bis der Koprozessor bereit ist, läßt dann den Koprozessor mit der Ausführung der Anweisung beginnen und fährt mit dem nächsten Befehl fort. Falls beim Eintreffen des nächsten Koprozessor-Befehls der 8087 noch mit der Bearbeitung des letzten Befehls beschäftigt ist, wartet der Mikroprozessor auf den 8087.

Dieser Ablauf ist für den Programmierer nicht transparent; für ihn arbeiten die beiden Prozessoren wie eine Einheit zusammen.

Die Synchronisation zwischen Mikroprozessor und 8087 übernimmt eine Leitung, die den BUSY-Ausgang vom 8087 mit dem Test-Eingang vom Mikroprozessor verbindet. Den Datentransfer zwischen Mikroprozessor und 8087 übernehmen die beiden Leitungen RQ/GT (Request/Grant = Anfrage/Genehmigen), wobei nur die RQ/GT0-Leitung benötigt wird. RQ/GT1 ist mit 5V zu verbinden und hat innerhalb des PC's keine Funktion. Für den Datentransfer fordert der 8087 über die RQ/GT0-Verbindung vom Mikroprozessor die Controller über den lokalen Bus an. Der Mikroprozessor quittiert diese Aufforderung über dieselbe Leitung, denn diese Verbindung arbeitet bidirektional. Für die Steuerung des Warteschlangenstatus (Instruction Queue-Status) benötigt man die beiden Eingänge QS0 und QS1 (Queue-Status) die mit den gleichnamigen Ausgängen des Mikroprozessors verbunden sind. Dann kann der 8087 die Vorgänge der Warteschlange im Mikroprozessor verfolgen:

QS1	QS0	
0	0	kein Ergebnis
0	1	erstes Befehlsbyte wird aus der Schlange geholt
1	0	restliche Bytes in der Schlange werden gelöscht
1	1	weiteres Befehlsbyte wird aus der Schlange geholt

Hauptaufgabe der beiden QS-Verbindungen ist die Steuerung der Befehlsübernahme vom Mikroprozessor zum Koprozessor. Damit erkennt der 8087, ob noch Befehle in der Warteschlange sind und damit reduziert sich der Decodierungsmechanismus im 8087 auf ein Minimum.

8. Fehlersuche und -beseitigung

8.1 Syntaxfehler

Ein Programm besteht aus einer großen Zahl einzelner Anweisungen. Diese Anweisungen müssen den Definitionen der verwendeten Programmiersprache entsprechen. Dies wird die Syntax eines Programmes genannt. Die Syntax wird vom Compiler überprüft, der jede Unstimmigkeit mit einer (mehr oder weniger) entsprechenden Fehlermeldung aufzeigt.

Aus einem Quelltext mit syntaktischen Fehlern kann nie ein ablauffähiges Programm entstehen. Die Spannweite der Syntaxfehler ist sehr groß. Es fängt bei der falschen Schreibweise der Befehle an, geht über fehlende oder falsche Parameterangaben bei Unterprogrammaufrufen bis hin zu fehlenden Befehlstrennzeichen.

8.2 Logische Fehler

Sind die Syntaxfehler beseitigt, funktioniert das Programm wahrscheinlich trotzdem nicht so, wie es eigentlich sollte. Diese im Leben eines Programmierers alltägliche Erscheinung kann viele Ursachen haben. Viele Fehler rühren daher, daß eine Aufgabenstellung und deren Lösungsstrategie nicht in allen Details durchdacht wurden und übereinstimmen. Logische Fehler unterlaufen einem Programmierer häufig bei der Auswahl der bedingten Sprunganweisung. Es passiert schnell, daß bei einem Vergleich die mögliche „Gleichheit“ vergessen wurde.

8.3 Einsatz eines Debuggers

Um ein Programm auf Ungereimtheiten zu untersuchen, gibt es zwei Wege.

Man kann an allen kritischen Programmstellen Anweisungen einfügen, um Variablenwerte auf dem Bildschirm anzuzeigen. Diese Strategie hat jedoch ein wesentliches Manko: der Quelltext des Programms muß zur Fehlersuche verändert werden. Sind alle Fehlerquellen beseitigt, muß das Programm erneut bearbeitet werden, um alle Bildschirm Ausgaben wieder zu beseitigen. Jedes Editieren eines Programmes birgt in sich jedoch die Möglichkeit für neue Fehlerquellen.

Ein besserer Weg ist der Einsatz eines Debuggers (engl.: to debug - Fehler beseitigen). Der Turbo-Debugger von Borland hat einige grundlegende Eigenschaften:

- das Programm läßt sich an jeder beliebigen Stelle unterbrechen („Breakpoint“)
- jedes Register und jede Variable kann angezeigt und verändert werden
- es lassen sich auf Assemblerebene einzelne Befehle verändern

Das „debuggen“ erfolgt dabei wahlweise auf zwei Ebenen: Entweder auf Quelltextebene, wenn das Programm mit einer entsprechenden Compileroption übersetzt wurde. Die Untersuchung eines Programmes auf Prozessorebene erfolgt immer in Maschinencode und steht jederzeit zur Verfügung.

Zur Erleichterung der Fehleraufspürung können bedingte Programmunterbrechungen eingefügt werden. Auf diese Weise kann ein Programm bis zum Eintreten eines bestimmten Ereignisses ablaufen. Nun kann der Zustand aller relevanten Größen untersucht und gegebenenfalls verändert werden.

Alle Änderungen, die am Programm vorgenommen werden, wirken sich nur temporär aus, d.h. die notwendigen Modifikationen sind am Quelltext getrennt durchzuführen. Borland's Turbo-Debugger bietet die Möglichkeit, das Debugging zu protokollieren, so daß handschriftliche Notizen entfallen können.

8.4 Debug-Strategie

Der Debugger wird unter Angabe des zu untersuchenden Programms mit

td <Dateiname>

gestartet. Mit der Taste <F9> wird das Programm gestartet und läuft ohne Unterbrechungen ab. Tritt kein Fehler auf, meldet der Debugger nach Programmende

Terminated, exit code 0.

Tritt jedoch ein Fehler auf, so erscheint ein Textfehler mit der Meldung

Runtime Error

und die dazugehörige Befehlszeile wird invers dargestellt. Jetzt kann im Handbuch der Programmiersprache nachgeschlagen werden, auf welche Ursache der Laufzeitfehler hinweist. Entsprechend dieser Auskunft werden die einzelnen Variablen geändert.

Mit der Tastenkombination <Strg> und <F7> wird der Inhalt der Variablen angezeigt. Mit der Taste <F2> kann im Zweifelsfall ein Haltepunkt gesetzt werden und das Programm mit <F9> neu gestartet werden. Die Ausführung wird bei Erreichen des Haltepunktes automatisch unterbrochen. Mit <Strg> und <F4> wird der Wert der Variablen überprüft und im Zweifelsfall geändert. Mit der Taste <F7> läßt man nun den nächsten Befehl aus. Tritt nun kein Fehler mehr auf, ist die Ursache gefunden und der Quelltext entsprechend zu ändern.

8.5 Checkliste

Die folgende Checkliste von J. Erdweg enthält die wichtigsten Grundregeln bei der Assembler-Programmierung und zeigt die schlimmsten Auswirkungen bei Fehlern auf. Ist in der Tabelle lediglich ein „falsches Ergebnis“ vermerkt, kann es bei bestimmten Systemkonstellationen („verbogene“ Zeiger usw.) dennoch zu einem Systemabsturz kommen; der Systemabsturz ist *immer* ein falsches Ergebnis.

Grundregel	Auswirkung bei Fehler
am Ende eines Programms muß ein Rücksprung zum Betriebssystem erfolgen; dies kann durch die Funktion 4Ch (INT 21h) erfolgen	Systemabsturz
für jede „PUSH“-Anweisung muß eine entsprechende „POP“-Anweisung vorhanden sein und umgekehrt	Systemabsturz
am Ende einer Prozedur (PROC) muß eine Rücksprunganweisung stehen	Systemabsturz
der Rücksprung aus einer Prozedur (PROC) muß dem Typ des Unterprogramms entsprechen (NEAR bzw. FAR)	Systemabsturz
wenn eine Prozedur (PROC) über den Stapel mit Parametern versorgt wird, müssen diese vor dem eigentlichen Rücksprung beseitigt werden	Systemabsturz
alle Register, die in einem Unterprogramm verwendet werden, sollten auf dem Stapel gesichert werden, wenn das Hauptprogramm deren Inhalte benötigt	Falsche Ergebnisse
der Stack eines Programmes muß ausreichend groß sein; bei „EXE“-Dateien sollte eine Minimalgröße von 512 Byte angesetzt werden	Systemabsturz durch Überschreiben von Programmen; Falsche Ergebnisse durch Überschreiben von Daten
die Reihenfolge der Operanden sollte überprüft werden; die Reihenfolge legt eindeutig fest, in welches Register das Ergebnis geschrieben wird	Falsche Ergebnisse
Zugriffe auf Werte im Datensegment dürfen nicht mit dem Zugriff auf den Offset dieses Wertes verwechselt werden; für die Adresse einer Speicherstelle immer den expliziten Operator „OFFSET“ verwenden	Falsche Ergebnisse
bei Stringbefehlen (LODS, MOVS usw.) den korrekten Zustand des Direction-Flag kontrollieren, da hiervon die Wirkungsrichtung der Befehle abhängt	Systemabsturz durch Überschreiben von Programmen; Falsche Ergebnisse durch Überschreiben von Daten
Stringbefehle verändern den Inhalt der Indexregister entsprechend dem Typ des Datenzugriffs, d.h. um 1 bei Bytes und um 2 bei Wörtern; ein explizites In- bzw. Dekrementieren von Zeigern und Zählern muß dem Rechnung tragen	Falsche Ergebnisse
bedingte Sprünge sollen immer dann ausgeführt werden, wenn die Flags definiert gesetzt werden, da sehr viele Befehle den Inhalt des PSW verändern	Falsche Ergebnisse
bei der Verwendung von arithmetischen Operationen wird häufig mehr als ein Register zur Speicherung des Ergebnisses verwendet, ohne das dies ausdrücklich gefordert wird (MUL, DIV usw.)	Falsche Ergebnisse

Anhang A: Beispiele

Einbinden von Unterprogrammen

Beispielprogramm zur Demonstration, wie Unterprogramme mit dem Hauptprogramm in der gleichen Quelldatei programmiert werden:

```
*****
;*
;* Beispielprogramm: Interne Unterprogramme *
;*
*****

; TASM_Festlegungen:
        IDEAL          ; TASM-Ideal-Modus
        MODEL TINY    ; kleinste Programmart: COM

; Gleichsetzungen:
BILDAUS    equ 9      ; Funktion: Bildschirmausgabe
DOSFUNK    equ 21h    ; MS-DOS Interrupt-Nummer
ENDE       equ 4CH    ; Funktion: Programm beenden
ClearBuffer equ 0CH   ; Funktion: Puffer löschen
DirectConsoleIN equ 07h ; Funktion: Standard-Input
;          ;          ; wait: +
;          ;          ; echo: -
;          ;          ; break: -
CR         equ 13     ; <Carriage-Return>
LF         equ 10     ; <Line Feed>
EDB       equ 2$     ; „Ende der Botschaft“

;=====

; Anfang des Codesegments:
        CODESEG      ; Beginn des Codesegments

        ORG 100h    ; Anfang für COM-File
ANFANG: JMP START

;=====

; Datenbereich

STRING    db 27,2J$2 ; ANSI-ESC-Sequenz: Bildschirm löschen
; Ausgabetexte

TEXT1     db 'Noch ist der Bildschirm nicht gelöscht.'
          db CR, LF, EDB

TEXT2     db 'Jetzt ist der Bildschirm sauber.'
          db CR, LF, EDB

TEXT3     db 'Bitte eine beliebige Taste drücken ...'
          db CR, LF, EDB

;=====
```

*; Unterprogramm: ClearScreen
; Wirkung: Löschen des Bildschirms*

PUBLIC ClearScreen

*PROC ClearScreen NEAR
mov dx, offset STRING ; DX: Adresse der ESC-Sequenz
mov ah, BILDAUS
int DOSFUNK
ret ; zurück ins Hauptprogramm*

ENDP ClearScreen

;=====

*; Unterprogramm: Wait4Key
; Wirkung: Warten auf beliebigen Tastendruck („weit for kii“)*

PUBLIC Wait4Key

*PROC Wait4Key NEAR
mov ah, ClearBuffer ; AH: DOS-Funktion
mov al, DirectConsoleIN ; AL: Unterfunktion
int DOSFUNK
xor ah, ah ; AH löschen
ret ; zurück ins Hauptprogramm*

ENDP Wait4Key

;=====

; Hauptprogramm

*Start: mov dx, OFFSET TEXT1 ; Text 1 ausgeben
mov ah, BILDAUS
int DOSFUNK

mov dx, OFFSET TEXT3 ; Text 3 ausgeben
mov ah, BILDAUS
int DOSFUNK

call Wait4Key ; Unterprogramm aufrufen

call ClearScreen ; Unterprogramm aufrufen

mov dx, OFFSET TEXT2 ; Text 2 ausgeben
mov ah, BILDAUS
int DOSFUNK

mov dx, OFFSET TEXT3 ; Text 3 ausgeben
mov ah, BILDAUS
int DOSFUNK

call Wait4Key ; Unterprogramm aufrufen

xor al, al ; Errorlevel := 0
mov ah, ENDE
int DOSFUNK

end ANFANG*

Equipment-Check

Interrupt Typ 11h ermöglicht die Erkennung der Systemkonfiguration eines PC's. Der Interrupt 11h ruft die Routine EQUIPMENT (F000:F84D) auf. In AX wird der Equipment-Status gespeichert. Es gilt:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
mit:	Bit 0	1= Laufwerk, 0 = kein Laufwerk														
	Bit 2/3	Speicher auf Systemplatine: 00 = 16 Kbyte 01 = 32 Kbyte 10 = 48 Kbyte 11 = 64 Kbyte														
	Bit 4/5	Initialisierungs-Video-Modus 00 = nicht benutzt 01 = 40X25 S/W (Farbkarte) 10 = 80X25 S/W (Farbkarte) 11 = 80X25 S/W (S/W-Karte)														
	Bit 6/7	Anzahl Laufwerke (nur wenn Bit 0 = 1) 00 = 1 Laufwerk 01 = 2 Laufwerke 10 = 3 Laufwerke 11 = 4 Laufwerke														
	Bit 9/10/11	Anzahl RS232-Karten														
	Bit 12	0 = kein Spieleadapter angeschlossen 1 = Spieleadapter angeschlossen														
	Bit 14/15	Anzahl angeschlossener Drucker														

Testprogramm, ob das System eine Farbgrafikkarte besitzt; wenn ja, erscheint keine Meldung, wenn nein, so erscheint der Hinweis „Achtung: Keine Farbgrafik-Karte“

```

stack      segment para stack 'stack'
           db 64 dup('stack')
stack      ends

textdata   segment para 'data'
string     db 'Achtung: Keine Farbgrafik-Karte$'
textdata   ends

textcode   segment para 'code'
           assume cs:textcode,ds:textdata,ss:stack

check      proc
           push ax                               ; Rette Register auf Stack
           push dx

teste:     int 11h                               ; Equipment-Test
           and ax,0000000000110000b           ; Maskierung von AX:
           cmp al, 00010000b                   ; durch AND-Verknüpfung werden alle
           je pgm_ende                          ; Bits außer 4 und 5 auf Null gesetzt,
           cmp al, 00100000b                   ; anschließend auf 01 bzw. auf 10 verglichen
           je pgm_ende

warnung:   mov ax, seg textdata                 ;initialisiere DS
           mov ds, ax
           lea dx, string                       ;initialisiere DX
           mov ah, 9                            ; gebe Textstring aus
           int 21 h

```

```

pgm_ende:  pop dx                ;hole Register vom Stack
           pop ax
           mov ah, 4ch           ;Rückkehr nach DOS
           int 21h

check      endp
textcode   ends
end check

```

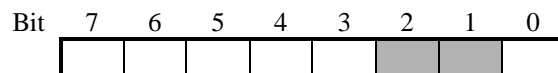
Drucker-Ansteuerung

Mit Interrupt Typ 17h können bis zu 3 Druckern gesteuert werden. Dieser Interrupt ruft die Routine PRINTER_IO (F000:EFD2) auf.

Mit Nr. 0 (nach AH) wird ein Charakter (nach AL) gedruckt,

mit Nr. 1 (nach AH) wird der Drucker initialisiert,

mit Nr. 2 (nach AH) wird der Drucker-Status gelesen. Dieser Status wird bei jeder der drei Operationen in AH zurückgespeichert:



mit:

Bit 0	1 = Time out
Bit 3	1 = E/A-Fehler
Bit 4	1 = selektiert
Bit 5	1 = Papierende
Bit 6	1 = Empfangsbestätigung
Bit 7	1 = not busy.

Mit Interrupt 21h ist ebenfalls eine Druckerausgabe möglich. Hierzu wird das zu druckende Zeichen nach DL geschrieben und Funktion Nr. 5 aufgerufen. Der Drucker-Status wird hierbei nicht zurückgelesen.

Das folgende Listing gibt eine Druckzeile mit 80 Zeichen (von „!“ bis „p“) aus:

```

stack      segment para stack 'stack'
           db 64 dup(?)

stack      ends

druckcode  segment para 'code'
           assume cs:druckcode, ds:druckcode, ss:stack

drucke     proc
           mov cx, 80                ; drucke 80 Zeichen
           mov dl, 21h              ; beginne mit Zeichen 21h (= !)
           mov ah, 5                 ; wähle Funktions-Aufruf Nr. 5

naechst:   int 21h                  ; drucke Zzeichen
           inc dl                    ; inkrementiere dl = nächster Zeichencode
           loop naechst              ; drucke nächstes Zeichen

           mov ah, 4ch               ;Zurück nach DOS
           INT „!H

drucke     endp
druckcode  ens
end

```

Interrupt Typ 5 ermöglicht aus einem Programm heraus das Abdrucken des Bildschirms. Die Funktion ist die gleiche wie die durch Taste „Druck“ (= „PrtSc“) ausgelöste, wird aber aus einem Programm heraus gesteuert. Dieser Interrupt ruft die Routine PRINT_SCREEN (F000:FF54) auf. Die aktuelle Cursorposition wird gerettet, der Bildschirminhalt zum Drucker geschickt und die Cursorposition wiederhergestellt.

In der Routine steht an der Stelle 50:0 der Status der Routine:

0 = Druckoperation wurde erfolgreich durchgeführt,
1 = während der Durchführung.

Tritt ein Fehler auf, so ist der Inhalt der Speicherstelle 0FFh.

Anhang B: Die BIOS-Funktionen

Über die Interrupts 10h bis 1Ah können die verschiedenen Funktionen erreicht werden, die das ROM-BIOS zur grundlegenden Kommunikation zwischen einem Programm und der Hardware zur Verfügung stellt. Dazu zählen neben Funktionen zum Zugriff auf die Video-Hardware, Tastatur, Festplatten und Diskettenlaufwerke, aber auch die Abfrage von Konfigurationsdaten, sowie die Programmierung der seriellen und parallelen Schnittstelle und der batteriegepufferten Echtzeituhr.

Hier zunächst eine Übersicht über die verschiedenen Interrupts und ihre Dienste. Es ist zu beachten, daß die verschiedenen Funktionen des Interrupts 13h getrennt nach ihrem Einsatz in bezug auf Disketten- und Festplattenlaufwerke zweimal aufgeführt werden.

Sofern nicht anders vermerkt, stehen die verschiedenen Funktionen auf allen Arten von PCs zur Verfügung. Eine ganze Reihe von Funktionen gibt es aber erst seit der Einführung des XT's, andere erst seit der Einführung des AT's.

Interrupt 10h Bildschirm

Funktion	Beschreibung
00h	Setzen des Video-Modus
01h	Definition des Erscheinungsbildes des Cursors
02h	Positionierung des Cursors
03h	Auslesen der Cursor-Position
04h	Auslesen der Lichtstiftposition
05h	Auswahl der aktuellen Bildschirmseite
06h	Textzeilen nach oben schieben (scrollen)
07h	Textzeilen nach unten schieben (scrollen)
08h	Auslesen eines Zeichens/Farbe
09h	Schreiben eines Zeichens/Farbe
0Ah	Schreiben eines Zeichens
0Bh/00h	Auswahl der Rahmen-/Hintergrundfarbe
0Bh/01h	Auswahl der Farbpalette
0Ch	Schreibe Grafikpunkt
0Dh	Lese Grafikpunkt
0Eh	Schreiben eines Zeichens
0Fh	Auslesen des Video-Modus

13h Ausgabe einer Zeichenkette (ab AT)

Interrupt 11h Feststellung der Konfiguration

Interrupt 12h Feststellung der Speichergröße

Interrupt 13h Diskette

Funktion	Beschreibung
00h	Reset
01h	Status Lesen
02h	Lesen
03h	Schreiben
04h	Verifizieren
05h	Formatieren
06h	Festplatte
07h	Festplatte
08h	Format abfragen (ab AT)
09h	Festplatte
0Ah	Festplatte
0Bh	Festplatte
0Ch	Festplatte
0Dh	Festplatte

Funktion	Beschreibung
0Eh	Festplatte
0Fh	Festplatte
10h	Festplatte
11h	Festplatte
12h	Festplatte
13h	Festplatte
14h	Festplatte
15h	Feststellung des Laufwerktyps (ab AT)
16h	Feststellung eines Diskettenwechsels (ab AT)
17h	Diskettenformat festlegen (ab AT)
18h	Diskettenformat festlegen (ab AT)

Interrupt 13h Festplatte

Funktion	Beschreibung
00h	Reset (ab XT)
01h	Status lesen (ab XT)
02h	Lesen (ab XT)
03h	Schreiben (ab XT)
04h	Verifizieren (ab XT)
05h	Formatieren (ab XT)
08h	Format erfragen (ab XT)
09h	Anpassung fremder Laufwerke (ab XT)
0Ah	Erweitertes Lesen (ab XT)
0Bh	Erweitertes Schreiben (ab XT)
0Ch	Schreib-/Lesekopf bewegen (ab XT)
0Dh	Reset (ab XT)
0Eh	Controller-Lese-Test (PS/2)
0Fh	Controller-Schreib-Test (PS/2)
10h	Ist das Laufwerk bereit? (ab XT)
11h	Rekalibrieren des Laufwerks (ab XT)
12h	Controller-RAM-Test (PS/2)
13h	Laufwerk-Test (PS/2)
14h	Controller-Diagnose (ab XT)
15h	Feststellung des Laufwerktyps (ab AT)

Interrupt 14h Serielle Schnittstelle

Funktion	Beschreibung
00h	Initialisierung
01h	Zeichen ausgeben
02h	Zeichen einlesen
03h	Status erfragen

Interrupt 15h Alter Kassetten-Interrupt (ab AT)

Funktion	Beschreibung
83h	Flag nach Zeitintervall setzen (ab AT)
84h/00h	Abfrage des Status der Feuerknöpfe der Joysticks (ab AT)
84h/01h	Abfrage der Stellung der Joysticks (ab AT)
85h	SysReq-Taste betätigt (ab AT)
86h	Warten (ab AT)
87h	Speicherbereiche verschieben (ab AT)
88h	Speichergröße über 1 MByte ermitteln (ab AT)
89h	Umschaltung in den Protected Mode (ab AT)

Interrupt 16h Tastatur

Funktion	Beschreibung
00h	Zeichen auslesen
01h	Zeichen vorhanden?
02h	Status der Tastatur erfragen
03h	Wiederholrate einstellen (ab AT)
05h	Tastendruck simulieren (ab AT)
10h	Tastaturabfrage für erweiterte Tastaturen (ab AT)
11h	Tastaturabfrage für erweiterte Tastaturen (ab AT)

Interrupt 17h (paralleler) Drucker

Funktion	Beschreibung
00h	Zeichen ausgeben
01h	Drucker initialisieren
02h	Status des Druckers erfragen

Interrupt 18h ROM-BASIC

Interrupt 19h Booten des Rechners

Interrupt 1Ah Datum und Zeit

Funktion	Beschreibung
00h	Zeit-Zähler auslesen
01h	Zeit-Zähler setzen
02h	Auslesen der Echtzeit-Uhr (ab AT)
03h	Setzen der Echtzeit-Uhr (ab AT)
04h	Auslesen des Datums aus der Echtzeit-Uhr (ab AT)
05h	Setzen des Datums der Echtzeit-Uhr (ab AT)
06h	Alarmzeit setzen (ab AT)
07h	Alarmzeit löschen (ab AT)

Anhang C: Übersicht der Funktionen des Interrupts 21h

Zeicheneingabe

Funktion	Beschreibung
01h	Zeicheneingabe mit Ausgabe
03h	Empfang eines Zeichens von der seriellen Schnittstelle
06h	Direkte Zeichenein-/ausgabe
07h	Direkte Zeicheneingabe ohne Ausgabe
08h	Zeicheneingabe ohne Ausgabe
0Ah	Eingabe einer Zeichenkette
0Bh	Lese Eingabestatus
0Ch	Lösche Eingabepuffer und rufe Eingabefunktion auf

Zeichenausgabe

Funktion	Beschreibung
02h	Ausgabe eines Zeichens
04h	Ausgabe eines Zeichens auf die serielle Schnittstelle
05h	Ausgabe auf parallele Schnittstelle
06h	Direkte Zeichenein-/ausgabe
09h	Ausgabe einer Zeichenkette

Programmbeendigung

Funktion	Beschreibung
00h	Programm beenden
31h	Programm beenden, aber im Speicher belassen
4Ch	Programm mit Ende-Code beenden

Zugriff auf Unterverzeichnisse

Funktion	Beschreibung
39h	Unterverzeichnis erstellen
3Ah	Unterverzeichnis löschen
3Bh	Aktuelles Verzeichnis setzen
47h	Aktuelles Verzeichnis ermitteln

RAM-Speicher-Verwaltung

Funktion	Beschreibung
48h	RAM-Speicher reservieren
49h	RAM-Speicher freigeben
4Ah	Größe eines Speicherbereichs ändern
58h/00h	Konzept der Speicherverteilung lesen
58h/01h	Konzept der Speicherverteilung setzen
58h/02h	Einbindung der UMBs abfragen
58h/03h	Einbindung der UMBs festlegen

Uhrzeit und Datum

Funktion	Beschreibung
2Ah	Datum abfragen
2Bh	Datum setzen
2Ch	Uhrzeit abfragen
2Dh	Uhrzeit setzen

Zugriff auf Gerätetreiber

Funktion	Beschreibung
44h/00h	IOCTL: Lesen des Geräte-Attributs
44h/01h	IOCTL: Setzen des Geräte-Attributs
44h/02h	IOCTL: Daten von einem Zeichentreiber empfangen
44h/03h	IOCTL: Daten an einen Zeichentreiber senden
44h/04h/1	IOCTL: Daten von einem Blocktreiber empfangen
44h/04h/2	DBLSPC: Internen Cache schreiben
44h/04h/3	DBLSPC: Internen Cache schreiben und invalidieren
44h/05h	IOCTL: Daten an einen Blocktreiber übertragen
44h/06h	IOCTL: Eingabestatus abfragen
44h/07h	IOCTL: Ausgabestatus abfragen
44h/08h	IOCTL: Ist das Medium wechselbar?
44h/09h	IOCTL: Device-Remote-Test
44h/0Ah	IOCTL: Handle-Remote-Test
44h/0Bh	IOCTL: Zugriffswiederholung setzen
44h/0Ch	IOCTL: Kommunikation mit einem Zeichentreiber
44h/0Dh	IOCTL: Kommunikation mit einem Blocktreiber
44h/0Eh	IOCTL: Letzte Laufwerksbezeichnung ermitteln
44h/0Fh	IOCTL: Nächste Laufwerksbezeichnung definieren
44h/10h	IOCTL-Unterstützung auf Handle-Ebene abfragen
44h/11h	IOCTL-Unterstützung auf Geräte-Ebene abfragen

Diskettenübertragungsbereich

Funktion	Beschreibung
1Ah	Setzen der DTA-Adresse
2Fh	DTA ermitteln

Directory durchsuchen

Funktion	Beschreibung
11h	Suche ersten Directory-Eintrag (FCB)
12h	Suche nächsten Directory-Eintrag (FCB)
4Eh	Ersten Directory-Eintrag suchen (Handle)
4Fh	Nächsten Directory-Eintrag suchen (Handle)

Dateizugriff (FCB)

Funktion	Beschreibung
0Fh	Datei schließen (FCB)
13h	Datei(en) löschen (FCB)
14h	Sequentielles Lesen (FCB)
15h	Sequentielles Schreiben (FCB)
16h	Erstellen oder Leeren einer Datei (FCB)
17h	Datei(en) umbenennen (FCB)
21h	Wahlfreies Lesen (FCB)
22h	Wahlfreies Schreiben (FCB)
23h	Lese Dateigröße (FCB)
24h	Setze Datensatznummer
27h	Wahlfreies Lesen mehrerer Datensätze (FCB)
28h	Wahlfreies Schreiben mehrerer Datensätze (FCB)
29h	Dateinamen in FCB übertragen

Dateizugriff (Handle)

Funktion	Beschreibung
3Ch	Datei erstellen oder leeren (Handle)
3Dh	Datei öffnen (Handle)
3Eh	Datei schließen (Handle)
3Fh	Datei lesen (Handle)
40h	Datei beschreiben (Handle)
41h	Datei löschen (Handle)
42h	Dateizeiger bewegen (Handle)
45h	Handle verdoppeln
46h	Handles angleichen
56h	Datei umbenennen oder verschieben (Handle)
5Ah	Temporäre Datei erstellen (Handle)
5Bh	Neue Datei erstellen (Handle)
5Ch/00h	Bereich einer Datei gegen Zugriff schützen
5Ch/01h	Freigabe eines gesperrten Bereichs in einer Datei
6Ch	Erweiterte OPEN-Funktion

Netzwerk-Aufrufe

Funktion	Beschreibung
5Eh/00h	Namen des Rechners im Netzwerk ermitteln
5Eh/02h	Initialisierungs-String für Netzwerkdrucker festlegen
5Eh/03h	Initialisierungs-String für Netzwerkdrucker ermitteln
5Fh/02h	Eintrag aus der Netzwerkliste holen
5Fh/03h	Eintrag in der Netzwerkliste definieren
5Fh/04h	Eintrag aus der Netzwerkliste entfernen

Zugriff auf Interrupt-Vektoren

Funktion	Beschreibung
25h	Setze Interrupt-Vektor
35h	Inhalt eines Interrupt-Vektors auslesen

Zugriff auf Disketten/Festplatten

Funktion	Beschreibung
0Dh	Reset der Blocktreiber
0Eh	Auswahl des aktuellen Laufwerks
19h	Gerätebezeichnung des aktuellen Laufwerks erfragen
1Bh	Informationen über das aktuelle Laufwerk einholen
1Ch	Informationen über ein beliebiges Laufwerk einholen
1Fh	DPB-Zeiger für das aktuelle Laufwerk ermitteln
32h	Zeiger auf DPB für ein beliebiges Laufwerk ermitteln
36h	Verbleibende Plattenkapazität ermitteln
53h	BPB in DPB umsetzen

Zugriff auf den PSP

Funktion	Beschreibung
6h	Erstelle neuen PSP
50h	Aktiven PSP setzen
51h	Aktiven PSP ermitteln
55h	Neuen PSP erstellen
62h	Adresse des PSP ermitteln

Zugriff auf DOS-Flags

Funktion	Beschreibung
2Eh	Setzen des Verify-Flags
33h/00h	Lesen des Break-Flags
33h/01h	Setzen des Break-Flags
34h	DOS Zeiger auf das INDOS-Flag ermitteln
37h/00h	Kennzeichen für Kommandozeilen-Schalter ermitteln
37h/01h	Kennzeichen für Kommandozeilen-Schalter setzen
52h	Zeiger auf DOS-Info-Block ermitteln
54h	Verify-Flag lesen

Zugriff auf Datei-Informationen

Funktion	Beschreibung
43h/00h	Attribut einer Datei ermitteln
43h/01h	Attribut einer Datei setzen
57h/00h	Datum und Uhrzeit der letzten Modifikation einer Datei ermitteln
57h/01h	Datum und Uhrzeit der letzten Modifikation einer Datei setzen

Zugriff auf landesspezifische Parameter

Funktion	Beschreibung
38h	Landesspezifische Symbole und Formate ermitteln
38h/00h	Landesspezifische Symbole und Formate ermitteln
38h/01h	Land setzen

Verschiedene Funktionen

Funktion	Beschreibung
30h	DOS-Versionsnummer ermitteln
4Bh/00h	EXEC: anderes Programm ausführen
4Bh/03h	EXEC: anderes Programm als Overlay laden
4Bh/05h	EXEC: eigene EXECs anpassen
4Dh	Ende-Code ermitteln
59h	Erweiterte Fehlerinformationen einholen
60h	Dateinamen erweitern
66h/01h	Aktuelle Code-Page ermitteln
66h/02h	Aktuelle Code-Page festlegen
67h	Anzahl der verfügbaren Handles festlegen
68h	Dateipuffer leeren

Anhang D: Der Multiplexer-Interrupt 2Fh

Der Interrupt 2Fh wird als Multiplexer-Interrupt bezeichnet, weil er eine Schnittstelle zu verschiedenen residenten DOS-Programmen bietet und auch TSR-Programmen die Möglichkeit gibt, ihre Funktionen für andere Programme verfügbar zu machen. Hier nun eine Übersicht der Funktionen des Interrupts 2Fh

MUX-Code	Funktion	Beschreibung
01h	00h	PRINT: Installations-Status abfragen
01h	01h	PRINT: Datei in die Warteschlange einfügen
01h	02h	PRINT: Datei aus Warteschlange entfernen
01h	03h	PRINT: Warteschlange löschen
01h	04h	PRINT: Ausgabe anhalten und Status abfragen
01h	05h	PRINT: Druckausgabe fortsetzen
01h	06h	PRINT: Drucker ermitteln
06h	00h	ASSIGN: Installations-Status abfragen
10h	00h	SHARE: Installations-Status abfragen
15h	00h	MSDEX: Anzahl der CD-ROM-Laufwerke ermitteln
15h	01h	MSDEX: Info über CD-ROM-Treiber einholen
15h	02h	MSDEX: Name des Copyright File ermitteln
15h	03h	MSDEX: Name des Abstract File ermitteln
15h	04h	MSDEX: Name des Bibliographic File ermitteln
15h	05h	MSDEX: VTOC lesen
15h	06h	MSDEX: reserviert
15h	07h	MSDEX: reserviert
15h	08h	MSDEX: Sektoren lesen
15h	09h	MSDEX: Sektoren schreiben
15h	0Ah	MSDEX: reserviert
15h	0Bh	MSDEX: CDROM Laufwerk abfragen
15h	0Ch	MSDEX: Versionsnummer abfragen
15h	0Dh	MSDEX: CDROM Laufwerke ermitteln
15h	0Eh	MSDEX: Volume Voreinstellung ermitteln
15h	0Eh/01	MSDEX: Volume Voreinstellung
15h	0Fh	MSDEX: Directory Eintrag lesen
15h	10h	MSDEX: Anfrage an Gerätetreiber senden
1Ah	00h	ANSI.SYS: Installations-Status abfragen
43h	00h	HIMEM.SYS: Installations-Status abfragen
43h	10h	HIMEM.SYS: Adresse zum Aufruf der XMS Funktionen ermitteln
48h	00h	DOSKEY: Installations-Status abfragen
48h	10h	DOSKEY: Eingabe des Anwenders entgegennehmen
4A11h	00h	DBLSPC: Versionsinformation einholen
4A11h	01h	DBLSPC: Abbildung eines Laufwerks abfragen
4A11h	02h	DBLSPC: Laufwerksbezeichnung tauschen
4A11h	03h	DBLSPC: Nur für internen Gebrauch
4A11h	04h	DBLSPC: Nur für internen Gebrauch
4A11h	05h	DBLSPC: Komprimiertes Laufwerk einbinden
4A11h	06h	DBLSPC: Deaktivieren eines Doublespace-Laufwerks
4A11h	07h	DBLSPC: Speicherplatz ermitteln
4A11h	08h	DBLSPC: Informationen über Fragm. der CVF-Dateien
4A11h	09h	DBLSPC: Max Anzahl der komp. Laufwerke abfragen
ADh	80h	KEYB.COM: Versionsnummer ermitteln
ADh	81h	KEYB.COM: Aktive Code-Page einstellen
B0h	00h	GRAFTABL: Installations-Status abfragen
B7h	00h	APPEND: Installations-Status abfragen
B7h	02h	APPEND: DOS 5-Kompatibilität abfragen
B7h	04h	APPEND: Liste der APPEND-Directories ermitteln
B7h	06h	APPEND: Operationsmodus abfragen
B7h	07h	APPEND: Operationsmodus einstellen

Anhang E: Die Funktionen des XMS

Der XMS-Standard, die Extended Memory Specification, wurde von Microsoft in Zusammenarbeit mit einigen anderen Firmen als Pendant zum EMS-Standard ins Leben gerufen, um den Zugriff auf den Extended Memory jenseits der 1-MByte-Grenze besser koordinieren zu können.

Allerdings werden die Funktionen des XMS-Standards im Gegensatz zu vielen anderen Funktionsschnittstellen nicht über einen speziellen Interrupt aufgerufen. Statt dessen erfolgt der Aufruf mit Hilfe eines FAR-CALL-Befehls, wobei zuvor die Adresse des XMS-Handlers ermittelt worden sein muß.

Hier nun eine Übersicht der XMS-Funktionen nach Einsatzgebiet:

Zugriff auf Extended Memory

Funktion	Beschreibung
08h	Größe des freien Extended Memory abfragen
09h	Allokiere einen Extended-Memory-Block (EMB)
0Ah	Freigabe eines allokierten Extended-Memory-Blocks (EMB)
0Bh	Kopiere Speicher
0Ch	Sperrt einen Extended-Memory-Block (EMB) gegen seine Verschiebung
0Dh	Gesperrten Extended-Memory-Block (EMB) wieder entsperren
0Eh	Informationen über einen EMB einholen
0Fh	Allokierten Extended-Memory-Block (EMB) vergrößern oder verkleinern

Zugriff auf Upper Memory Blocks

Funktion	Beschreibung
10h	Upper Memory Block (UMB) allokiieren
11h	Allokierten Upper Memory Block (UMB) wieder freigeben

Zugriff auf die HMA

Funktion	Beschreibung
01h	High-Memory-Area (HMA) in Besitz bringen
02h	HMA freigeben

Adreßleitung A20

Funktion	Beschreibung
03h	Globale Aktivierung der Adreßleitung A20
04h	Globale Schließung der Adreßleitung A20
05h	Lokale Freigabe der Adreßleitung A20
06h	Lokales Sperren der Adreßleitung A20
07h	Status der Adreßleitung A20 abfragen

Diverses

Funktion	Beschreibung
00h	XMS-Versionsnummer ermitteln

Anhang F: Die Funktionen des EMM

Den EMS-Standard regelt den Zugriff auf Speichererweiterungskarten, die nach dem Banking-Prinzip arbeiten und immer nur Zugriff auf einen kleinen Teil des gesamten Erweiterungsspeichers bieten. Die meisten im Umlauf befindlichen EMS-Karten unterstützen die Version 3.2 dieses Standards, d.h. die Funktionen, die unter den Versionen 3.0 und 3.2 definiert wurden. Nur wenige Speichererweiterungskarten unterstützen jedoch die EMS-Spezifikation 4.0 und die mit ihr verbundenen Funktionen. Die meisten kommerziellen EMS-Emulatoren, die EMS-Speicher durch Extended Memory simulieren, sind in der Regel auf die Version 4.0 abgestellt.

Hier nun eine Übersicht der Funktionen des EMS-Standards nach Funktionsnummern:

Funktion	Beschreibung	Version
40h	Status des EMM ermitteln	3.0
41h	Segmentadresse des Page-Frame ermitteln	3.0
42h	Anzahl der EMS-Pages ermitteln	3.0
43h	EMS-Speicher allokatieren	3.0
44h	Mapping setzen	3.0
45h	Pages freigeben	3.0
46h	EMM-Version ermitteln	3.0
47h	Seitentabelle sichern	3.0
48h	Mapping zurücksetzen	3.0
49h	Undokumentiert	3.0
4Ah	Undokumentiert	3.0
4Bh	Anzahl der Handles ermitteln	3.0
4Ch	Anzahl allokatierter Pages ermitteln	3.0
4Dh	Alle Handles ermitteln	3.0
4Eh/00h	Seitentabelle sichern	3.2
4Eh/01h	Seitentabelle zurücksetzen	3.2
4Eh/02h	Seitentabelle sichern und gleichzeitig zurücksetzen	3.2
4Eh/03h	Größe der Seitentabelle ermitteln	3.2
4Fh/00h	Einen Teil der Seitentabelle sichern	4.0
4Fh/01h	Eine partiell gesicherte Seitentabelle zurücksetzen	4.0
4Fh/02h	Größe des Puffers zur Aufnahme einer partiellen Seitentabelle ermitteln	4.0
50h/00h	Logische Seiten auf physikalische Seiten nach deren Nummer abbilden	4.0
50h/01h	Logische Seiten auf physikalische Seiten nach deren Adresse abbilden	4.0
51h	Anzahl der allokatierten logischen Seiten verändern	4.0
52h/00h	Attribut eines Handles ermitteln	4.0
52h/01h	Attribut eines Handles festlegen	4.0
52h/02h	Verfügbarkeit nicht-volatiler EMS-Seiten abfragen	4.0
53h/00h	Namen eines Handles ermitteln	4.0
53h/01h	Namen eines Handles festlegen	4.0
54h/00h	Namen aller Handles ermitteln	4.0
54h/01h	Handle mit einem bestimmten Namen ermitteln	4.0
55h/00h	Seiten abbilden und in Extended Memory springen	4.0
55h/01h	Seiten abbilden und in Extended Memory springen	4.0
56h/00h	Seiten abbilden und Routine im Extended Memory aufrufen	4.0
56h/01h	Seiten abbilden und Routine im Extended Memory aufrufen	4.0
56h/02h	Benötigten Stack-Speicher für die Unterfunktionen 00h und 01h ermitteln	4.0
57h/00h	Speicherbereich kopieren	4.0
57h/01h	Speicherbereiche austauschen	4.0
58h/00h	Adressen der physikalischen EMS-Seiten und ihre Nummern ermitteln	4.0
58h/01h	Anzahl der physikalischen EMS-Seiten ermitteln	4.0
59h/00h	Informationen über die EMS-Hardware liefern	4.0
59h/01h	Anzahl nichtstandardisierter EMS-Seiten ermitteln	4.0
5Ah/00h	EMS-Seiten allokatieren	4.0
5Ah/01h	Nichtstandardisierte EMS-Seiten allokatieren	4.0
5Bh	Dienste im Zusammenhang mit DMA-Transfers	4.0
5Ch	EMM auf einen Warmstart vorbereiten	4.0
5Dh	Dienste zur Abschottung des Betriebssystems	4.0

Anhang G: Die Hardware-Interrupts

Interrupt 0: Division durch null

Der 8088 verfügt auf Maschinensprache-Ebene über die beiden Befehle DIV und IDIV die eine Integer-Division durchführen. Nach den allgemeinen Rechenregeln ist dabei eine Division durch null nicht erlaubt und darf deshalb nicht ausgeführt werden. Aus diesem Grund löst der Prozessor in einem solchen Fall das Interrupt 0 aus.

Der zugehörige Vektor wird vom DOS bei dessen Initialisierung auf eine eigene Routine gelegt, so daß bei Aufruf dieses Interrupts die entsprechende DOS-Routine aufgerufen wird.

In den meisten DOS-Versionen gibt sie lediglich eine Bildschirrmeldung in der Form "Division durch null" aus und fährt dann in der Programmausführung mit dem Befehl fort, der auf den Divisionsbefehl folgt.

Interrupt 1: Einzelschritt

Dieses Interrupt wird von der CPU nach jeder Befehlsausführung aufgerufen, wenn sie sich nach dem Setzen des TRAP-Bits im Flag-Register im Einzelschrittmodus befindet.

Debugger nutzen diesen Modus, um die Programmausführung verfolgen zu können. Sie stellen deshalb einen Interrupt-Handler für dieses Interrupt zur Verfügung.

Damit nun aber nicht auch nach der Ausführung jedes einzelnen Befehls der Interrupt-Routine wiederum das Interrupt 1 aufgerufen wird, löscht der Prozessor das TRAP-Bit beim Eintritt in den Interrupt-Handler.

Zuvor hat er jedoch das gesamte Flag-Register und damit auch das TRAP-Bit auf dem Stack gespeichert. Wird die Interrupt-Routine durch den IRET-Befehl beendet, setzt er das TRAP-Bit automatisch wieder auf seinen alten Wert, indem er das gesamte Flag-Register wieder vom Stack holt. Dadurch wird nach der Ausführung des nächsten Befehls im zu verfolgenden Programm wieder ein Interrupt 1 aufgerufen.

Hat der Programmierer alle nötigen Informationen über das Programm erhalten, kann der TRAP-Modus (bzw. das TRAP-Bit) wieder abgeschaltet werden.

Die einzige Möglichkeit, das TRAP-Bit zu löschen, besteht deshalb darin, innerhalb der Interrupt-Routine das Flag-Register wieder vom Stack zu holen, das TRAP-Bit zu löschen und das gesamte Flag-Register dann wieder an seine ursprüngliche Position auf den Stack zu bringen. Wird die Interrupt-Routine dann durch den IRET-Befehl beendet, holt die CPU das Flag-Register wieder vom Stack. Da das TRAP-Bit nun aber nicht mehr gesetzt ist, erfolgen keine weiteren Aufrufe der Interrupt-Routine mehr, und das Programm kann ungestört weiterlaufen.

Interrupt 2: Das NMI

Dieses Interrupt wird als Non-Maskable-Interrupt (NMI) bezeichnet, weil es "nicht maskierbar" ist. Dies bedeutet, daß die Ausführung dieses Interrupts auch mit Hilfe des CLI-Befehls, der alle Interrupts unterdrückt, nicht verhindert werden kann.

Beim PC hat er die Aufgabe, auf Fehler im RAM aufmerksam zu machen, die vermuten lassen, daß einer der RAM-Bausteine defekt ist.

Beim Hochfahren des Systems legt das BIOS den zugehörigen Vektor auf eine eigene Routine, so daß bei Auftreten eines RAM-Fehlers die entsprechende BIOS-Routine aufgerufen wird, die eine Meldung auf dem Bildschirm ausgibt und das System anhält.

Interrupt 3: Breakpoint

Das Besondere an diesem Interrupt ist, daß es im Gegensatz zu allen Interrupts mit einem speziellen Maschinensprache-Befehl aufgerufen werden kann, der nur für ihn gilt. Während nämlich jedes Interrupt über einen 2 Byte langen Maschinensprache-Befehl (erstes Byte CDh, zweites Byte Nummer des Interrupts) aufgerufen werden kann, gibt es einen besonderen, nur ein Byte langen Befehl (Code CCh), mit dem das Interrupt 3 aufgerufen wird. Dadurch eignet sich dieses Interrupt gut für das Debuggen von Programmen, wo es interessant sein kann, ein Programm bis zu einer bestimmten Stelle ausführen zu lassen, um es dann zu unterbrechen und sich z.B. die aktuellen Registerinhalte anzeigen zu lassen.

Innerhalb eines Debuggers wird das dadurch realisiert, daß an die Stelle im Programm, an der es unterbrochen werden soll, der Aufruf des Interrupts 3 plaziert wird. Wird das Programm ausgeführt und gelangt der Prozessor an die entsprechende Stelle, ruft er das Interrupt 3 auf. Hinter diesem Interrupt verbirgt sich dann meistens ein Interrupt-Handler des Debuggers, der die aktuellen Registerinhalte und ähnliches anzeigt. Außerdem ersetzt diese Routine den Aufruf des Interrupts 3 wieder durch den Befehl, der zuvor seine Stelle eingenommen hat.

Bis zum Aufruf eines Testprogramms findet dieses Interrupt normalerweise keine Verwendung und wird von DOS deshalb auf eine Routine gelegt, die nur einen IRET- (Interrupt Return) Befehl enthält und damit sofort in das unterbrochene Programm zurückführt.

Interrupt 4: Überlauffehler

Dieses Interrupt kann durch einen bestimmten, an eine Kondition gebundenen Maschinensprache-Befehl aufgerufen werden. Es handelt sich dabei um den INTO- (Interrupt on Overflow) Maschinensprache-Befehl, der nur dann in dem Aufruf des Interrupts 4 resultiert, wenn bei seiner Ausführung das Overflow- (Überlauf) Bit im Flag-Register gesetzt ist.

Dies kann nach arithmetischen Operationen (z.B. der Multiplikation mit Hilfe des MUL-Befehls) der Fall sein, falls sich das Ergebnis dieser Operation nicht mehr mit einer bestimmten Anzahl von Bits darstellen läßt.

Natürlich kann dieses Interrupt auch mit Hilfe des normalen INT-Befehls aufgerufen werden, doch ist dieser nicht an die Kondition des gesetzten Overflow-Bits gebunden. Da dieses Interrupt kaum genutzt wird, legt ihn DOS auf einen IRET-Befehl.

Interrupt 5: Hardcopy

Das Interrupt 5 gehört zu den Interrupts, die durch das BIOS genutzt werden. Er wird vom BIOS immer dann aufgerufen, wenn [PrtSc] (bei einer amerikanischen Tastatur) oder [Druck] (bei einer deutschen Tastatur) betätigt wird.

Seine Aufgabe ist es dann, eine Abbildung des aktuellen Bildschirms, eine sogenannte Hardcopy, an den Drucker zu senden.

Aus diesem Grund wird der Vektor dieses Interrupts in der Vektortabelle auch vom BIOS initialisiert. Natürlich können auch Assemblerprogramme, oder Programme die in einer Hochsprache formuliert sind, dieses Interrupt durch Aufruf des INT-Befehls nutzen, um während ihrer Ausführung eine Hardcopy an den Drucker zu senden.

Interrupt 6, 7: Unbenutzt

Unbenutzt bleiben im Augenblick noch die Interrupts 6 und 7. Sie sind von IBM für eine spätere Verwendung vorgesehen und können im Augenblick noch vom Programmierer für eigene Zwecke benutzt werden.

Interrupt 8: Der Timer

Der Timer-Chip im PC (ein 8253) erhält pro Sekunde 1.193.180 Signale vom Herzen des Systems, einem oszillierenden Quartz.

Nach jeweils 65536 dieser Signale, also ca. 18,2mal (genau 18,20648193) in der Sekunde, erzeugt er daraufhin einen Aufruf des Interrupts 8, der durch den 8259 an die CPU weitergeleitet wird.

Da die Häufigkeit des Aufrufs dieses Interrupts unabhängig von der Taktfrequenz des Systems ist, eignet sich dieses Interrupt besonders gut zur Zeitmessung, da nach 18,2 Aufrufen eine Sekunde vergangen ist.

In diesem Sinne wird er deshalb auch innerhalb des PCs verwendet. BIOS legt den Interrupt-Vektor dieses Interrupts auf eine eigene Routine, die dadurch 18,2mal in der Sekunde aufgerufen wird. Ihre Aufgabe ist es, bei jedem Aufruf einen (Zeit-)Zähler zu erhöhen sowie den Diskettenmotor abzuschalten, falls für eine bestimmte Zeit kein Diskettenzugriff erfolgen soll.

Nachdem diese Aufgaben erledigt sind, ruft diese Routine das Interrupt 1Ch auf. Dieses kann vom Benutzer für Routinen belegt werden, die von einem kontinuierlichen Signal abhängig sind.

Interrupt 9: Die Tastatur

In der Tastatur befindet sich ein Intel-Prozessor mit der Bezeichnung 8048 (bzw. ein 8042 beim AT). Er überwacht die Tastatur und registriert, ob eine Taste niedergedrückt oder losgelassen bzw. für längere Zeit niedergedrückt wird.

In einem solchen Fall muß er das natürlich der CPU mitteilen, damit der Code der betätigten Taste in das System gelangen und weiterverarbeitet werden kann. Zunächst sendet er in einem solchen Fall ein Signal an den 8259, der, sofern gerade keine Interrupt-Anforderung mit einer höheren Priorität vorliegt, die CPU dazu veranlaßt, das Interrupt 9 aufzurufen. Dieses ruft eine BIOS-Routine auf, die das Zeichen von der Tastatur ausliest und es gegebenenfalls in den Tastaturpuffer aufnimmt.

Interrupts 10 bis 12: Je nach der angeschlossenen Hardware

Interrupt 13: Festplatte

Ist eine Festplatte an das System angeschlossen, wird das Interrupt 13 von ihr aufgerufen. Dies geschieht z.B. immer dann, wenn eine Lese- oder Schreiboperation beendet wurde und dies dem BIOS mitgeteilt werden soll.

Interrupt 14: Diskette

Dieses Interrupt wird vom Controller des bzw. der Diskettenlaufwerke immer dann mit Hilfe des 8259 aufgerufen, wenn er die Aufmerksamkeit der CPU benötigt.

Hinter diesem Interrupt verbirgt sich eine BIOS-Routine, die auf unterster Ebene mit dem Controller kommuniziert. Bei Aufruf dieses Interrupts übermittelt ihr der Controller bestimmte Statusinformationen, um dem BIOS z.B. mitzuteilen, daß eine Lese- oder Schreiboperation abgeschlossen wurde.

Interrupt 15: Drucker

Dieses Interrupt wird über das parallele Druckerkabel von einem angeschlossenen Drucker immer dann mit Hilfe des 8259 aufgerufen, wenn dieser die Aufmerksamkeit der CPU benötigt.

Bedingt durch den zweiten Interrupt-Controller des AT verfügt dieser PC auch über mehr Hardware-Interrupts als andere PCs. Durch diesen Interrupt-Controller sind die acht Interrupts 70h bis 77h aufrufbar, die bei den früheren PCs noch für die Benutzung durch Anwendungsprogramme freigegeben waren, nun aber nicht mehr genutzt werden dürfen. Wie auch bei den acht Geräten, die mit dem ersten Interrupt-Controller verbunden sind, gilt hier, daß das Gerät, das mit dem Bit 0 des Interrupt-Mask-Registers des zweiten Interrupt-Controllers verbunden ist, das Interrupt 70h auslöst. Das Gerät an Bit 1 ruft das Interrupt 71h auf, Bit 2 das Interrupt 72h usw.

Wirklich durch den Interrupt-Controller aufgerufen werden dabei allerdings nur die Interrupts 70h und 75h, weil nur mit den Bits 0 und 5 des Interrupt-Mask-Registers Geräte verbunden sind. Trotzdem sollten auch die Interrupt-Vektoren der Interrupts 71h bis 74h und 76h und 77h nicht "verbogen" werden.

Interrupt 70h: Echtzeituhr

Für die Echtzeituhr gibt es mehrere Gründe, ein laufendes Programm durch das Interrupt 70h zu unterbrechen. Dies kann zum einen das Erreichen einer Alarmzeit, die abgeschlossene Aktualisierung der Uhrzeit und des Datums oder einfach ein von einem Programm ausgelöster zyklischer Interrupt-Aufruf sein.

Das Interrupt wird normalerweise von einer BIOS-Routine bedient, die zunächst einmal den Interrupt-Grund herausfindet und dann entsprechend reagiert.

Interrupt 75h: Mathematischer Coprozessor

Über das Interrupt 75h teilt ein an das System angeschlossener mathematischer Coprozessor (80287) der CPU mit, daß er Ihre Aufmerksamkeit benötigt, z.B. weil ein Fehler aufgetreten ist.

Interrupt 76h: Festplatte des AT

Dieses Interrupt wird durch den Festplatten-Controller des ATs aufgerufen, wenn es einen Festplattenzugriff abgeschlossen hat.

Anhang H: Maustreiber-Funktionen

Von Anfang an hat Microsoft seine Maus durch eine Software-Schnittstelle unterstützt, die in Form des Gerätetreibers MOUSE.SYS bzw. des Programms MOUSE.COM implementiert wird. Aufgerufen werden die verschiedenen Funktionen der Maus-Schnittstelle über den Interrupt 33h, wobei als Funktionsnummer ein 16-Bit-Wert Verwendung findet, der im AX-Register übergeben werden muß.

Die vorliegende Version 8.0 des Maustreibers unterstützt insgesamt 53 verschiedene Funktionen mit den Funktionsnummern von 0000h bis 00034h. Allerdings sind die Funktionen 11h und 12h sowie 002Eh undokumentiert und werden nur innerhalb des Maustreibers eingesetzt.

Leider gibt Microsoft das Einführungsdatum der verschiedenen Maustreiber-Funktionen erst ab der Funktion 25h an, die mit der besagten Version 6.26 eingeführt wurde. Viele der vorangehenden Funktionen gibt es bereits seit der Version 1.0, doch wurden nicht wenige im weiteren Verlauf der Entwicklung zwischen der Version 1.0 und 6.26 hinzugefügt. Da genauere Informationen jedoch fehlen, wird im Rahmen dieser Referenz davon ausgegangen, daß diese Funktionen seit der Version 1.0 verfügbar sind. Auch wenn dies im Einzelfall nicht immer zutrifft, kann bei der Programmierung davon ausgegangen werden, daß diese Funktionen durch alle Maustreiber unterstützt werden, die sich im praktischen Einsatz befinden.

Die meisten Mausfunktionen arbeiten mit den Registern AX, BX, CX und DX, um Informationen vom Aufrufer entgegenzunehmen, und liefern ihm etwaige Funktionsergebnisse auch in diesen Registern zurück. Nur in Ausnahmefällen werden die Register ES, SI und DS für den Funktionsaufruf herangezogen, vornehmlich, wenn die Adresse von Puffern übergeben werden muß.

Grundsätzlich gilt, daß sich nach dem Funktionsaufruf nur der Inhalt der Register verändert hat, die für Rückgabe von Funktionsergebnissen verwendet werden. Der Inhalt aller anderen Register bleibt gegenüber dem Funktionsaufruf unverändert. Bei den meisten Funktionen wird allerdings das AX-Register für die Rückgabe eines Funktionsstatus verwandt, der im Fehlerfall ein Scheitern der jeweiligen Operation anzeigt.

Interaktion mit dem Maustreiber

Funktion	Beschreibung	Version
00h	Reset des Maustreibers	1.0
15h	Größe des Maus-Status-Puffers ermitteln	1.0
16h	Maus-Status sichern	1.0
17h	Maus-Status restaurieren	1.0
1Ch	Interrupt-Rate der Maus-Hardware einstellen	1.0
1Fh	Maustreiber deaktivieren	1.0
20h	Maustreiber aktivieren	1.0
21h	Reset des Maustreibers	1.0
24h	Maustyp ermitteln	1.0
25h	Allgemeine Informationen abfragen	6.26
26h	Ausdehnung des virtuellen Maus-Bildschirms abfragen	6.26
2Fh	Reset der Maus-Hardware	7.02
31h	Ausdehnung des virtuellen Bildschirms abfragen	7.05
32h	Unterstützte Funktionen ermitteln	7.05
33h	Einstellungen ermitteln	7.05

Mausabfrage

Funktion	Beschreibung	Version
03h	Mausposition und Status der Mausknöpfe ermitteln	1.0
05h	Anzahl der Betätigungen eines Mausknopfs ermitteln	1.0
06h	Wie oft wurde ein Mausknopf losgelassen?	1.0
0Bh	Bewegungswerte ermitteln	1.0

Cursorsteuerung

Funktion	Beschreibung	Version
01h	Maus-Cursor auf dem Bildschirm anzeigen	1.0
02h	Maus-Cursor ausblenden	1.0
04h	Bewege den Maus-Cursor	1.0
07h	Horizontalen Bewegungsbereich für Maus-Cursor festlegen	1.0
08h	Vertikalen Bewegungsbereich für Maus-Cursor festlegen	1.0
09h	Definiert den Maus-Cursor im Grafikmodus	1.0
0Ah	Definiert den Maus-Cursor im Textmodus	1.0
0Fh	Verhältnis zwischen Mickeys und Punkten setzen	1.0
10h	Ausschluß-Bereich definieren	1.0
13h	Schwelle für Verdopplung der Mausgeschwindigkeit setzen	1.0
1Ah	Maus-Sensitivität einstellen	1.0
1Bh	Maus-Sensitivität ermitteln	1.0
27h	Bit-Masken des Bildschirmcursors abfragen	7.01
2Ah	Informationen über Maus-Cursor abfragen	7.02
2Bh	Beschleunigungskurven einstellen	7.0
2Ch	Aktuelle Beschleunigungskurve auslesen	7.0
2Dh	Aktuelle Beschleunigungskurve einstellen/abfragen	7.0

Event-Handler

Funktion	Beschreibung	Version
0Ch	Event-Handler installieren	1.0
14h	Austauschen der Event-Handler	1.0
18h	Alternativen Event-Handler installieren	1.0
19h	Adresse eines alternativen Event-Handlers ermitteln	1.0

Bildschirmsteuerung

Funktion	Beschreibung	Version
1Dh	Bildschirmseite für Maus-Cursor setzen	1.0
1Eh	Bildschirmseite des Maus-Cursors ermitteln	1.0
28h	Videomodus setzen	7.0
29h	Liste der verfügbaren Video-Modi abfragen	7.0

Light-Pen

Funktion	Beschreibung	Version
0Dh	Emulation des Lichtgriffels anschalten	1.0
0Eh	Emulation des Lichtgriffels abschalten	1.0

Ballpoint-Mouse

Funktion	Beschreibung	Version
30h	Einstellungen der Ballpoint-Mouse setzen/abfragen	7.04

Windows

Funktion	Beschreibung	Version
34h	Lage der Datei MOUSE.INI ermitteln	8.0

Undokumentiert

Funktion	Beschreibung	Version
11h	Undokumentiert	1.0
12h	Undokumentiert	1.0
2Eh	Undokumentiert	6.26

Literaturhinweise

- Bernstein, Herbert; PC-Tuning: vom Standard-PC zum Personal-Super-Computer;
Markt & Technik Verlag AG; 1991
- Bradley, David J.; Programmieren in Assembler für die IBM Personal Computer;
Coedition der Verlage Carl Hanser Verlag München Wien und Prentice-Hall International INC. London;
1986
- Brors, Isa; Maschinsprache des IBM-PC / AT und Kompatibler in der Praxis; 2. Auflage;
Dr. Alfred Hüthig Verlag GmbH, Heidelberg 1988
- Dieterich, Ernst-Wolfgang; Turbo Assembler; 3. Auflage;
R. Oldenbourg Verlag, Wien München 1995
- Erdweg, Joachim; Assembler-Programmierung mit dem PC; 1. Auflage;
Friedr. Vieweg & Sohn Verlagsgesellschaft mbH, Braunschweig 1991
- Hanke, Gustav; Erste Einführung in die IBM-Assembler-Sprache; 3. Auflage;
R. Oldenbourg Verlag, Wien München 1985
- Kaier, W., Rudolf, E.; Turbo Assembler-Wegweiser;
Friedr. Vieweg & Sohn Verlagsgesellschaft mbH, Braunschweig 1990
- Monadjemi, Peter; PC-Programmierung in Maschinsprache; 2. Auflage;
Markt & Technik Verlag AG
- Scanlon, Leo J.; Die Assemblersprache des IBM-PC & XT; 4. Auflage;
Markt & Technik Verlag AG; 1986
- Tischer, Michael; PC intern 4 Systemprogrammierung; 1 Auflage
Data Becker GmbH & Co. KG Düsseldorf; 1994