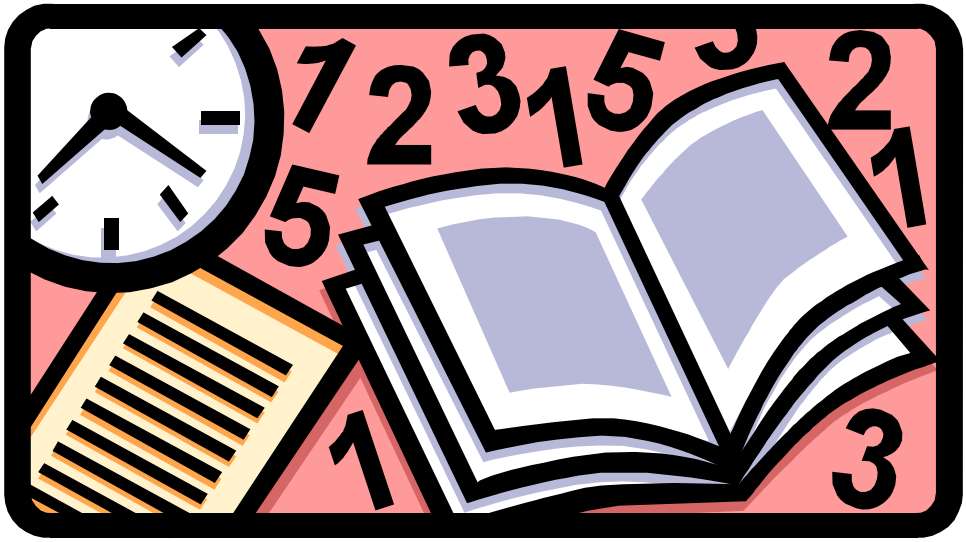
Assembler I - Grundlagen 2. Teil

Assembler – Adressierung und Speichermodelle



1. Adressierung

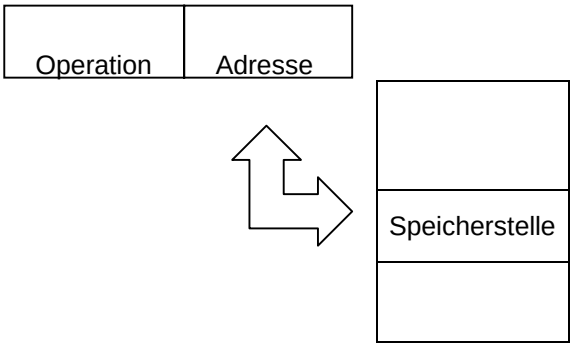
Im letzten Teil dieses Kapitels werden wir uns mit der *direkten Adressierung* und der *indirekten Adressierung* beschäftigen.

Ich denke ein anschauliches Beispiel verdeutlicht auch hier den Unterschied. Jeder hat diese Situation schon erlebt. Man hat es eilig und jemand quatscht einen auf der Strasse an: " Äh, wissen Sie wo das Kino ist ?".
Jetzt gibt es die Möglichkeit, dass man es weiß und dem Ratsuchenden sagt: " Ja, da gehen Sie jetzt da hoch und dann oben rechts. Dann stehen Sie genau davor." Das wäre die direkte Adressierung.

Wenn man es nicht weiß könnte man sagen: "Hm, ich weis nicht wo das Kino ist. Aber drei Straßen weiter auf der linken Seite ist ein Taxistand. Der kann Ihnen sagen wo Sie das Kino finden." Das wäre die indirekte Adressierung. Man hat einen Ort angegeben an dem man die eigentliche Adresse findet.

1.1 Direkte Adressierung

Bei diesem Speicherzugriff folgt dem Operationsteil die effektive Adresse.



Zunächst ist ein generelles Problem zu beachten:

Die Konventionen zur Adressierung werden zum Teil lasch gehandhabt. Das hat zur Folge, dass es in manchen Fällen zu mehrdeutigen oder missverständlichen Ausdrücken kommt.

Beispiel:

Es wird eine Variable VAR1 definiert und mit dem Wert 0 belegt.

```
VAR1    DW    0
```

Danach wird folgende Operation durchgeführt:

```
MOV AX, VAR1
```

Diese Anweisungen führen bei manchen Assemblern zu einer Warnmeldung. Es ist nicht ganz klar was gemeint ist. Man kann der Ansicht sein, dass die Zahl 0 in das Register AX zu kopieren ist. Jetzt wissen wir aber auch, dass der Ausdruck VAR1 genauso als Bezeichner für die Speicherstelle verwendet werden kann. Nehmen wir einmal an, dass besagte Speicherstelle im Segment die Offset – Adresse 50H hat. Dann könnte man auch der Ansicht sein, dass VAR1 ein Bezeichner ist der für die Zahl 50H steht.

Im ersten Fall würde in AX eine 0 stehen. Im zweiten Fall würden wir in AX die Zahl 50H vorfinden. Wie schon gesagt wollen uns manche Assembler diese Entscheidung nicht abnehmen und verlangen Eindeutigkeit.

1. Fall: Es soll die 0 nach AX kopiert werden:

MOV AX, [VAR1]

Hier steht VAR1 für die Adresse der Speicherstelle . Die eckigen Klammern geben nun an, dass der INHALT von dieser Speicherstelle nach AX zu kopieren ist.

2. Fall: Es soll die Offset – Adresse der Speicherstelle nach AX kopiert werden:

MOV AX, OFFSET VAR1

Hier wurde das AX Register direkt angesprochen. Es ist auch möglich den Inhalt einer bestimmten Speicherstelle direkt auszulesen. Zum Beispiel mit

MOV AX, [0040:0923]

Übrigens:
Man kann auch einen CALL direkt adressieren:
Einen NEAR CALL mit: CALL NEAR PTR SEGMENT:OFFSET
Einen FAR CALL mit: CALL FAR PTR SEGMENT:OFFSET

Beispiel:

```
DATEN1 SEGMENT

      MELDUNG1  DB   "HALLO WELT 1!",10,13,"$"

DATEN1 ENDS

DATEN2 SEGMENT

      MELDUNG2  DB   "HALLO WELT 2!",10,13,"$"

DATEN2 ENDS

STAPEL SEGMENT BYTE STACK

      DW 128 DUP(0)

STAPEL ENDS

CODE2  SEGMENT

      ASSUME CS:CODE2

      DRUCK PROC FAR

      MOV AH, 09H
      INT 21H

      RETF
      DRUCK ENDP

CODE2  ENDS

CODE1  SEGMENT

      ASSUME CS:CODE1,SS:STAPEL,ES:NOTHING

START:  MOV DX, OFFSET DATEN1:MELDUNG1
            MOV AX, DATEN1
            MOV DS, AX
            CALL FAR PTR CODE2:DRUCK

      MOV DX, OFFSET DATEN2:MELDUNG2
      MOV AX, DATEN2
      MOV DS, AX
      CALL FAR PTR CODE2:DRUCK

      MOV AH, 4CH
      INT 21H

CODE1   ENDS
      END START
```

Es ist nun (nach dem Assemblieren und Linken) möglich mit

C:\asm>DEBUG DIREKT.EXE

und anschließender Eingabe des Parameters t durch das Programm klicken.. äh ENTERn ;-)
(Wie schon gesagt, bei einem INT 21H - Befehl sollte mit einem p weiter geschaltet werden...)

Auf meinem Rechner ergibt sich unter anderem...

```
C:\asm>debug direkt.exe
-t
AX=0000  BX=0000  CX=014E  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=1B95  ES=1B95  SS=1BA7  CS=1BB8  IP=0003   NV UP EI PL NZ NA PO NC
1BB8:0003 B8A51B        MOV     AX,1BA5
-t
AX=1BA5  BX=0000  CX=014E  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=1B95  ES=1B95  SS=1BA7  CS=1BB8  IP=0006   NV UP EI PL NZ NA PO NC
1BB8:0006 8ED8          MOV     DS,AX
-t
AX=1BA5  BX=0000  CX=014E  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=1BA5  ES=1B95  SS=1BA7  CS=1BB8  IP=0008   NV UP EI PL NZ NA PO NC
1BB8:0008 9A0000B71B    CALL    1BB7:0000
-t
AX=1BA5  BX=0000  CX=014E  DX=0000  SP=00FC  BP=0000  SI=0000  DI=0000
DS=1BA5  ES=1B95  SS=1BA7  CS=1BB7  IP=0000   NV UP EI PL NZ NA PO NC
1BB7:0000 B409          MOV     AH,09
-t
```

Es ist zu erkennen, dass bei mir (bei Dir mit Sicherheit nicht) der Bezeichner CODE1 für die Segment – Adresse 1BB8 steht. Im Instruction Pointer steht immer schon die Offset – Adresse des nächsten Befehls.
Zunächst das Übliche: Die Adresse des Strings "HALLO WELT 1!" wird geladen. Der CALL – Befehl kommt uns auch nicht mehr ganz neu vor. Wir sehen, dass die Sprungadresse DIREKT angegeben ist. Was aber dem ungeübten Auge nicht auffällt ist, dass der Stack Pointer (Pfeile) um 4 Bytes (Speicherplätze) gewachsen ist (ja, gewachsen. Wie gesagt: Der Stack wächst von oben nach unten.).

Nun, da wir einen FAR – CALL haben springen wir zu einem Speicherplatz in ein anderes Segment mit einem anderen Offset. Dies bedeutet: Wir müssen zwei Adressen speichern. Denn der Prozessor will ja beim RETF – Befehl wissen wohin er zurückspringen soll !
Jede Adresse benötigt 2 Byte.

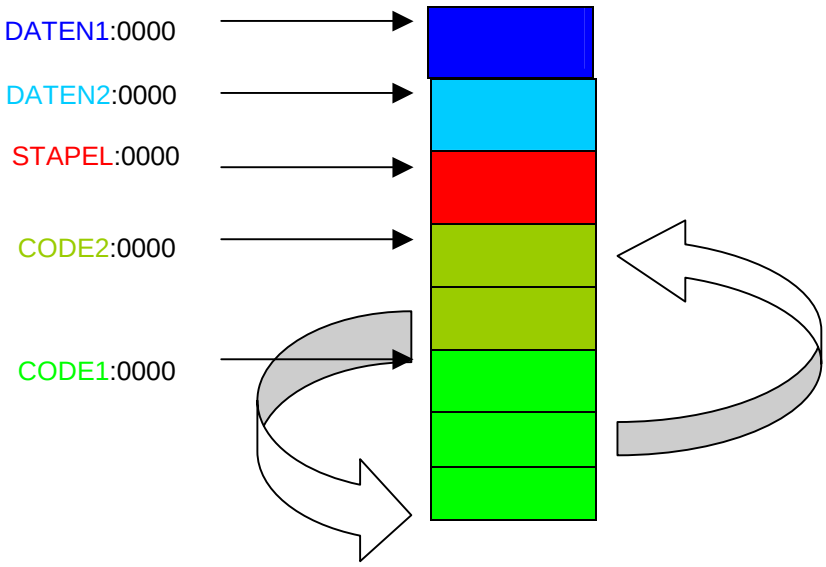
Mit der Segmentadresse 1BB7 befinden wir uns im Segment CODE2 . Hier ist das Unterprogramm, welches mit der Anweisung MOV AH, 09H beginnt.

```
AX=09A5  BX=0000  CX=014E  DX=0000  SP=00FC  BP=0000  SI=0000  DI=0000
DS=1BA5  ES=1B95  SS=1BA7  CS=1BB7  IP=0002   NV UP EI PL NZ NA PO NC
1BB7:0002 CD21          INT     21
-p
HALLO WELT 1!

AX=0924  BX=0000  CX=014E  DX=0000  SP=00FC  BP=0000  SI=0000  DI=0000
DS=1BA5  ES=1B95  SS=1BA7  CS=1BB7  IP=0004   NV UP EI PL NZ NA PO NC
1BB7:0004 CB            RETF
-t
AX=0924  BX=0000  CX=014E  DX=0000  SP=0100  BP=0000  SI=0000  DI=0000
DS=1BA5  ES=1B95  SS=1BA7  CS=1BB8  IP=000D   NV UP EI PL NZ NA PO NC
1BB8:000D BA0000        MOV     DX,0000
-t
```

Mit dem RETF endet das Unterprogramm. Es ist zu sehen, dass in das Segment CODE1 gesprungen wird. Hierzu wird die Rücksprungadresse benötigt. Die wird vom Stack genommen. Dadurch schrumpft der Stack wieder um 4 Bytes.

Wir haben hier als Beispiel den Prozessor ohne "Taxistand" *direkt* zum Unterprogramm geschickt.



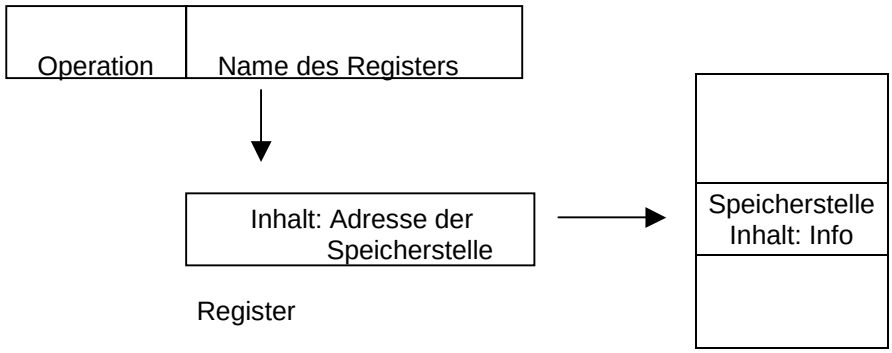
1.2 Indirekte Adressierung

Bei dieser Art der Adressierung kann man im wahrsten Sinne des Wortes alle Register ziehen. Auch hier gibt es zunächst ein paar kleine Gemeinheiten.

Es gibt viele "Orte" an denen man im PC Speicheradressen verstauen kann. Beim 8086 gab es bereits 17 Möglichkeiten der Adressierung.

1.2.1 Register – Adressierung

Hier wird mittels eines Registers adressiert. Dies bedeutet: Wir schauen in ein Register um dort eine Adressangabe zu finden. Danach suchen wir den Speicherplatz, der die angegebenen Adresse hat auf. In diesem Speicherplatz ist die eigentliche Information die wir suchen. Wir können hier aber auch Daten ablegen.



Beispiel:

MOV [BX], DX

Dies bedeutet: Schreibe den Inhalt des Registers DX an die Adresse im Arbeitsspeicher, die in BX angegeben ist.

MOV AX, [BX]

Dies bedeutet: Lies den Inhalt der Adresse, die in BX angegeben ist. Schiebe diesen Inhalt in das Register AX.

Jetzt die Fallen:

1. Falle:

Wir wollen die Zahl 10 an eine Speicherstelle schreiben deren Offset - Adresse (Segmentadresse liegt, sofern nicht durch ASSUME geändert, in DS vor) in BX angegeben ist.

Also: MOV [BX], 10

Der Assembler macht jetzt "Hä ??" und gibt eine Fehlermeldung aus. Warum ? Die Zahl 10 ist binär 1010. Uns ist klar: 1010 braucht 4 Bit. Da eine Speicherstelle 8 Bit hat, passt unsere Zahl 10 wunderbar in eine Speicherstelle rein. Manchen Assemblern ist das aber nicht klar ! Denn es könnte sich ja auch um den Ausdruck 0000 0000 0000 1010 (Word) handeln.

Wir müssen darauf hinweisen, dass wir 0000 1010 meinen:

MOV [BYTE PTR BX], 10

Manche werden sich nun fragen: Warum funktioniert dann MOV [BX], DX . Ja, hier haben wir es nur mit Registern zu tun. Und dass diese beiden Register jeweils eine Breite von 16 Bit haben rafft der Assembler dann doch.

2. Falle:

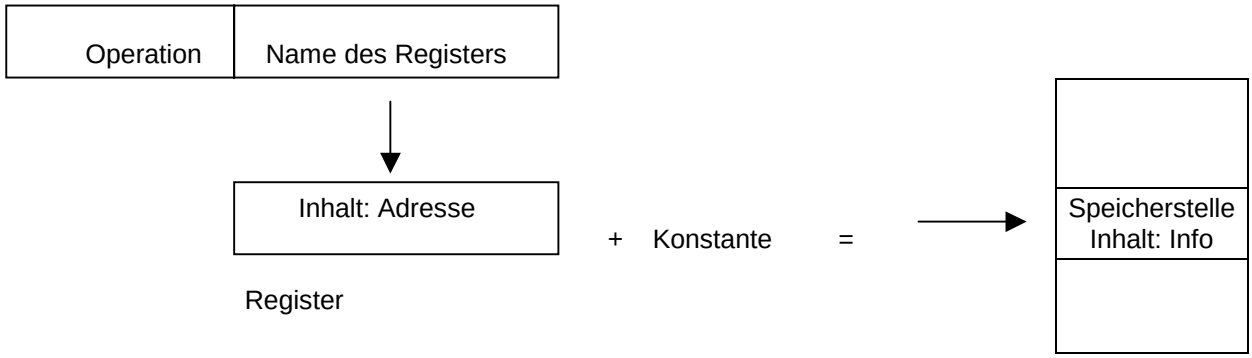
Hier tappen viele Anfänger rein:

MOV DL, BX

Wir schreiben Daten mit einer Länge von 16 Bit an einen Ort, der nur 8 Bit groß ist. Oh, Oh.....

1.2.2 Basis-relative – Adressierung

Hier wird zu einer in einem Register gespeicherten Adresse eine Zahl addiert.
Die Summe aus Adresse und Konstante ergibt die Adresse der gesuchten Speicherstelle.



MOV [BX + 10H], DX

Die Adresse der Speicherstelle an der der Inhalt des Registers DX gespeichert wird ist wie folgt bestimmt: Man nimmt den Inhalt des Registers BX als Adresse und addiert die hexadezimale Zahl 10H zu dieser Adresse hinzu. Damit erhält man die Adresse der gesuchten Speicherstelle.

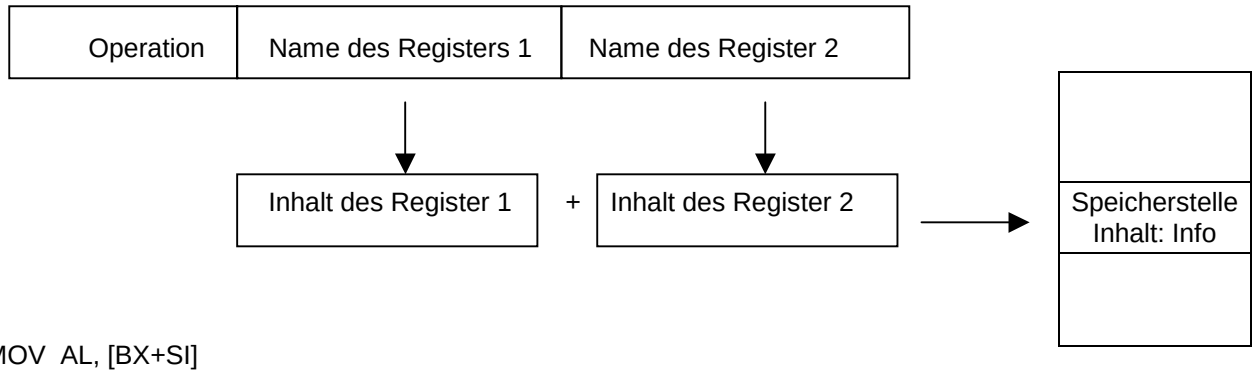
Dieser Ausdruck kann auch auf andere Weise dargestellt werden:

MOV [BX]10H, DX

Diese Befehlsform ist gleichbedeutend wie die obere Schreibweise.

1.2.3 Basis-indizierte Adressierung

Hier muss der Inhalt von Register1 und Register2 addiert werden um die Adresse der gesuchten Speicherstelle zu erhalten .



MOV AL, [BX+SI]

Beim 8086 sind folgende Kombinationen erlaubt:

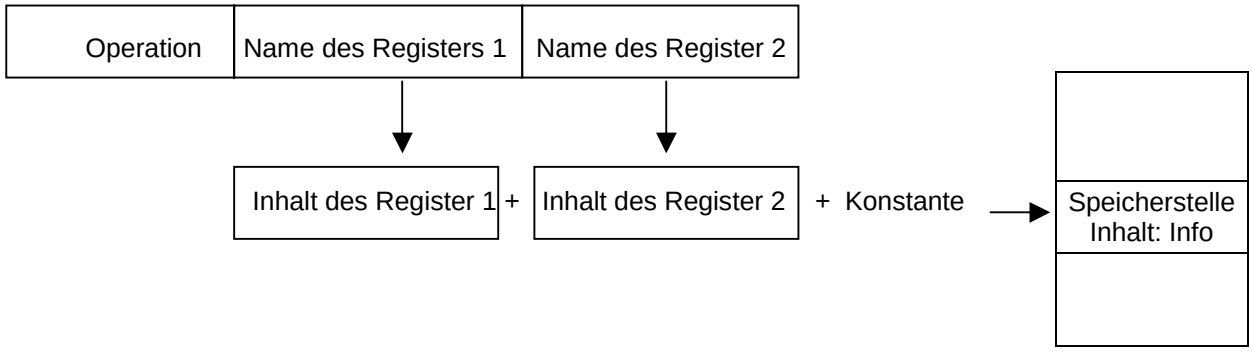
BX+SI
BX+DI
BP+SI
BP+DI

Ab dem 80386 sind auch alle Kombinationen der 32 – Bit Register (außer ESP) erlaubt.

1.2.4 Basis-indizierte Adressierung mit Displacement

Die Adressbildung ist hier folgende:

Es wird der Inhalt des ersten Registers zum Inhalt des zweiten Registers und dazu noch eine Konstante addiert. Das Ergebnis ergibt die Adresse der gewünschten Speicherstelle.



MOV [BX+SI+80H], DX

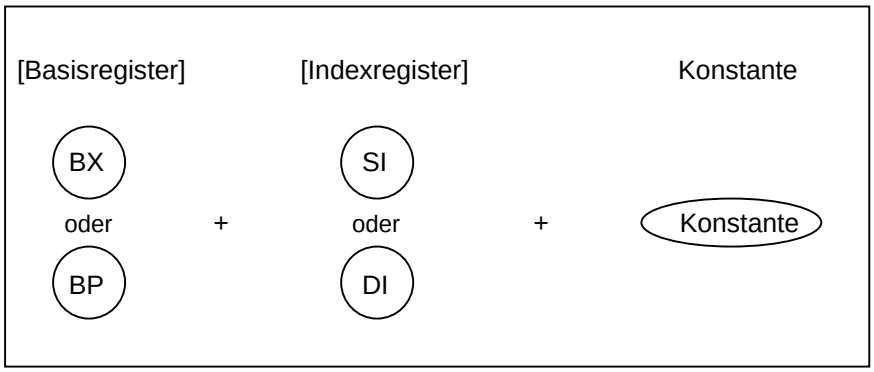
MOV LISTE[BX][DI], AL Liste ist eine Variable

1.3 Zusammenfassung zur Adressierung

Man erkennt, dass alle bisher behandelten Adressierungsarten Spezialfälle der basis-indizierten Adressierung sind.

Da unsere Programme auch auf jeder alten Krücke laufen sollen, werde ich hier nur die 8086 Konventionen darstellen:

Mit folgendem Schema kann man sich eine Eselsbrücke bauen:



Somit ergeben sich folgende gültigen Kombinationen:

[BX]
[SI]
[BX+SI]
[BX+DI]
[BX+Konstante]
[BX+SI+Konstante]
[BX+DI+Konstante]
[SI+Konstante]
[Konstante]

[BP]
[DI]
[BP+SI]
[BP+DI]
[BP+Konstante]
[BP+SI+Konstante]
[BP+DI+Konstante]
[DI+Konstante]

2. Speichermodelle

Alle Programme bisher wurden mit Hilfe der direkten Segmentierung erstellt. Jedes Segment wurde mit der Anweisung:

```
NAME SEMGMENT


NAME ENDS
```

erzeugt.

Darüber hinaus mussten von uns auch immer mittels der Zeile

```
ASSUME CS:CODE, DS:DATEN,ES:NOTHING,SS:STAPEL usw. …..
```

die Segmentregister geladen werden.

Zusätzlich war noch das Daten-Segment zu initialisieren :

```
MOV AX, DATEN
MOV DS, AX
```

Nicht zu vergessen ist auch die Startmarke und Endmarke

```
START:



        END START
```

Wichtig war auch, dass am Ende des Programms immer das Betriebssystem die Kontrolle erhält:

```
MOV AH, 4CH
INT 21H
```

Diese Einzelschritte können "automatisiert" werden, da sie bei jedem Programm auftreten.

Zuerst wenden wir uns der Segmentierung durch den Assembler zu:
Mit der direkten Segmentierung konnten wir die Abfolge der Segmente, die Anzahl der Segmente sowie ihre Größe beeinflussen.
Dies kann auch der Assembler für uns machen. Die Auswahl ist dabei aber auf einige Typen beschränkt. Diese Typen nennt man Modelle.

Im Angebot sind die *Typen* :

TINY	Programm-Code und Daten müssen in **ein** 64 KByte – Segment passen ! Code und Daten sind NEAR.
SMALL	Programm-Code und Daten müssen in **je** ein 64 KByte – Segment passen ! Code und Daten sind NEAR.
MEDIUM	Programm-Code kann größer als 64 KByte sein; Daten müssen in ein 64 KByte-Segment passen. Code ist FAR, Daten sind NEAR.
COMPACT	Programm-Code muss in ein 64 KByte-Segment passen; Daten können größer als 64 KByte sein, aber kein Datenbereich darf für sich allein größer als 64 KByte sein. Daten und Code sind FAR.
LARGE	Programm-Code und Daten können beide größer als 64 KByte sein, aber kein Datenbereich darf für sich allein größer als 64 KByte sein. Daten und Code sind FAR.
HUGE	Programm-Code und Daten können beide größer als 64 KByte sein, auch einzelne Datenbereiche dürfen größer als 64 KByte sein. Daten und Code sind FAR.

Ein solches "Grundgerüst" wird durch dir Direktive

.MODEL *Typ*

ausgewählt. Bei soviel Auswahl hat man die Qual der Wahl. Auch hier gibt es eine einfache Faustregel:
Verwende ein möglichst einfaches (weit oben stehendes) Modell.

Der entstehende Maschinencode ist somit schneller und auch einfacher zu analysieren.

Trotzdem müssen wir nicht jegliche Möglichkeit zur Einflussnahme aus der Hand geben. Wir können dem Assembler auch bei Modellen sagen, welcher Programmteil in welchen Segmenten abgelegt werden soll. Es gibt hier die Direktiven:

.STACK *Zahl*

Mit dieser Direktive wird der STACK angelegt. Diese Direktive ersetzt sozusagen den Ausdruck:

```
STAPEL SEGMENT BYTE STACK
      DW 128 DUP(0)
STAPEL ENDS
```

Mit *Zahl* kann man den zu reservierenden Speicher festlegen.

.DATA

Damit wird das Daten - Segment angelegt.

.CODE [*Name*]

Damit wird das Code – Segment angelegt. Bei den Modellen MEDIUM, LARGE und HUGE können in einem Assembler – Programm mehrere Code – Segmente vorkommen. Um diese unterscheiden zu können, müssen sie mit einem Namen versehen werden.

Damit wirklich kein Zweifel über den Programmanfang besteht, wird nach der .CODE Anweisung noch die Anweisung

.STARTUP

gesetzt.

Diese kleine Anweisung ersetzt :

```
ASSUME CS:CODE,DS:DATEN,ES:NOTHING, SS:STAPEL

        MOV DX, DATEN
        MOV DS, DX

START:
```

Wichtig ist natürlich auch der Programmabschluss.

Mit

.EXIT

werden die Befehle

```
MOV AH, 4CH
INT 21H
```

ersetzt. Zum Schluss nicht das

END

vergessen. Dies entspricht dem

```
        END START
```

Damit würde unser Programm HALLO.ASM nun so aussehen:

```
.MODEL SMALL

.DATA

    MELDUNG DB "Hallo Welt!","$"

.STACK

.CODE

.STARTUP

    MOV DX, OFFSET MELDUNG
    MOV AH, 09H
    INT 21H

.EXIT

END
```

Achtung: Die Segmentierung über Modelle ist immer eine Quelle von endlosen Problemen und viel Ärger ! Die Programmierung über Modelle wird nicht von allen Versionsnummern von TASM unterstützt. Auch sind die Konventionen bei MASM (von Microsoft) anders. Das Beispielprogramm oben wurde mit

Turbo Assembler Version 4.0 Copyright (c) 1988, 1993 Borland International

assembliert.

Dies hier soll lediglich eine Einführung sein, die erste Gehversuche ermöglichen soll.

Noch ein Tipp: Es gibt den Borland Turbo – Assembler 4.0 auch auf CD. Inbegriffen ist da ein Online – Handbuch und Anschauungsprogramme. Zu beziehen über den Buchhandel. Verlag: Franzis