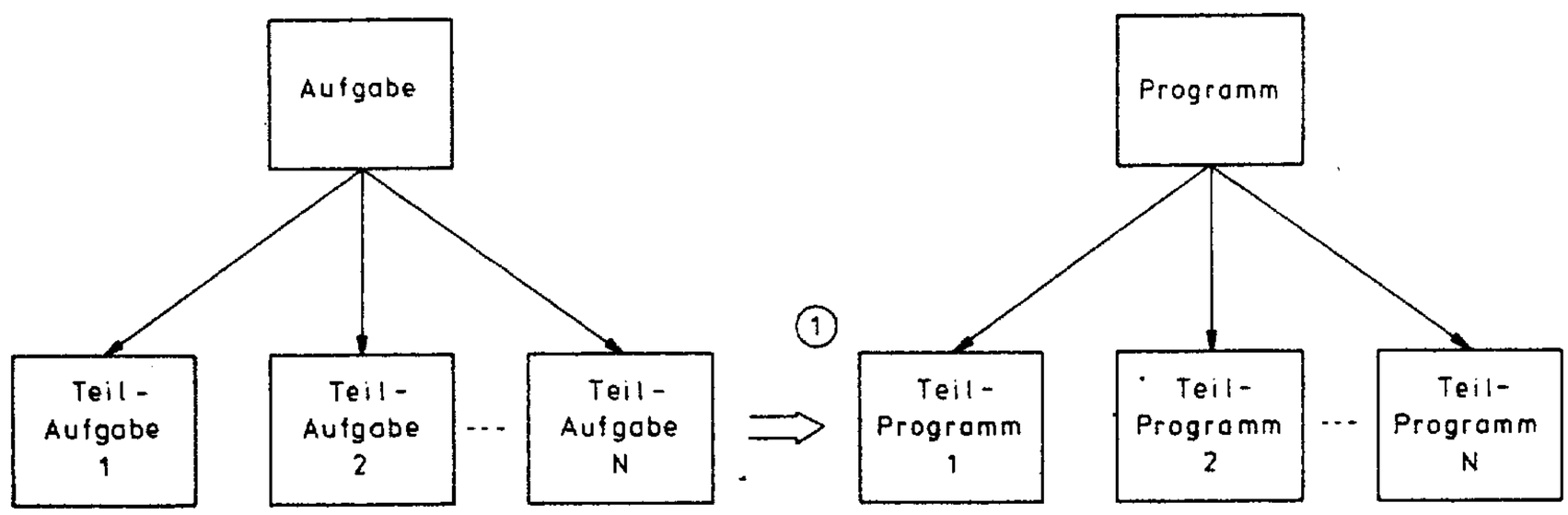
Kapitel 5

Modulare Programmierung

(für MASM-, ASM86- und ASM286-Programmierer)

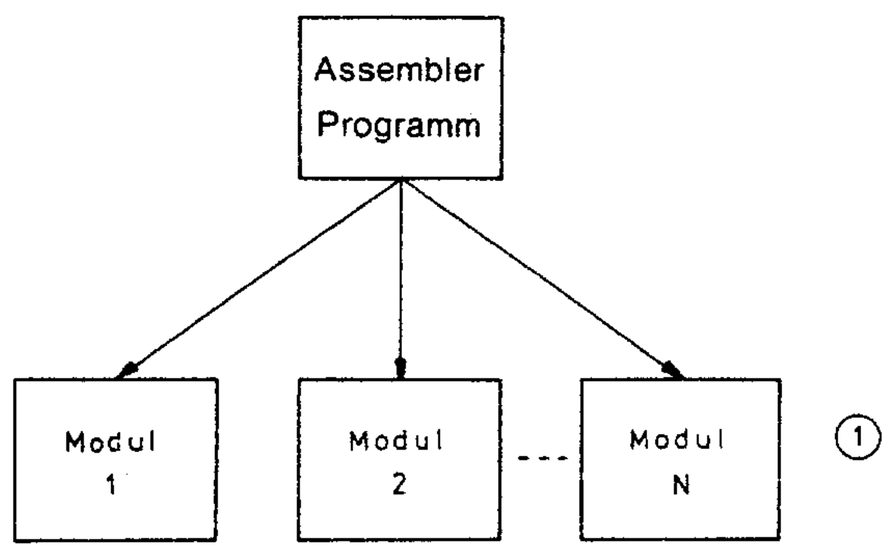
Jedes Assembler-Programm erledigt eine bestimmte Aufgabe (Task). Da im Normalfall die Aufgabe komplex und umfangreich ist, wird man sie in Teilaufgaben zerlegen. Jede Teilaufgabe kann nun von einem Teilprogramm erledigt werden, das als abgeschlossener funktionaler Block innerhalb des gesamten Programms betrachtet werden kann ①.



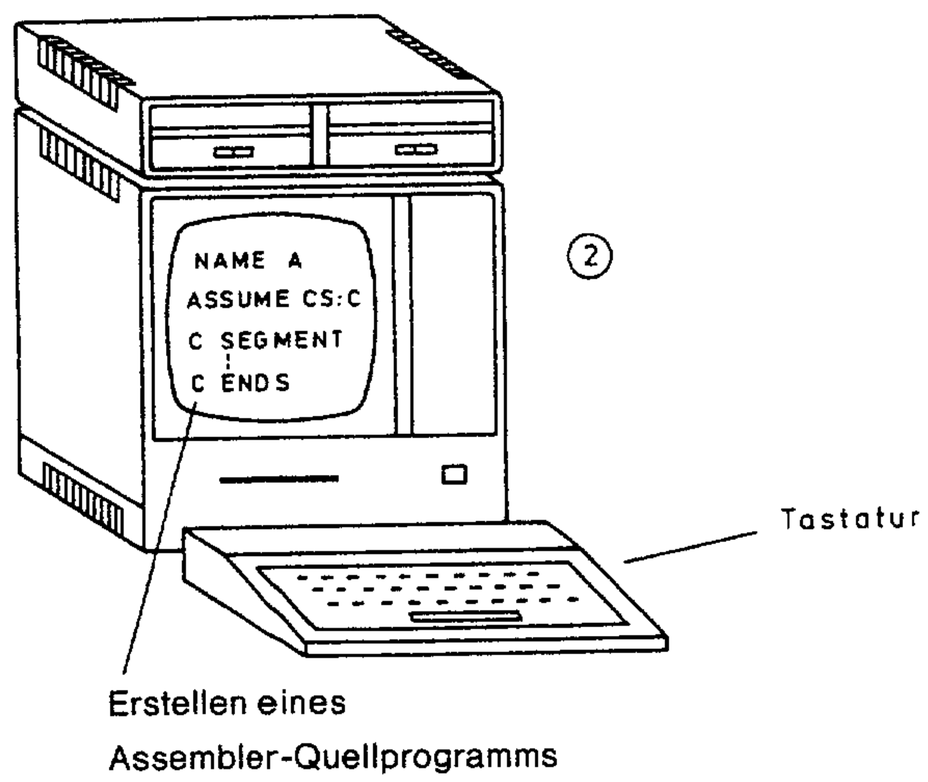
Dieses Verfahren hat folgende Vorteile:

- Die kleineren Teilprogramme sind bequemer zu warten. Es existiert weniger Text und Symbole können leichter gefunden werden.
- Jedes Teilprogramm kann selbständig als Funktionseinheit entwickelt und getestet werden.
- Auf Variablen und Labels, die in einem Teilprogramm definiert sind, kann von anderen Teilprogrammen zugegriffen werden.
- Es ist möglich einige Teilprogramme in höheren Programmiersprachen wie PL/M, PASCAL, FORTRAN oder C zu schreiben und andere in Assembler.

Jedes Assembler-Programm ist in einem sogenannten **Modul** gespeichert, für das der Anwenderprogrammierer einen Namen vergibt (Modulname). Soll nun ein Assembler-Programm in Teilprogramme zerlegt werden, wird man es auf mehrere Module verteilen ①



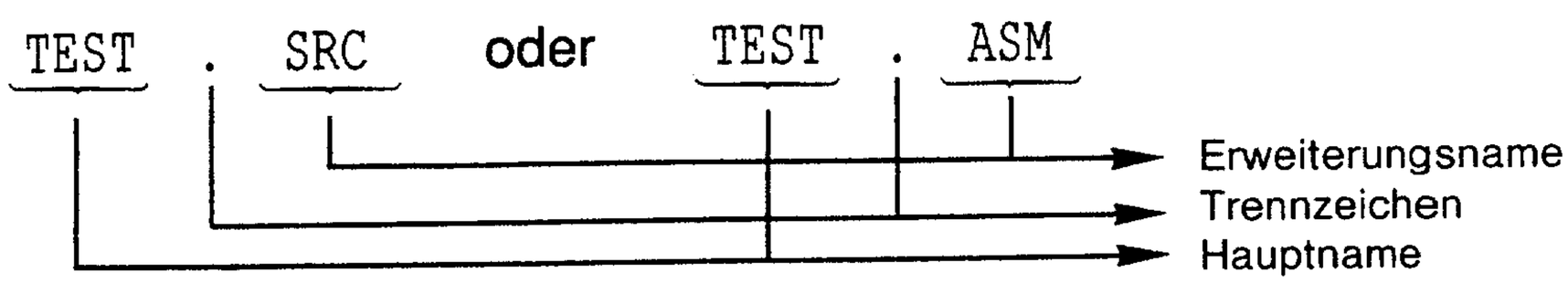
Jedes Modul enthält ein sogenanntes **Quellprogramm**, das zunächst mit Hilfe eines Texteditors (z.B. AEDIT.EXE von INTEL) auf einem Personal-Computer (z.B. IBM-AT) unter MS-DOS erstellt wird ②.



Hinweis: Wird ein Personal-Computer benützt, um mit INTEL-Werkzeugen wie ASM86.EXE, LINK86.EXE (Linker), LOC86.EXE (Locater), C86.EXE und entsprechenden Emulatoren Software für Zielmaschinen (z.B. Roboter-Steuerungen) zu entwickeln, wird es als **M**ikrocomputer-**E**ntwicklungs-**S**ystem (MES) bezeichnet. Wenn die Software für Zielmaschinen in PROMs untergebracht ist, wird sie als Firmware (Software in Silicon) bezeichnet.

Ist der Editiervorgang, die sogenannte Aufbereitungssitzung, beendet, wird das erstellte Quellprogramm in einer **Datei** (File) gespeichert. Die Datei richtet sich der Anwenderprogrammierer z.B. auf einer Floppy Disk selbst ein und kennzeichnet sie mit einem zweiteiligen Namen.

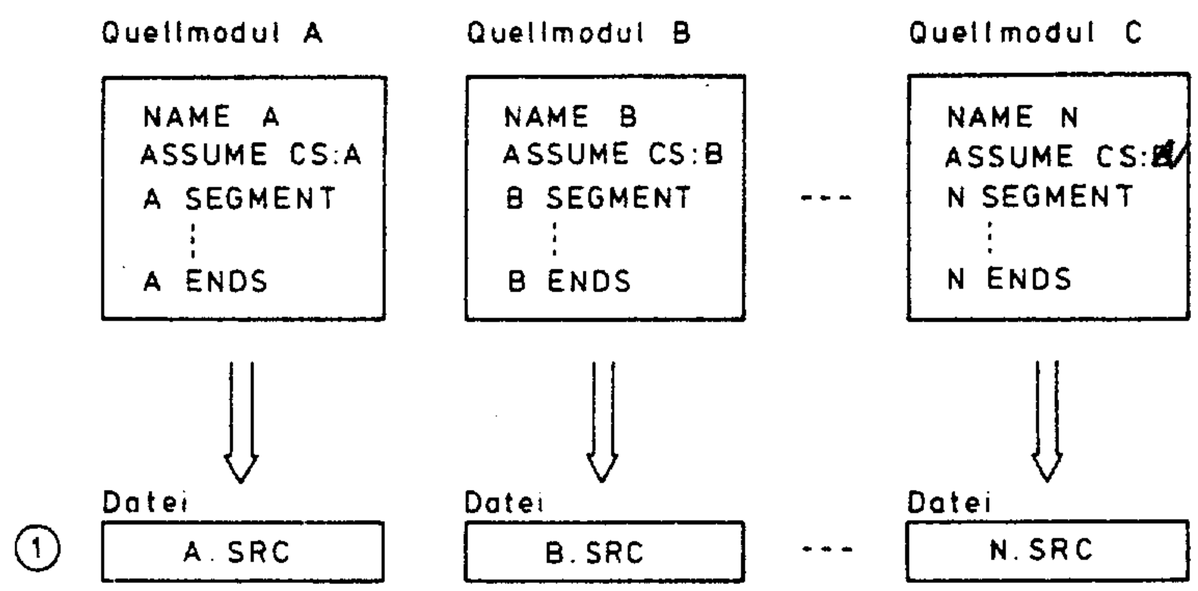
Beispiel für einen Dateinamen:



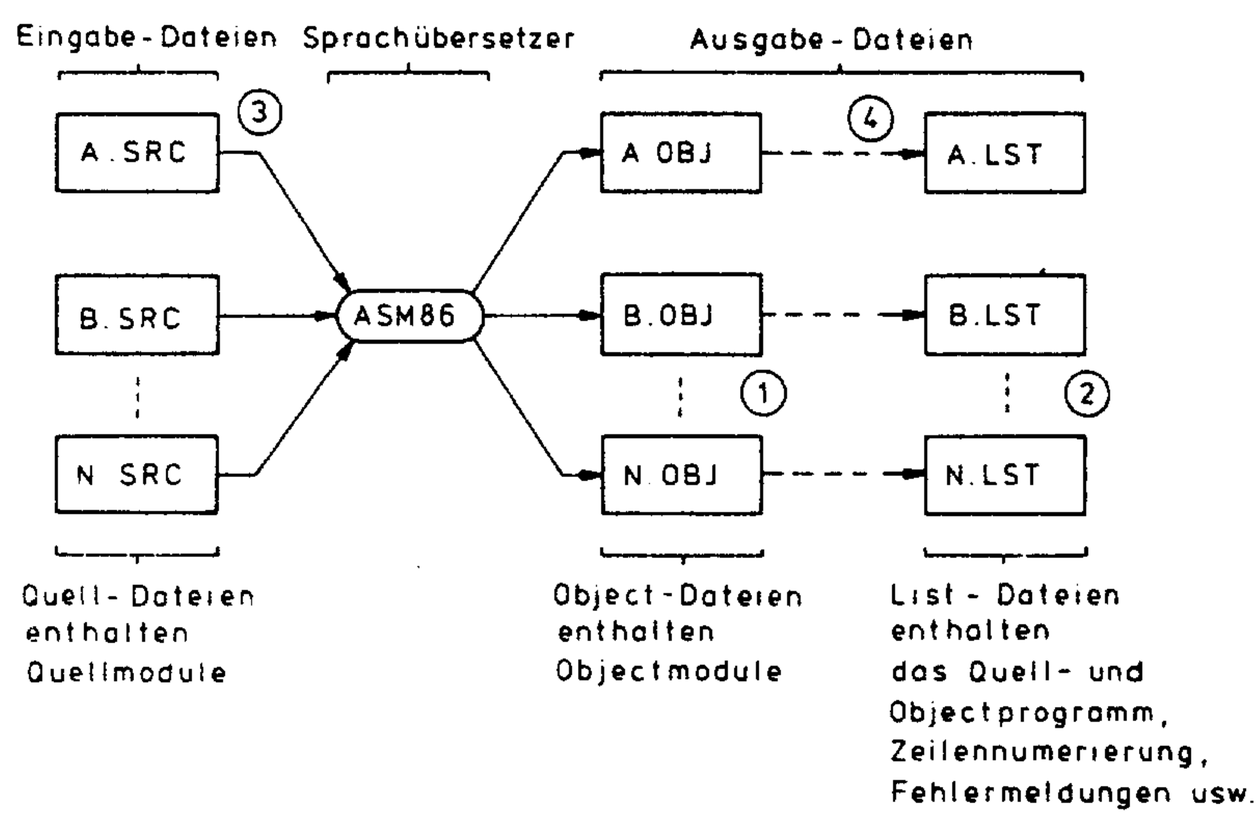
- Den Hauptnamen der Datei kann der Anwenderprogrammierer wählen.
- Das Trennzeichen (.) zwischen Hauptnamen und Erweiterungsnamen muß angegeben werden.

- Der Erweiterungsname (Extension) muß angegeben werden. Er bezeichnet den **Typ** des in der Datei gespeicherten **Datenbestands**. So deutet SRC = Soure oder ASM = Assembler darauf hin, daß in der Datei ein Modul gespeichert ist, dessen Daten einem Assembler-Quellprogramm entsprechen. Man sagt: Die Datei enthält ein **Quellmodul**.

Werden n Quellmodule entwickelt, bedeutet dies, daß n Dateien vom Typ SRC existieren, in denen die Quellmodule gespeichert sind ①.



Da die Quellmodule ASCII-Code enthalten, also nicht ablauffähig sind, müssen sie einem Sprachübersetzer (Assembler) zugeführt werden. Der Assembler generiert aus dem mnemonischen Code eines Quellmodules einen ablauffähigen Maschinencode. D.h. ein mnemonisch codiertes Quellprogramm steht nun in seiner binären Form als sogenanntes **Object-Programm** zur Verfügung. Der Assembler übernimmt den Namen des Quellmoduls, den der Anwenderprogrammierer gewählt hat und kennzeichnet damit das Object-Programm als **Object-Modul**.

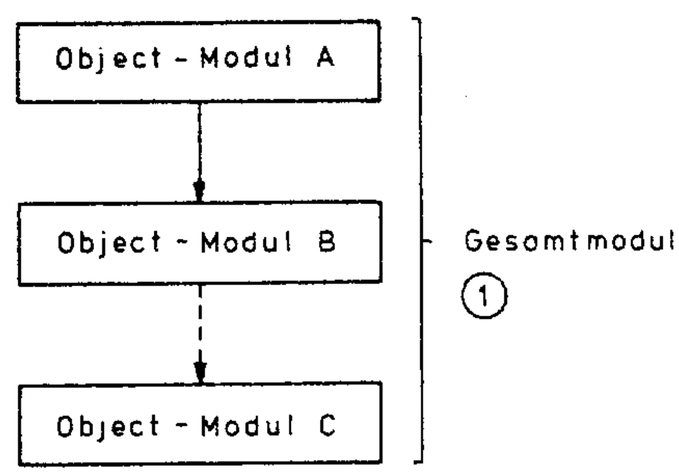


Die Object-Module speichert der Assembler in den sogenannten Object-Dateien. Zur Kennzeichnung dieser Dateien übernimmt er die Hauptnamen der Quelldateien und spezifiziert ihre Datenbestände mit dem Zusatznamen OBJ (Object) ①.

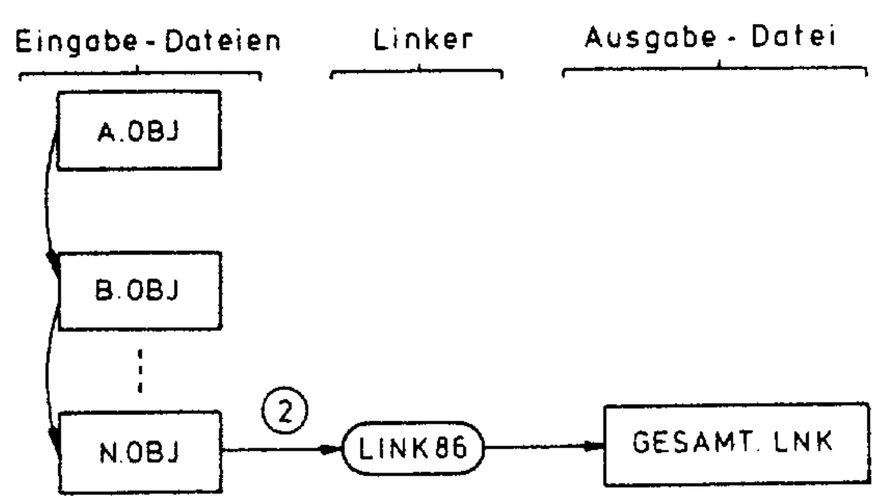
Neben den Object-Dateien generiert der Assembler sogenannte List-Dateien, die als Protokolle der Sprachübersetzungen betrachtet werden können. Sie dienen als Arbeitsmittel beim Programmtest. In ihnen sind Quell- und Objectprogramm, Meldungen über syntaktische Fehler und alle verwendeten Symbole gespeichert. Der Assembler kennzeichnet diese Dateien mit dem Zusatznamen LST ②.

Für den Assembler sind Quelldateien (Typ SRC oder ASM) Eingabe-Daten ③, während Objectdateien (Typ OBJ) und Listdateien (Typ LST) von ihm als Ausgabe-Dateien angelegt werden ④.

Nachdem zur Erledigung einer Aufgabe das Programm auf mehrere Quellmodule verteilt worden ist, werden diese einzeln assembliert, bzw. compiliert. Danach existieren mehrere Object-Dateien, in denen die übersetzten Quellmodule als Object-Module gespeichert sind. Es besteht nun die Notwendigkeit, die einzelnen Objectmodule zu einem Gesamtmodul zusammenzufügen ①.



Diese Aufgabe übernimmt der **Linker** (z.B. der INTEL-Linker LINK86.EXE für **8086/80186-Module**).



Hinweis: Für **80286-Module** übernimmt diese Aufgabe der **Binder** BND286.EXE oder der **System Builder** BLD286.EXE.

Damit LINK86 mehrere Objectmodule kombinieren kann, werden ihm Objectdateien (Typ OBJ) in einer bestimmten Reihenfolge zugeführt ②.

Beispiel eines Linkeraufrufs unter MS-DOS:

```
>LINK86   A.OBJ,B.OBJ,N.OBJ  TO  GESAMT.LNK   ④
```

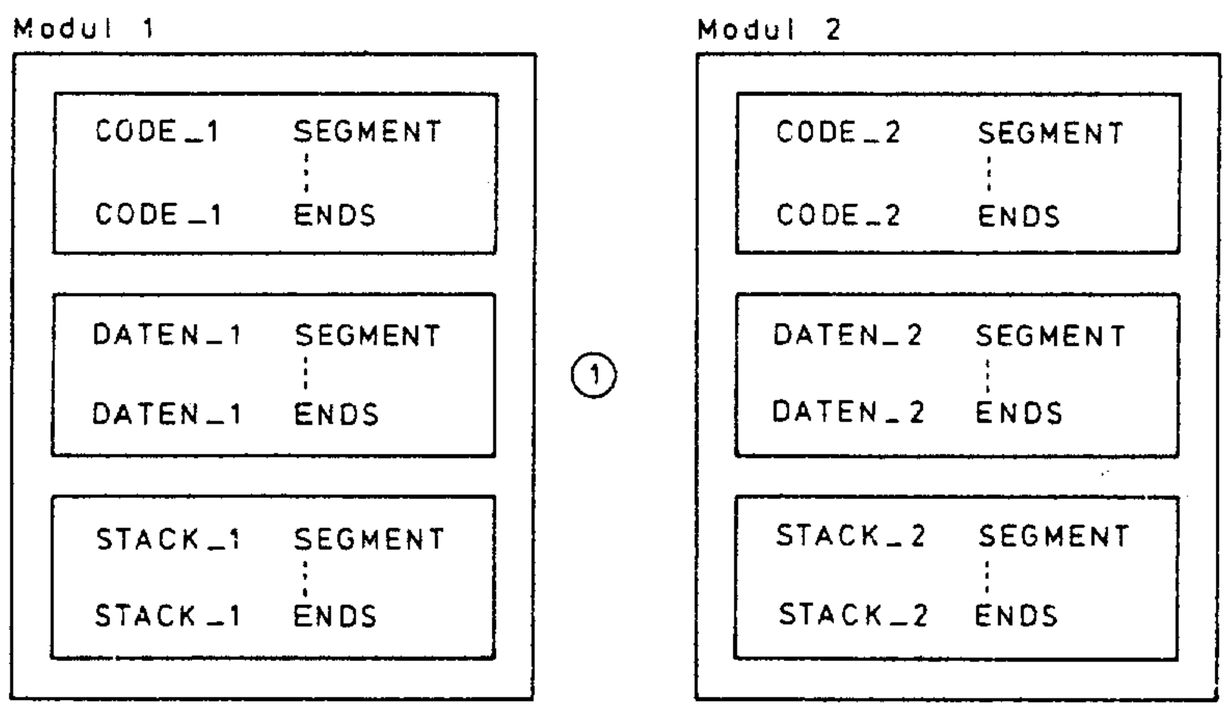
③ → Eingabedateien

④ → Ausgabedatei

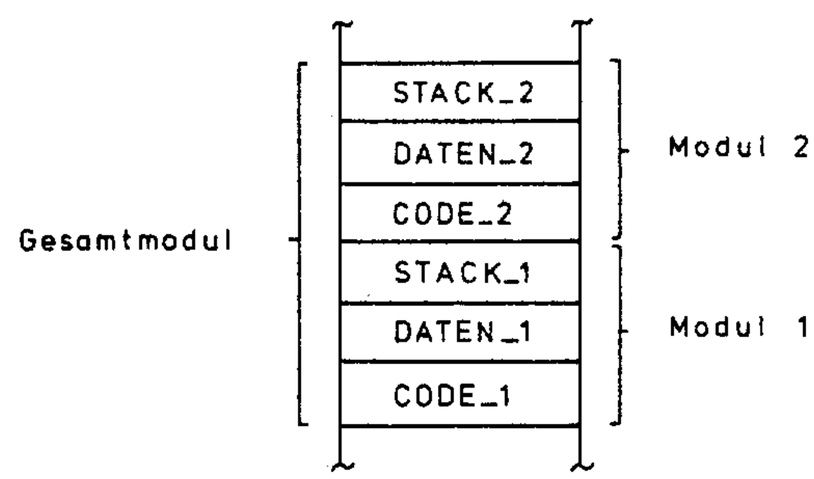
LINK86 bindet die in den Objectdateien gespeicherten Objectmodule in der angegebenen Reihenfolge ③ und hinterlegt den Gesamtmodul ebenfalls in einer Objectdatei. Diese Datei entspricht der Ausgabedatei des Linkers ④. Sie bekommt vom Anwenderprogrammierer den Zusatznamen LNK (Link). Der Hauptname ist frei wählbar.

5.1 Kombinieren logischer Segmente

Es sei vorausgesetzt, daß ein Programm auf zwei Module aufgeteilt worden ist. Jeder Modul enthalte drei logische Segment für die Code-, Daten- und Stack-Regionen. Diese Segmente sollen für Modul 1 CODE_1, DATEN_1, STACK_1 und für Modul 2 CODE_2, DATEN_2, STACK_2 heißen ①.



Werden die beiden Objectmodule mit dem Linker gebunden, ergibt sich für den Gesamtmodul im Speicher folgendes Bild.



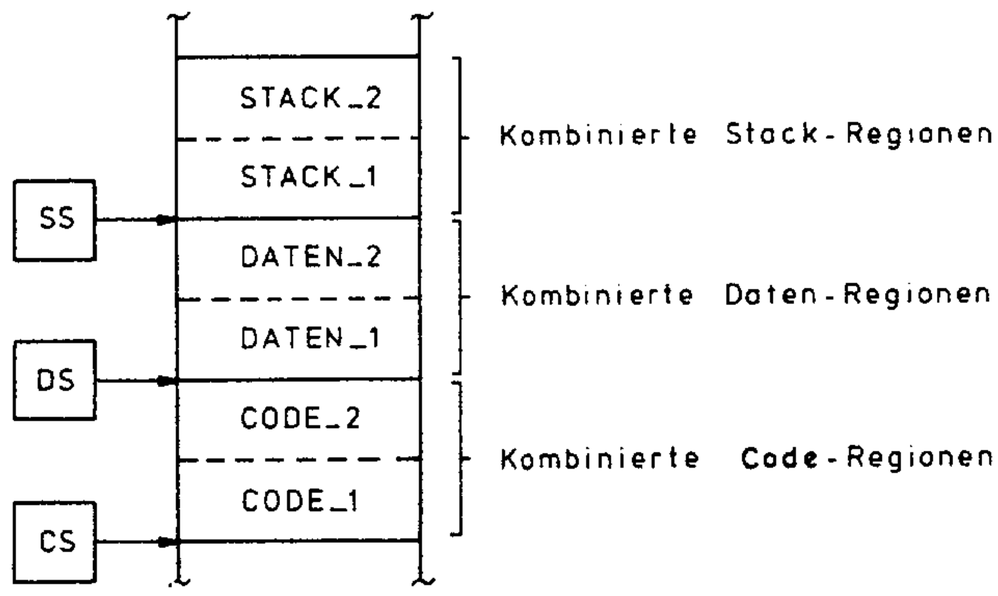
Es veranschaulicht, daß logisch zusammengehörende Segmente aus Modul 1 und Modul 2, wie CODE_1/CODE_2, DATEN_1/DATEN_2 und STACK_1/STACK_2 **nicht** von den gleichen Segment-Registern adressierbar sind.

- Beim Übergang von einem Modul zum anderen muß der Inhalt des CS-Registers geändert werden.
- Die beiden Daten-Regionen können nur durch permanente Änderungen des DS-Registerwertes adressiert werden. Wird DS zur Adressierung der einen Daten-Region und ES zur Adressierung der anderen Daten-Region benützt, bedeutet dies eine Fülle von Segment-Override-Prefix-Bytes.
- Da ein Programm nur eine Stack-Region braucht, würde die andere unbenutzt bleiben.

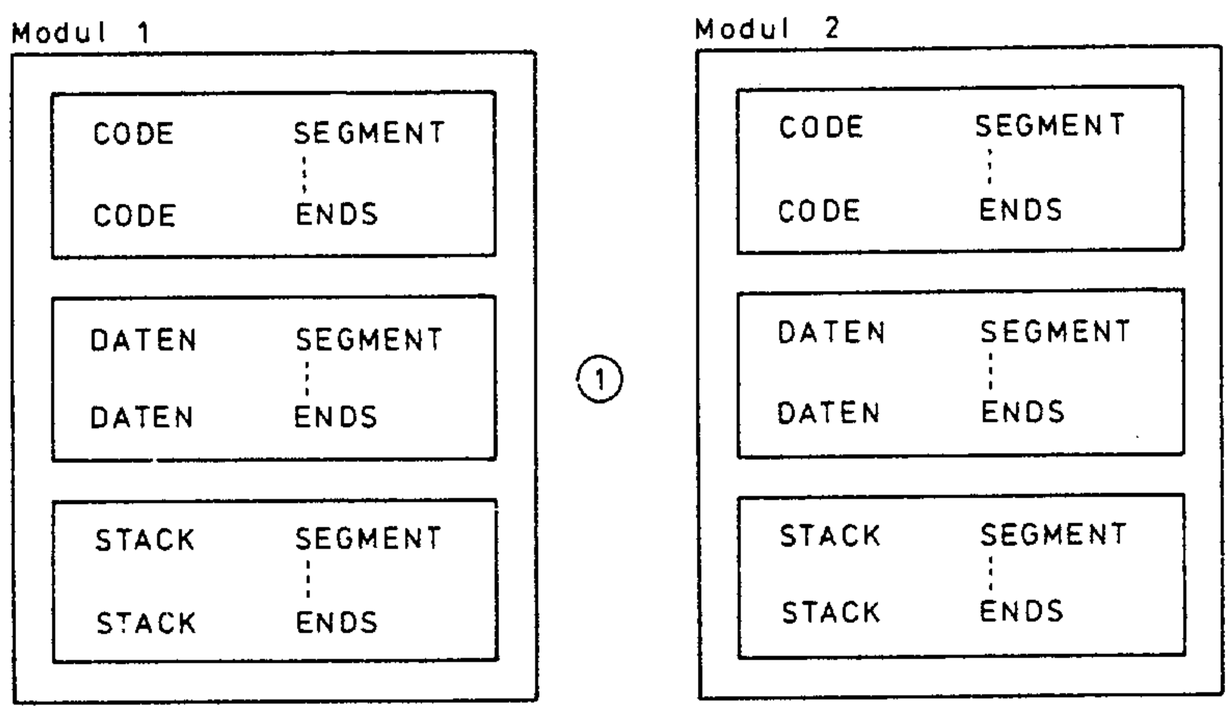
Es sei vorausgesetzt, daß die kombinierte Größe eines Code-Segments **kleiner** ist, als die Größe eines physikalischen Segments (64 KByte). Die Befehle in beiden Code-Regionen könnten nun von einem **gemeinsamen** CS-Wert erreicht werden. D.h. beim Übergang von einem Modul zum anderen wird nur der IP-Wert geändert. Die langen Formen der CALL- und JMP-Befehle werden vermieden.

Ein ähnliches Argument gilt auch für das Kombinieren der beiden Daten-Regionen. Beide könnten relativ zu DS adressiert werden.

Schließlich gelten auch für die beiden Stack-Regionen die angegebenen Überlegungen. Die kombinierten Segmente würden nun im Speicher folgendes Bild ergeben.



Logische Segmente, die in der angegebenen Weise kombiniert werden sollen, müssen **identische Segmentnamen** haben.



So können in den Modulen Modul 1 und Modul 2 die beiden Code-Regionen z.B. CODE, die beiden Daten-Regionen z.B. DATEN und die beiden Stack-Regionen z.B. STACK heißen ①.

Wie können nun die logischen Segmente von Modul 1 und Modul 2 korrekt kombiniert werden?

Die Antwort liegt in der **Attributliste** des Assembler-SEGMENT-Statements:

```
Segmentname SEGMENT [Attributliste]
                .
                .
                .
                .
Segmentname ENDS
```

Die Attributliste wurde in den bisherigen Programmbeispielen nicht benutzt. Der Grund dafür ist, daß es für logische Segmente **Default-Attribute** gibt. Sie werden akzeptiert für Fälle, in denen keine logischen Segmente kombiniert werden.

Die Attributliste wird bedeutungsvoll, wenn Segmente kombiniert werden. Sie ist aus drei Feldern zusammengesetzt:

- dem Align-Typ (Ausrichtung)
- dem Combine-Typ
- dem Class-Namen

Syntax des erweiterten SEGMENT-Statements:

```
Segmentname SEGMENT [Align-Typ] [Combine-Typ] ['Class-Name']
                .
                .
                .
Segmentname ENDS
```

Hinweis: Die eckigen Klammern bedeuten, daß die in ihnen angegebenen Attribute Optionen sind.

5.1.1 Das Combine-Typ-Attribut

Es gibt an, **wie** ein logisches Segment mit einem anderen logischen Segment **gleichen Namens** zu kombinieren ist. Ist für ein Segment kein Combine-Typ spezifiziert, wird es als **nicht-kombinierbar** betrachtet. Dieses Default-Combine-Typ-Attribut bedeutet also, daß das Segment mit keinem anderen logischen Segment kombiniert werden kann.

Als Combine-Typ-Optionen existieren die beiden Schlüsselworte

- PUBLIC und
- STACK

Zunächst sei der **PUBLIC**-Combine-Typ betrachtet. Er wird sowohl für Code-Segmente, als auch für Daten-Segmente verwendet. Seine Wirkung

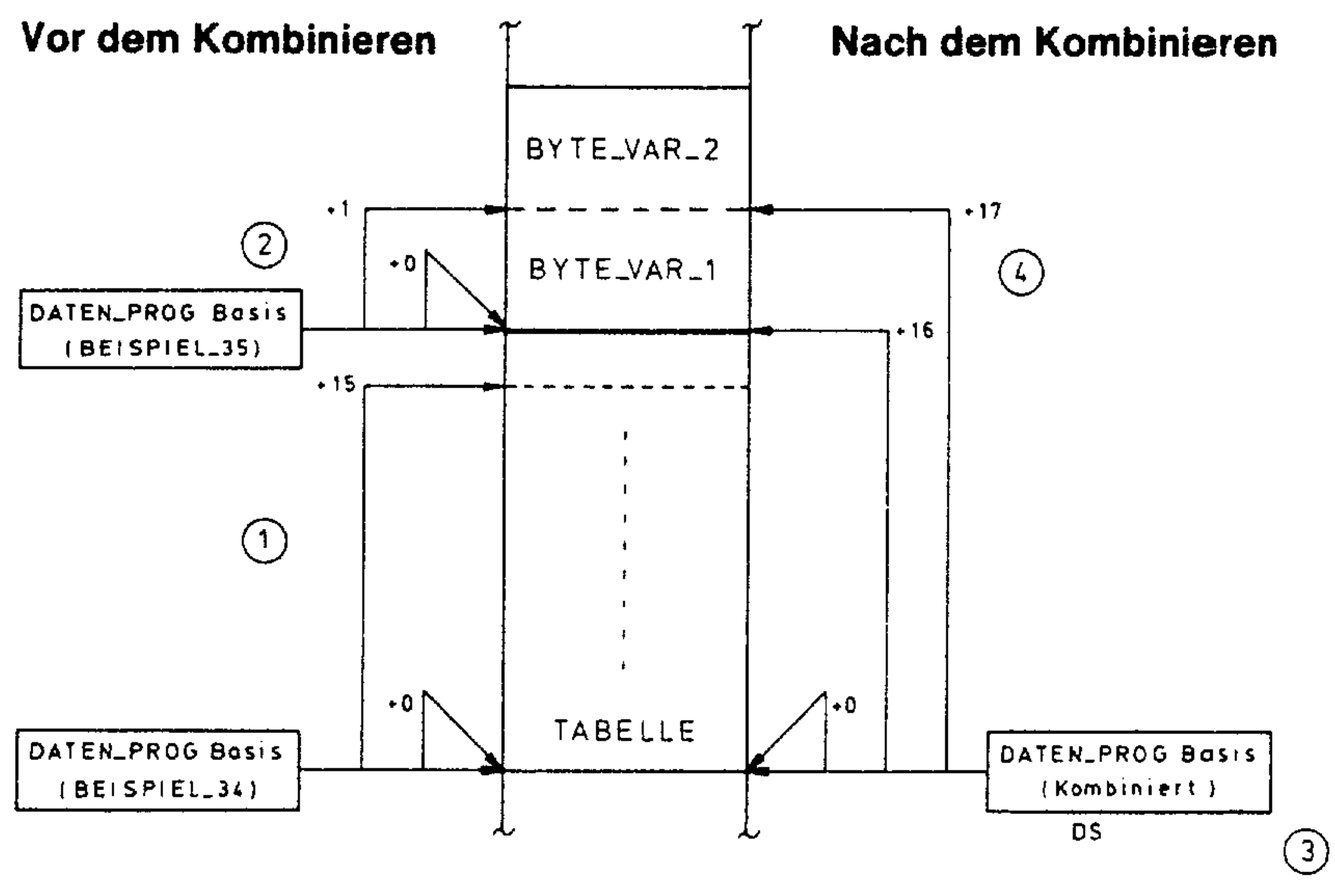
soll am Beispiel der beiden angegebenen Module BEISPIEL_34 und BEISPIEL_35 gezeigt werden.

```
LOC  OBJ                    LINE     SOURCE

                               1             NAME    BEISPIEL_34
                               2
                               3     ASSUME DS:DATEN_PROG
                               4
----                           5     DATEN_PROG SEGMENT PUBLIC
                               6
0000 (16                       7     TABELLE     DB      16 DUP (?)
     ??
     )      ①
                               8
----                           9     DATEN_PROG ENDS
                              10
                              11             END
```

```
LOC  OBJ                    LINE     SOURCE

                               1             NAME    BEISPIEL_35
                               2
                               3     ASSUME DS:DATEN_PROG
                               4
----                           5     DATEN_PROG SEGMENT PUBLIC
                               6
0000 ??        ②               7     BYTE_VAR_1      DB      ?
0001 ??                        8     BYTE_VAR_2      DB      ?
                               9
----                          10     DATEN_PROG ENDS
                              11
                              12             END
```

In beiden Modulen existieren die PUBLIC-Segmente DATEN_PROG. Im Modul BEISPIEL_34 hat das Segment eine Länge von 16 Bytes mit den Offsets 0 bis 15 ①. Im Modul BEISPIEL_35 hat das Segment eine Länge von 2 Bytes mit den Offsets 0 bis 1 ②.

Was passiert, wenn LINK86 die beiden Module in der Reihenfolge BEISPIEL_34, BEISPIEL_35 kombiniert?



Vor dem Kombinieren liegen die Offsetwerte von Tabelle zwischen +0 und +15D (16 Bytes) ①. Der Offsetwert von BYTE_VAR_1 liegt bei +0, der von BYTE_VAR_2 bei +1 ②.

Werden die beiden PUBLIC-Segmente mit dem Linker kombiniert, bedeutet dies, daß ihre Variablen von einer gemeinsamen Basisposition aus erreicht werden. Für beide Module stellt das Symbol DATEN_PROG die Basis der kombinierten logischen Segmente dar. Ist DS mit DATEN_PROG geladen, kann DS benützt werden, um TABELLE, BYTE_VAR_1 und BYTE_VAR_2 zu adressieren ②.

Da der Linker die Segmente in der Reihenfolge BEISPIEL_34, BEISPIEL_35 kombiniert, werden die Offsetwerte von BYTE_VAR_1 und BYTE_VAR_2 korrigiert. Relativ zur gemeinsamen Basisposition wird nun der Offsetwert von BYTE_VAR_1 mit +16D und der Offsetwert von BYTE_VAR_2 mit +17D festgelegt ④.

Hinweis: Die beschriebene Situation gilt in analoger Weiser auch für PUBLIC-Code-Segmente.

Es soll nun das Kombinieren von Stack-Segmenten betrachtet werden. Als Combine-Typ ist hier **STACK** zu verwenden. Seine Wirkung soll am Beispiel der beiden angegebenen Module BEISPIEL_36 und BEISPIEL_37 gezeigt werden.

```
LOC  OBJ                    LINE     SOURCE

                               1             NAME    BEISPIEL_36
                               2
                               3     ASSUME CS:CODE_PROG
                               4
----                           5     STACK_PROG SEGMENT STACK
                               6
0000 (24                       7             DW      24 DUP (?)
     ????
     )
0030                           8     TOP_STACK_1 LABEL WORD
                               9
----                          10     STACK_PROG ENDS
                              11
----                          12     CODE_PROG SEGMENT PUBLIC
                              13
0000 BC3000          R        14     START:  MOV     SP,OFFSET TOP_STACK_1       ;SP initialisieren
                              15
----                          16     CODE_PROG ENDS
                              17
                              18             END     START
```

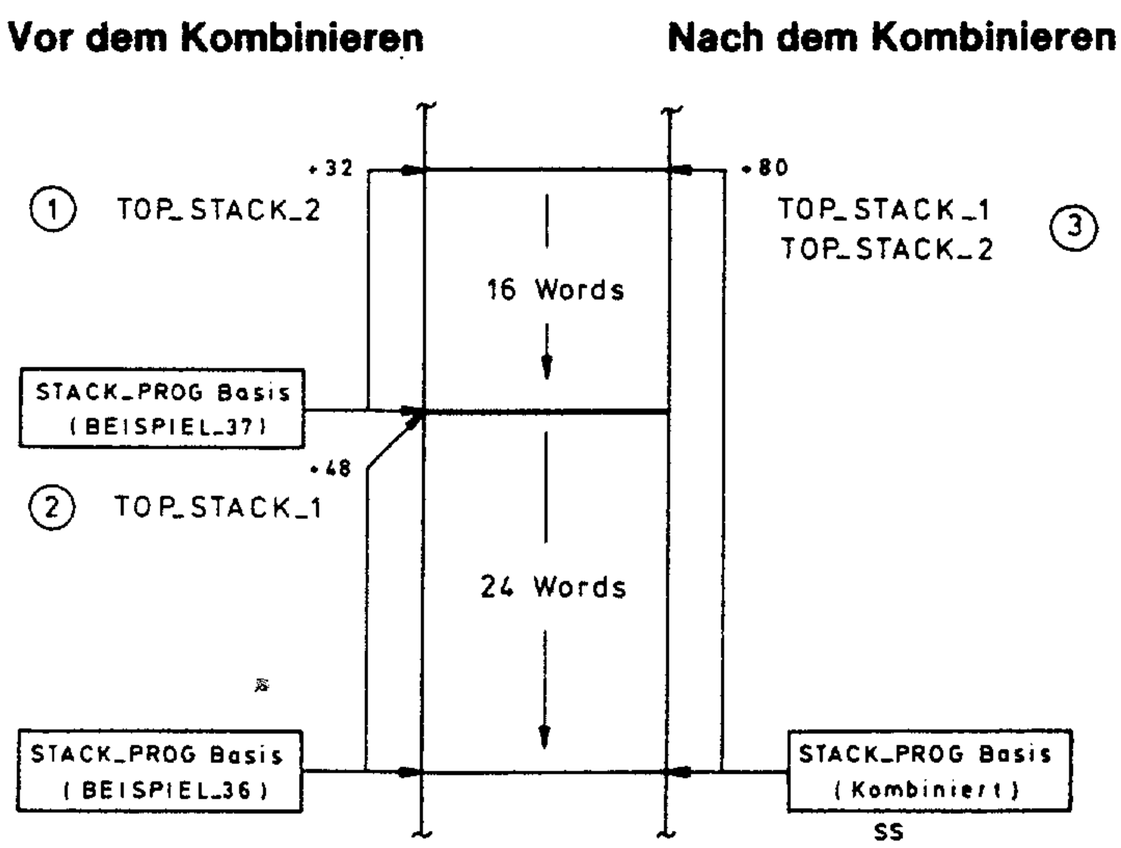
```
LOC  OBJ                    LINE     SOURCE

                              1              NAME    BEISPIEL_37
                              2
                              3     ASSUME CS:CODE_PROG
                              4
----                          5     STACK_PROG SEGMENT STACK
                              6
0000 (16                      7              DW      16 DUP (?)
     ????
     )
0020                          8     TOP_STACK_2 LABEL WORD
                              9
----                         10     STACK_PROG ENDS
                             11
----                         12     CODE_PROG SEGMENT PUBLIC
                             13
0000 BC2000          R       14     START:  MOV     SP,OFFSET TOP_STACK_2       ;SP initialisieren
                             15
----                         16     CODE_PROG ENDS
                             17
                             18              END
```

Es sei vorausgesetzt, daß im Modul BEISPIEL_36 ein **STACK**-Segment mit einer Länge von 24 Words definiert ist. Das Symbol TOP_STACK_1 kennzeichnet den Anfang der Stack-Region.

Weiterhin sei vorausgesetzt, daß im Modul BEISPIEL_37 ein **STACK**-Segment mit einer Länge von 16 Words definiert ist. Das Symbol TOP_STACK_2 kennzeichnet den Anfang der Stack-Region.

Da keine Zugriffe auf Variablen in den Stack-Regionen erfolgen, ist in beiden Modulen `ASSUME SS: NOTHING` wirksam.

Was passiert, wenn der Linker die beiden Module in der Reihenfolge BEISPIEL_36, BEISPIEL_37 kombiniert?



Vor dem Kombinieren ist das SP-Register im Modul BEISPIEL_37 mit dem Offset von TOP_STACK_2 geladen (`MOV SP, OFFSET TOP_STACK_2`). Damit zeigt SP im Abstand von +32D Bytes relativ zu STACK_PROG auf die leere Stack-Region ①.

Im Modul BEISPIEL_36 ist das SP-Register mit dem Offset von TOP_STACK_1 geladen (`MOV SP, OFFSET TOP_STACK_1`). Damit zeigt SP im Abstand von +48D Bytes relativ zu STACK_PROG auf die leere Stack-Region ②.

Werden die beiden **STACK**-Segmente mit dem Linker kombiniert, wird eine lückenlose Stack-Region mit einer Länge von 40 Words gebildet. Der Combine-Typ **STACK** korrigiert die Offsetwerte von TOP_STACK_1 und TOP_STACK_2. Beide beziehen sich nun auf die erste Position jenseits der Stack-Region, d.h. ihre Offsetwerte liegen identisch bei +80D Bytes ③.

In einem Programm ist das SP-Register nur einmal im **Haupt**-Modul zu initialisieren. Existieren jedoch in anderen Modulen ebenfalls Stack-Segmente, muß in jedem Modul für den individuellen Test das SP-Register initialisiert sein. Vor dem Kombinieren der Einzelmodule zu einem Gesamtmodul werden dann, bis auf den Hauptmodul, die SP-Initialisierungsbefehle entfernt.

Bei den angegebenen Programm-Listings entspricht der Modul BEISPIEL_38 dem Modul BEISPIEL_36 und der Modul BEISPIEL_39 entspricht dem Modul BEISPIEL_37.

```
LOC  OBJ                    LINE     SOURCE

                               1             NAME    BEISPIEL_38
                               2
                               3     ASSUME CS:CODE_PROG
                               4
----                           5     STACK_PROG SEGMENT STACK
                               6
0000 (24                       7             DW      24 DUP (?)
     ????
     )
0030                           8     TOP_STACK_1 LABEL WORD
                               9
----                          10     STACK_PROG ENDS
                              11
----                          12     CODE_PROG SEGMENT PUBLIC
                              13
0000 B8----          R        14     START:  MOV     AX,STACK_PROG          ;SS Register initialisieren
0003 8ED0                     15             MOV     SS,AX                                              ①
0005 BC3000          R        16             MOV     SP,OFFSET TOP_STACK_1  ;SP initialisieren
                              17
----                          18     CODE_PROG ENDS
                              19
                              20             END     START
```

```
LOC  OBJ                    LINE     SOURCE

                               1             NAME    BEISPIEL_39
                               2
----                           3     STACK_PROG SEGMENT STACK
                               4
0000 (16                       5             DW      16 DUP (?)
     ????
     )
                               6
----                           7     STACK_PROG ENDS
                               8
                               9             END
```

Das Modul BEISPIEL_38 ist das **Hauptmodul**. Es enthält ein Stack-Segment mit 24 Word. Zusätzlich werden die SS- und SP-Register für das Gesamtmodul initialisiert ①.

Das Modul BEISPIEL_39 enthält nur ein Stack-Segment mit 16 Words. Ein Symbol, das den Anfang dieser Stack-Region kennzeichnet, kann entfallen. Nach dem Kombinieren wird nun der Offset von STACK_TOP_1 mit dem Wert +80D den Anfang der gesamten Stack-Region festlegen.

5.1.2 Das Align-Typ-Attribut

Das Align-Typ-Attribut spezifiziert die Grenze (boundary) an die ein logisches Segment positioniert werden soll. Ein Segment kann eines von fünf Alignment-Attributen haben. Die drei Wichtigsten sollen erläutert werden.

- Der **PARA**-Align-Typ gibt an, daß ein Segment nur an eine Adresse positioniert werden kann, die durch 16 teilbar ist (Paragraph Boundary). Ist **kein** Align-Typ explizit spezifiziert, wird PARA als Default-Alignment-Attribut vorausgesetzt.
- Der **WORD**-Align-Typ gibt an, daß ein Segment nur an eine Adresse positioniert werden kann, die durch 2 teilbar ist (Adreßbit A0 = 0).
- Der **BYTE**-Align-Typ gibt an, daß ein Segment an jede Adresse (gerade oder ungerade) positioniert werden kann (Adreßbit A0 = 1 oder 0).

Hinweis: Der BYTE-Align-Typ sollte für Datenregionen in 8086/80186/80286-Programmen **nicht** verwendet werden, da jeder Zugriff auf eine Word-Variable unter ungerader Adresse **zwei** Buszyklen erfordert.

Die Wirkung der Align-Typ-Attribute **PARA**, **WORD** und **BYTE** soll an den folgenden Beispielen gezeigt werden.

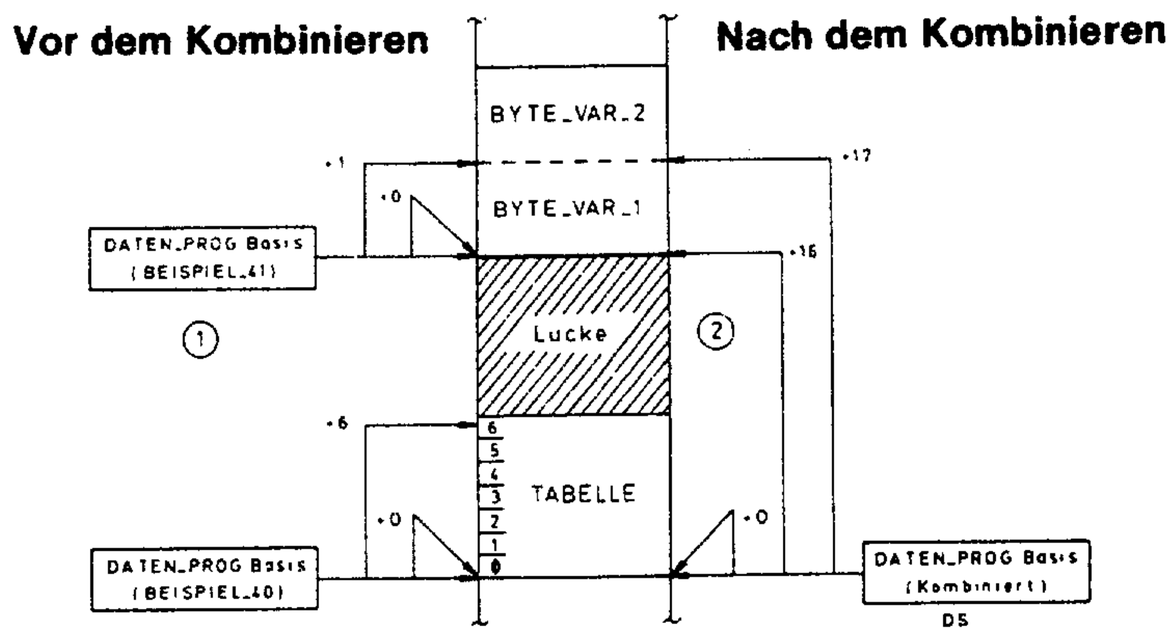
Beispiel 1: Align-Typ PARA

Vor dem Kombinieren existieren in den Modulen Beispiel_40 und Beispiel_41 die PUBLIC-Segmente DATEN_PROG mit dem Align-Typ PARA.

```
LOC  OBJ                  LINE     SOURCE

                            1              NAME    BEISPIEL_40
                            2
                            3     ASSUME DS:DATEN_PROG
                            4
----                        5     DATEN_PROG SEGMENT PARA PUBLIC
                            6
0000 (7                     7     TABELLE     DB      7 DUP (?)
     ??
     )
                            8
----                        9     DATEN_PROG ENDS
                           10
                           11              END
```

```
LOC  OBJ                  LINE     SOURCE

                            1              NAME    BEISPIEL_41
                            2
                            3     ASSUME DS:DATEN_PROG
                            4
----                        5     DATEN_PROG SEGMENT PARA PUBLIC
                            6
0000 ??                     7     BYTE_VAR_1      DB      ?
0001 ??                     8     BYTE_VAR_2      DB      ?
                            9
----                       10     DATEN_PROG ENDS
                           11
                           12              END
```

Im Modul BEISPIEL_40 liegen die Offsetwerte von TABELLE vor dem Kombinieren zwischen +0 und +6D (7 Bytes). Die Offsetwerte von BYTE_VAR_1 und BYTE_VAR_2 im Modul BEISPIEL_41 liegen bei +0 und +1 ①.



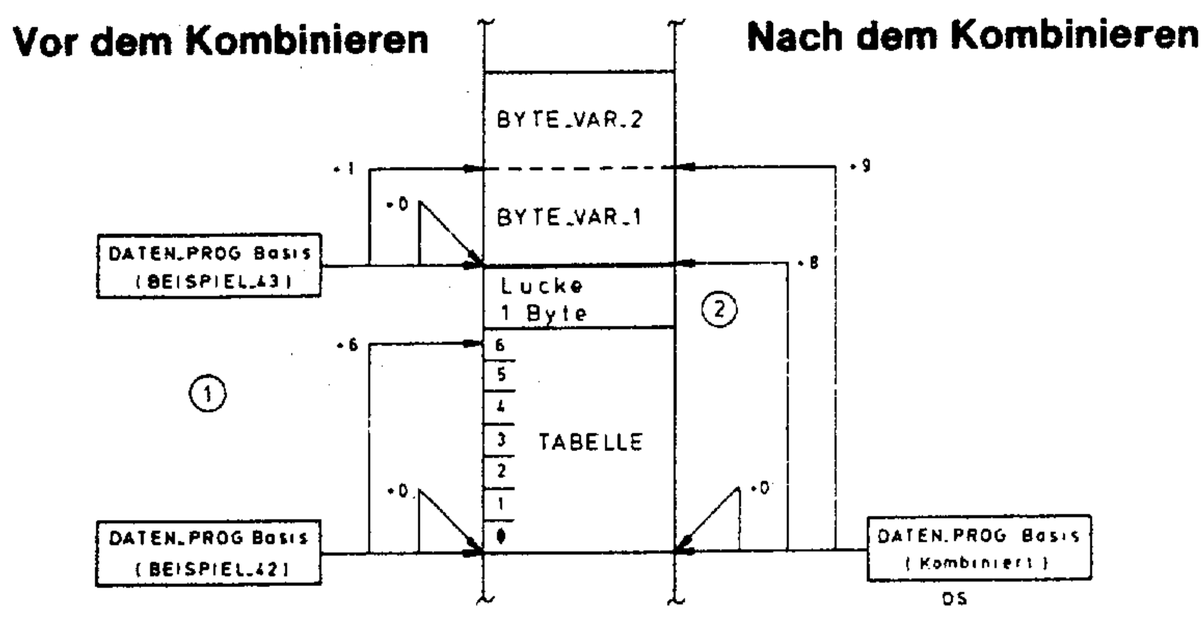
Nach dem Kombinieren beginnen wegen des Align-Typs **PARA** beide PUBLIC-Segmente an einer Adresse, die durch 16 teilbar ist. Während sich die Offsetwerte von TABELLE nicht ändern, wird der Offsetwert von BYTE_VAR_1 auf +16D (Paragraph Boundary) und zwangsläufig der von BYTE_VAR_2 auf +17D festgelegt. D.h. zwischen beiden logischen Segmenten entsteht eine Lücke ②.

Beispiel 2: Align-Typ WORD

Die angegebenen Module BEISPIEL_42 und BEISPIEL_43 entsprechen den Modulen BEISPIEL_40 und BEISPIEL_41. Die PUBLIC-Segmente DATEN_PROG haben nun den Align-Typ WORD.

```
LOC  OBJ                  LINE     SOURCE

                            1              NAME    BEISPIEL_42
                            2
                            3     ASSUME DS:DATEN_PROG
                            4
----                        5     DATEN_PROG SEGMENT WORD PUBLIC
                            6
0000 (7                     7     TABELLE     DB      7 DUP (?)
     ??
     )
                            8
----                        9     DATEN_PROG ENDS
                           10
                           11              END


LOC  OBJ                  LINE     SOURCE

                            1              NAME    BEISPIEL_43
                            2
                            3     ASSUME DS:DATEN_PROG
                            4
----                        5     DATEN_PROG SEGMENT WORD PUBLIC
                            6
0000 ??                     7     BYTE_VAR_1      DB      ?
0001 ??                     8     BYTE_VAR_2      DB      ?
                            9
----                       10     DATEN_PROG ENDS
                           11
                           12              END
```

Im Modul BEISPIEL_42 liegen die Offsetwerte von TABELLE vor dem Kombinieren zwischen +0 und +6D (7 Bytes). Die Offsetwerte von BYTE_VAR_1 und BYTE_VAR_2 im Modul BEISPIEL_43 liegen bei +0 und +1 ①.



Nach dem Kombinieren beginnen wegen des Align-Typs WORD beide PUBLIC-Segmente an einer Adresse, die durch 2 teilbar ist. Während sich die Offsetwerte von TABELLE nicht ändern, wird der Offsetwert von BYTE_VAR_1 auf +8D (Word Boundary) und zwangsläufig der von BYTE_VAR_2 auf +9D festgelegt. D.h. zwischen beiden logischen Segmenten entsteht eine Lücke von einem Byte ②.

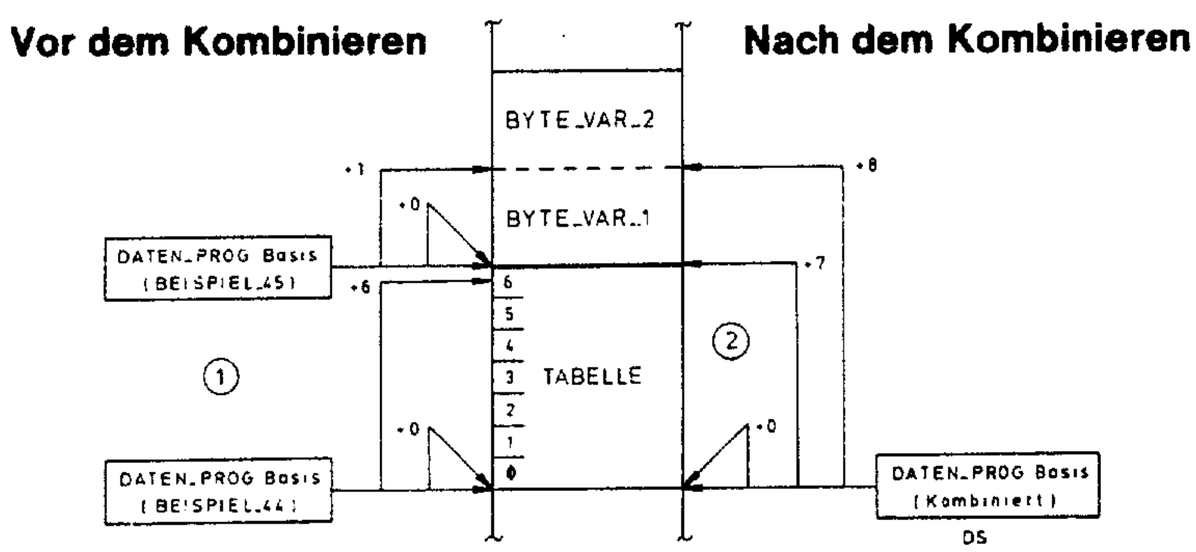
Beispiel 3: Align-Typ BYTE

Die angegebenen Module BEISPIEL_44 und BEISPIEL_45 entsprechen den Modulen BEISPIEL_40 und BEISPIEL_41. Die PUBLIC-Segmente DATEN_PROG haben nun den Align-Typ BYTE.

```
LOC  OBJ                    LINE     SOURCE

                               1             NAME    BEISPIEL_44
                               2
                               3     ASSUME DS:DATEN_PROG
                               4
----                           5     DATEN_PROG SEGMENT BYTE PUBLIC
                               6
0000 (7                        7     TABELLE     DB      7 DUP (?)
     ??
     )
                               8
----                           9     DATEN_PROG ENDS
                              10
                              11             END
```

```
LOC  OBJ                    LINE     SOURCE

                               1             NAME    BEISPIEL_45
                               2
                               3     ASSUME DS:DATEN_PROG
                               4
----                           5     DATEN_PROG SEGMENT BYTE PUBLIC
                               6
0000 ??                        7     BYTE_VAR_1      DB      ?
0001 ??                        8     BYTE_VAR_2      DB      ?
                               9
----                          10     DATEN_PROG ENDS
                              11
                              12             END
```

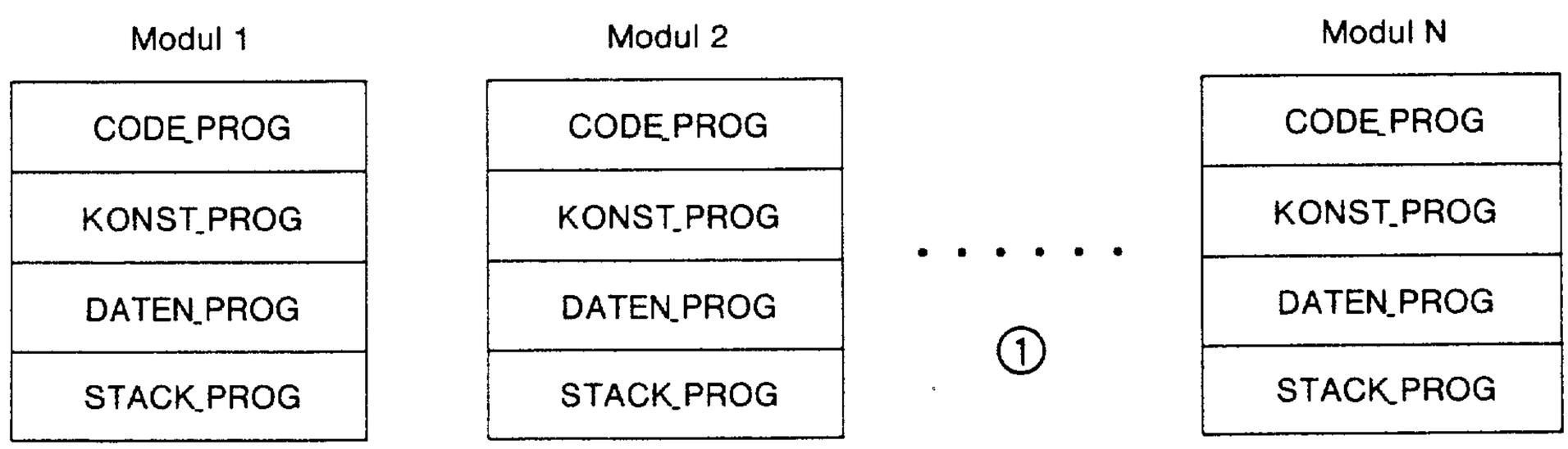
In BEISPIEL_44 liegen die Offsetwerte von TABELLE vor dem Kombinieren zwischen +0 und +6D (7 Bytes). Die Offsetwerte von BYTE_VAR_1 und BYTE_VAR_2 im Modul BEISPIEL_45 liegen bei +0 und +1 ①.



Da wegen des Alignment-Attributs **BYTE** ein logisches Segment an jede Adresse positioniert werden kann, wird nach dem Kombinieren zwischen den Segmenten **keine** Lücke entstehen. Während TABELLE bei Offset +0 (gerade Adresse) beginnt, wird BYTE_VAR_1 mit einem Offsetwert von +7D (ungerade Adresse) lückenlos an TABELLE angefügt. Der Offsetwert von BYTE_VAR_2 ergibt sich dann zwangsläufig zu +8D ②.

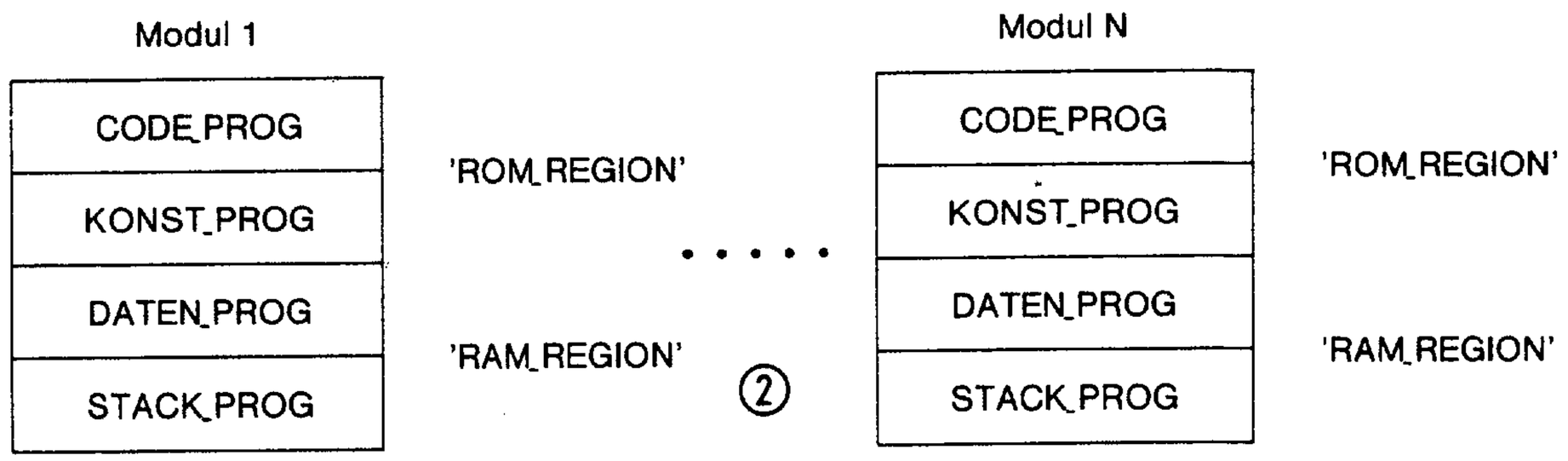
5.1.3 Das Class-Name Attribut

Es soll ein Programm existieren, das in mehrere Module zerlegt worden ist. Jedes Modul sei aus einer Code-Region (CODE_PROG), einer Konstanten-Region (KONST_PROG), einer Daten-Region (DATEN_PROG) und einer Stack-Region (STACK_PROG) zusammengesetzt ①.



Es gehört die Code-Region und die Konstanten-Region eines Programms zur Klasse von Segmenten, die physikalisch im ROM (PROM) hinterlegt werden. Im Gegensatz dazu, gehören die Daten- und Stack-Region eines Programms zur **Klasse** von Segmenten, die physikalisch im RAM stehen .

In Assembler ist es möglich, Programm-Regionen gleicher Klasse mit einem gemeinsamen **Class-Namen** zu kennzeichnen. So könnten z.B. die Code- und Konstanten-Regionen eines Programms mit 'ROM_REGION' und die Daten- und die Stack-Regionen mit 'RAM_REGION' bezeichnet werden ②.



Stellvertretend für alle Module zeigt das angegebene Programmlisting die Anwendung des Class-Name-Attributs.

```
LOC  OBJ                  LINE     SOURCE

                            1              NAME    BEISPIEL_46
                            2
                            3      ASSUME CS:CODE_PROG,DS:DATEN_PROG,ES:KONST_PROG
                            4
----                        5      CODE_PROG SEGMENT WORD PUBLIC 'ROM_REGION'
                            6
0000 B8----          R      7      START:  MOV     AX,KONST_PROG          ;ES Register initialisieren
0003 8EC0                   8              MOV     ES,AX
0005 B8----          R      9              MOV     AX,DATEN_PROG          ;DS Register initialisieren
0008 8ED8                  10              MOV     DS,AX
000A B8----          R     11              MOV     AX,STACK_PROG          ;SS Register initialisieren
000D 8ED0                  12              MOV     SS,AX
000F BC140090        R     13              MOV     SP,OFFSET TOP_STACK    ;SP Register initialisieren
0013 268B0E0000      R     14              MOV     CX,ES:KONST_1          ;CX = KONST_1 aus der ROM-Region
0018 890E0000        R     15              MOV     VAR_1,CX               ;KONST_1 in die RAM-Region laden
                           16
----                       17      CODE_PROG ENDS
                           18
----                       19      KONST_PROG SEGMENT WORD PUBLIC 'ROM_REGION'
                           20
0000 3412                  21      KONST_1 DW      1234H
                           22
----                       23      KONST_PROG ENDS
                           24
----                       25      DATEN_PROG SEGMENT WORD PUBLIC 'RAM_REGION'
                           26
0000 ????                  27      VAR_1   DW      ?
                           28
----                       29      DATEN_PROG ENDS
                           30
----                       31      STACK_PROG SEGMENT WORD STACK 'RAM_REGION'
                           32
0000 (10                   33              DW      10 DUP (?)
     ????
     )
                           34
0014                       35      TOP_STACK LABEL WORD
                           36
----                       37      STACK_PROG ENDS
                           38
                           39              END
```

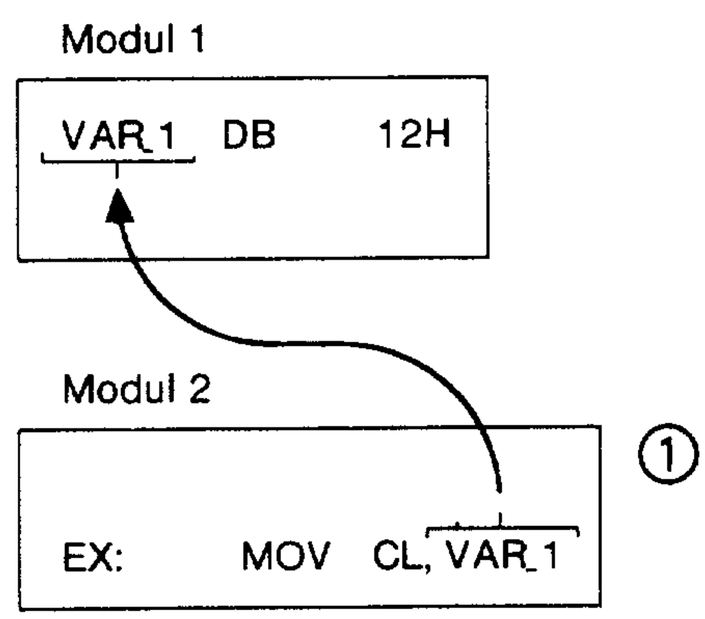
Die Segmente CODE_PROG und KONST_PROG gehören zur Klasse ROM_REGION und werden physikalisch im ROM hinterlegt. Die Segmente DATEN_PROG und STACK_PROG gehören zur Klasse RAM_REGION und werden physikalisch im RAM hinterlegt.

- Die "Executable"-Daten in CODE_PROG werden relativ zu CS erreicht.
- Die "Read-Only"-Daten in KONST_PROG werden relativ zu ES erreicht.
- Die "Read-Write"-Daten in DATEN_PROG werden relativ zu DS erreicht.
- Die "Read-Write"-Daten in STACK_PROG werden relativ zu (DS) SS erreicht.

Hinweis: Der Class-Name, der bestimmte bereits mit dem Linker kombinierte Programm-Regionen kennzeichnet, gibt nur an, daß die Regionen getrennt von anderen kombinierten Regionen im physikalischen Speicher hinterlegt werden. Im Unterschied zum **Combine**-Typ bewirkt der **Class**-Name nicht, daß kombinierte Regionen von gemeinsamen Segment-Registerwerten aus adressierbar sind.

5.1.4 Die PUBLIC- und EXTRN-Direktiven

Es soll ein Programm entwickelt werden, das aus zwei Modulen besteht: **Modul 1** und **Modul 2**. In Modul 1 sei die Variable VAR_1 definiert, auf die in Modul 2 zugegriffen wird ①.



Damit dieser Zugriff erfolgen kann, muß in Modul 1 die Variable VAR_1 für andere Module verfügbar gemacht werden. Dies ist mit Hilfe der PUBLIC-Direktive möglich ②.

Modul 1

```
VAR_1     DB     12H
PUBLIC    VAR_1
```

②

Im Modul 2 muß dem Assembler mitgeteilt werden, daß VAR_1 in einem anderen Modul als Variable vom Typ Byte definiert ist. Diese Information bekommt der Assembler von der **EXTRN**-Direktive ③.

Modul 2

```
EXTRN     VAR_1 : BYTE
          :
EX:       MOV   CL, VAR_1
```

Hinweis: Während bei der EXTRN-Direktive eine Typ-Information erforderlich ist, ist sie bei der PUBLIC-Direktive nicht notwendig. Der Grund ist einfach: Die PUBLIC-Direktive wird in dem Modul benützt, in dem VAR_1 definiert ist. Der Assembler kennt also den Typ von VAR_1. Da in Modul 2 der Typ von VAR_1 nicht bekannt ist, muß die EXTRN-Direktive die notwendige Typ-Information liefern.

Syntax des PUBLIC-Statements:

```
PUBLIC  Symbol  [,Symbol] ..
```

Syntax des EXTRN-Statements:

```
EXTRN  Symbol: Typ [,Symbol: Typ] ..
```

Die eckigen Klammern geben an, daß mehrere Symbole durch Komma getrennt in den PUBLIC- oder EXTRN-Statements angegeben werden können. Ein Symbol kann der Name einer Variablen, ein Label (im Code Segment) oder ein konstanter Wert (definiert mit EQU) sein.

Im EXTRN-Statement sind folgende Typ-Angaben möglich:

- BYTE – eine Variable vom Typ Byte
- WORD – eine Variable vom Typ Word
- DWORD – eine Variable vom Typ Double Word
- QWORD – eine Variable vom Typ Quad Word (Float)
- TBYTE – eine Variable vom Typ Ten Bytes (Long Float? BCD)
- NEAR – ein Label innerhalb eines physikalischen Segments
- FAR – ein Label außerhalb eines physikalischen Segments
- ABS – eine Word-Konstante (16 Bits)

5.1.4.1 Die Plazierung von EXTRNs

Es sei vorausgesetzt, daß zwei Module das gleiche logische Segment enthalten. Dabei soll im Segment des zweiten Moduls ein PUBLIC-Symbol definiert sein.

In diesem Fall ist die EXTRN-Direktive im ersten Modul zwischen einem SEGMENT/ENDS-Paar zu plazieren, das identisch ist mit dem SEGMENT/ENDS-Paar, in dem das Symbol als PUBLIC definiert ist. Das Symbol kann dann in gewohnter Weise wie eine normale Variable, ein Label oder eine Konstante benützt werden.

Beispiel: In den Modulen Modul1 und Modul2 existieren die kombinierbaren Segmente DATEN_PROG und CODE_PROG.

Modul1

```
DATEN_PROG SEGMENT WORD PUBLIC

MITTEL_WERT DW 1 DUP (?)
PUBLIC MITTEL_WERT ①
EXTRN ZAHLEN_PAAR:WORD ②
DATEN_PROG ENDS

CODE_PROG SEGMENT BYTE PUBLIC
          .
A:MOV AX,ZAHLEN_PAAR
          .
CODE_PROG ENDS
```

Modul2

```
DATEN_PROG SEGMENT WORD PUBLIC

ZAHLEN_PAAR DW 100,28
PUBLIC ZAHLEN_PAAR ①
EXTRN MITTEL_WERT:WORD ②
DATEN_PROG ENDS

CODE_PROG SEGMENT BYTE PUBLIC
          .
A:MOV MITTEL_WERT,DX
          .
CODE_PROG ENDS
```

Im Modul 1 wird mit `MOV AX, ZAHLEN_PAAR` auf die externe Variable ZAHLEN_PAAR im Modul 2 zugegriffen. Sie ist dort als **PUBLIC** ① definiert.

Im Modul 2 wird mit `MOV MITTEL_WERT, DX` auf die externe Variable MITTEL_WERT im Modul 1 zugegriffen. Sie ist dort als **PUBLIC** ① definiert.

In beiden Fällen sind die Variablen in jeweils einem anderen Modul, aber in bekannten Segmenten mit **identischen** Namen definiert.

Die EXTRN-Direktiven werden daher in beiden Modulen zwischen die identischen SEGMENT/ENDS-Paare plaziert ②.

Sollten in verschiedenen Modulen **keine** kombinierbaren Segmente existieren, müssen die EXTRN-Direktiven **außerhalb** der SEGMENT/ENDS-Paare plaziert werden.

Programmbeispiel:

Im folgenden Programmbeispiel wird der Mittelwert zweier Integer-Zahlen gebildet. Das Programm ist in die zwei Module BEISPIEL_47 und BEISPIEL_48 zerlegt.

```
LOC  OBJ                    LINE     SOURCE

                              1              NAME BEISPIEL_47
                              2
                              3      ASSUME CS:CODE_PROG,DS:DATEN_PROG
                              4
----                          5      DATEN_PROG SEGMENT WORD PUBLIC 'RAM_REGION'
                              6
0000 (1                       7      MITTEL_WERT     DW      1 DUP (?)       ;Mittelwert aus zwei externen
     ????
     )
                              8                                              ;Zahlen speichern
                              9      PUBLIC MITTEL_WERT
                             10      EXTRN ZAHLEN_PAAR:WORD
                             11
----                         12      DATEN_PROG ENDS
                             13
----                         14      CODE_PROG SEGMENT BYTE PUBLIC 'ROM_REGION'
                             15
0000 B8----          R       16      START:  MOV     AX,DATEN_PROG           ;DS Register initialisieren
0003 8ED8                    17              MOV     DS,AX
0005 BE0000                  18              MOV     SI,0                    ;Quellindex = 0 fuer externes Feld
0008 8B840000        E       19              MOV     AX,ZAHLEN_PAAR[SI]      ;(AX) = erste Zahl von extern
000C 8B9C0200        E       20              MOV     BX,ZAHLEN_PAAR[SI+2]    ;(BX) = zweite Zahl von extern
                             21
----                         22      CODE_PROG ENDS
                             23
                             24              END     START
```

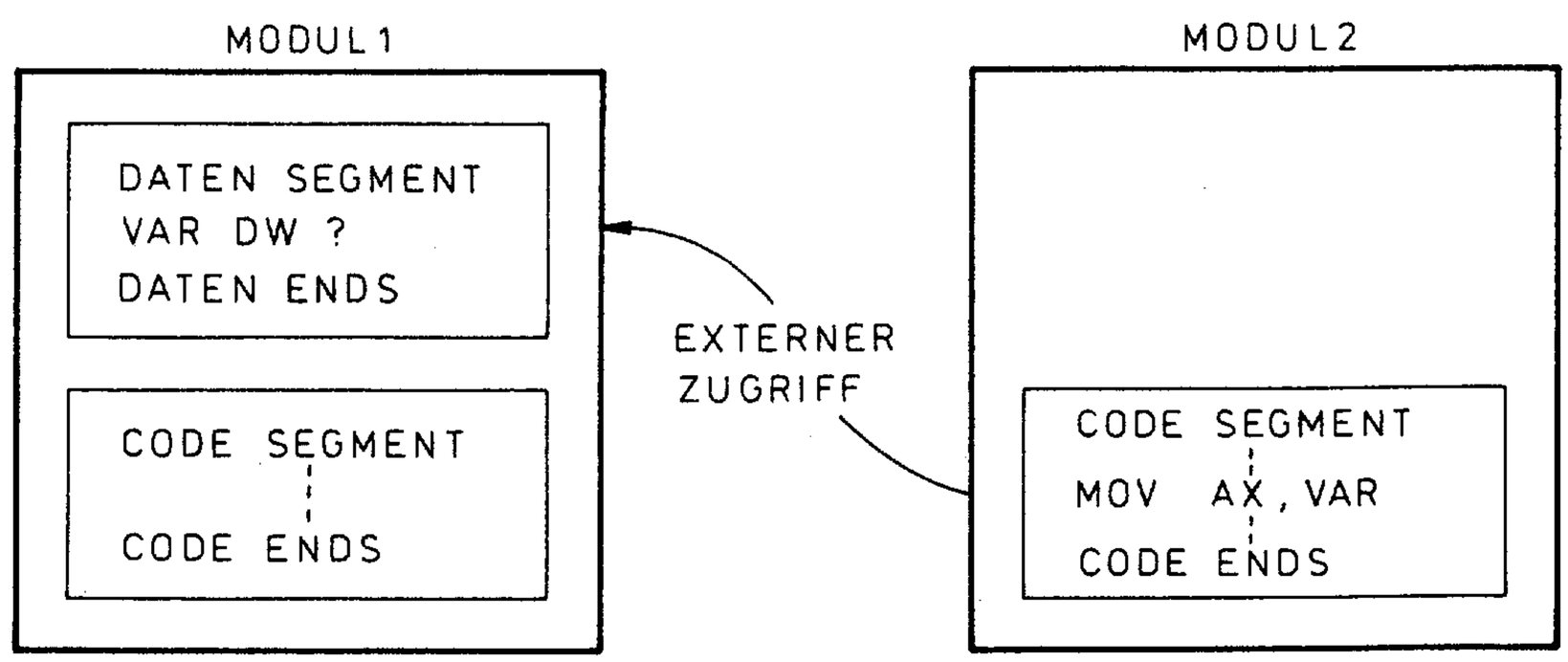
```
LOC  OBJ                  LINE     SOURCE

                            1              NAME    BEISPIEL_48
                            2
                            3     ASSUME CS:CODE_PROG,DS:DATEN_PROG
                            4
----                        5     DATEN_PROG SEGMENT WORD PUBLIC 'RAM_REGION'
                            6
0000 6400                   7     ZAHLEN_PAAR     DW      100,28          ;Feld mit zwei Zahlen
0002 1C00
                            8
                            9     PUBLIC ZAHLEN_PAAR
                           10     EXTRN MITTEL_WERT:WORD
                           11
----                       12     DATEN_PROG ENDS
                           13
----                       14     CODE_PROG SEGMENT BYTE PUBLIC 'ROM_REGION'
                           15
0000 8BD0                  16     MITTEL: MOV     DX,AX                   ;(DX) = erste Zahl
0002 03D3                  17             ADD     DX,BX                   ;(DX) = erste+zweite Zahl
0004 D1DA                  18             RCR     DX,1                    ;(DX) = (AX+BX)/2
                           19                                             ;     = (erste+zweite Zahl)/2
0006 89160000        E     20     FERTIG: MOV     MITTEL_WERT,DX          ;Mittelwert extern speichern
                           21
----                       22     CODE_PROG ENDS
                           23
                           24             END
```

Im Modul BEISPIEL_47 wird ein externes Zahlenpaar vom Typ WORD mit `MOV AX, ZAHLEN_PAAR [SI]` und `MOV BX, ZAHLEN_PAAR [SI+2]` in die Register AX und BX geladen.

Im Modul BEISPIEL_48 wird aus den Registerwerten AX und BX der Mittelwert gebildet. Er ist im DX-Register verfügbar. Der Mittelwert wird mit `MOV MITTEL_WERT, DX` in der externen Variablen MITTEL_WERT gespeichert.

5.1.4.2 Dummy-Segmente

Manchmal liegt eine Situation vor, wie sie beispielhaft im folgenden Bild illustriert ist:



Das MODUL1 enthält die Segmente DATEN und CODE, wobei im DATEN-Segment die Word-Variable VAR definiert ist. Im MODUL2 ist lediglich das CODE-Segment enthalten, es existiert aber **kein** DATEN-Segment.

Es soll nun im MODUL2 mit `MOV AX,VAR` auf die externe Variable VAR in MODUL1 zugegriffen werden. Bei dieser Situation weiß der Assembler nicht, daß sich VAR im DATEN-Segment von MODUL1 aufhält. Daher kann der Assembler die Variable VAR nicht erreichen.

Dieses Problem kann nur gelöst werden, wenn im MODUL2 eine Verbindung zwischen DS und dem DATEN-Segment hergestellt wird. Eine solche Verbindung ist über ein sogenanntes '**Dummy**'-Segment realisierbar. In diesem Segment sind keinerlei Variablen definiert. Es ist also leer und wird daher manchmal als **Null**-Segment bezeichnet.

Wie ein 'Dummy'-Segment zur Realisierung einer gültigen Modul-Verbindung benützt werden kann, illustrieren die folgenden Bilder:

```
                NAME        MODUL 1
                PUBLIC      VAR

DATEN           SEGMENT     PUBLIC
VAR             DW          ?
DATEN           ENDS

CODE            SEGMENT     PUBLIC
  ASSUME CS : CODE , DS : DATEN
                .
                .
                MOV         AX, VAR
                .
                .
CODE            ENDS
  END
```

```
                NAME        MODUL2

DATEN           SEGMENT     PUBLIC
  EXTRN VAR : WORD
DATEN           ENDS

CODE            SEGMENT     PUBLIC
  ASSUME CS : CODE , DS : DATEN
                .
                .
                MOV         AX, VAR
                .
                .
CODE            ENDS
  END
```

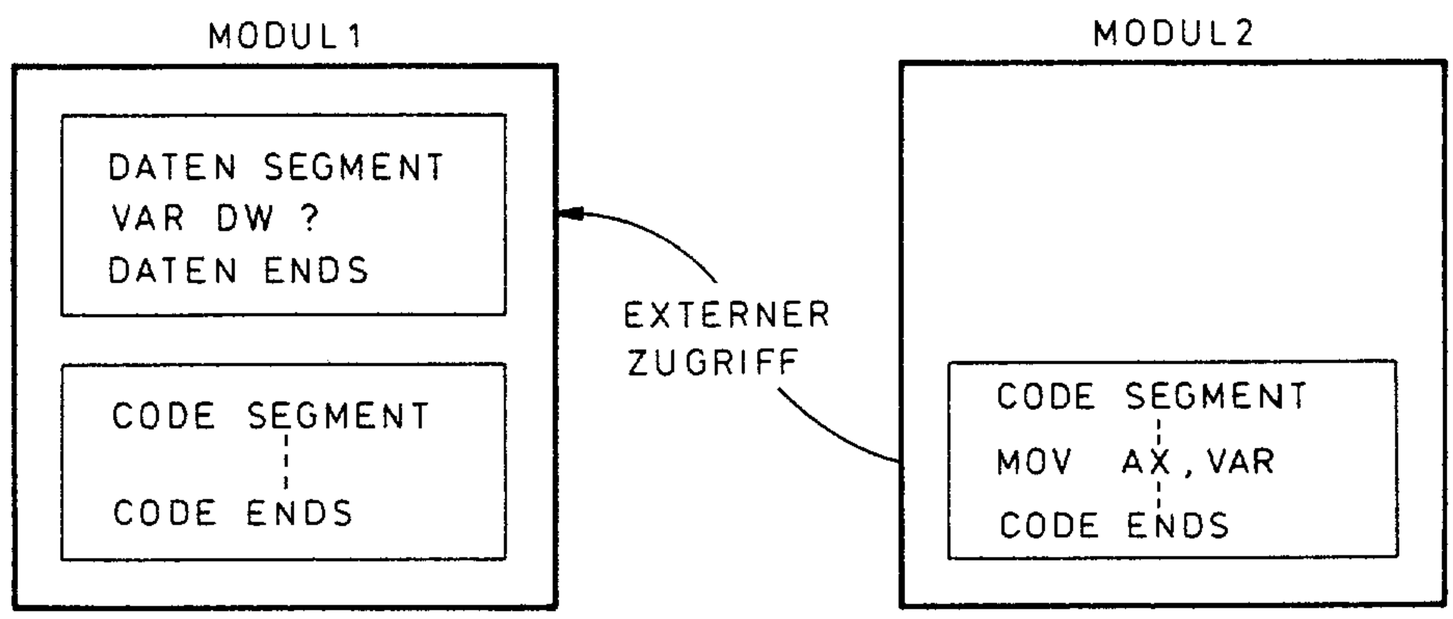
* Dummy-Segment

Man sieht, daß im MODUL2 das DATEN-Segment als 'Dummy'-Segment definiert ist. Gleichzeitig stellt `ASSUME DS:DATEN` die Verbindung zwischen DS und dem leeren DATEN-Segment her. Der Assembler kann nun wie im MODUL1 über `MOV AX,VAR` die Variable VAR problemlos erreichen.

Nach dem Binden der beiden Module sind die lokalen DATEN-Segmente zu einem **einzelnen** Segment kombiniert, wobei das resultierende Segment DS-relativ adressiert wird. Aus diesem Grund muß sich im MODUL2 die Anweisung `EXTRN VAR:WORD` **innerhalb** des DATEN-SEGMENT/ENDS-Paares aufhalten.

5.1.4.3 Der SEG-Operator

Wenn ein Programm aus einer Vielzahl von unterschiedlichen Segmenten besteht, liegt oft eine Situation vor, wie sie beispielhaft im folgenden Bild illustriert ist:



Das MODUL1 enthält die Segmente DATEN und CODE, wobei im DATEN-Segment die Word-Variable VAR definiert ist. Das MODUL2 enthält nur das CODE-Segment, wobei in diesem Modul mit `MOV AX,VAR` auf die externe Variable VAR zugegriffen werden soll. Bei dieser Situation weiß der Assembler nicht, daß sich VAR im DATEN-Segment von MODUL1 aufhält.

Es sei nun vorausgesetzt, daß für den Assembler diese Information verborgen bleibt. Der Zugriff auf VAR über ein 'Dummy'-Segment soll daher ausgeschlossen sein. Damit der Assembler aber trotzdem die externe Variable erreichen kann, hilft ihm der **SEG**-Operator. Er stellt in MODUL2 die Verbindung zwischen DS und dem externen DATEN-Segment in MODUL1 her.

Syntax:

```
SEG   varlab
```

Dabei ist **varlab** der Name einer Variablen oder eines Labels. Grundsätzlich läßt der SEG-Operator zwei Interpretationen zu. Die erste Interpretation hängt mit der ASSUME-Direktive zusammen. Hier kann SEG benützt werden, um dasjenige Segment zu spezifizieren, in dem sich die Variable oder der Label aufhalten.

Beispiel:

```
ASSUME DS:SEG VAR
```

Diese Anweisung teilt dem Assembler mit, daß DS auf ein Segment zeigt, in dem VAR definiert ist. Die zweite Interpretation des SEG-Operators hängt mit einem Segment-Selektor zusammen. In diesem Fall wird SEG benützt, um entweder den Selektor in einer Variablen zu speichern oder ein Segment-Register zu initialisieren.

Beispiel:

```
BEGIN   DW  SEG VAR         ;Speichere den Selektor
                            ;des Segments, in dem
                            ;VAR definiert ist

INIT:   MOV AX,SEG VAR      ;Initialisiere DS mit dem
        MOV DS,AX           ;Selektor des Segments, in
                            ;dem VAR definiert ist
```

Wie bei der oben beschriebenen Situation der SEG-Operator zur Realisierung einer gültigen Modul-Verbindung benützt werden kann, illustrieren die folgenden Bilder:

```
                NAME        MODUL 1
                PUBLIC      VAR

DATEN           SEGMENT     PUBLIC
VAR             DW          ?
DATEN           ENDS

CODE            SEGMENT     PUBLIC
  ASSUME CS : CODE , DS : DATEN

                .
                .
                MOV         AX, VAR
                .
                .
CODE            ENDS
  END
```

```
                NAME        MODUL 2

EXTRN VAR : WORD

CODE            SEGMENT     PUBLIC
  ASSUME CS : CODE , DS : SEG VAR

                .
                .
                MOV         AX, SEG VAR
                MOV         DS, AX
                MOV         AX, VAR
                .
                .
CODE            ENDS
  END
```

Wie zu sehen ist, wird dem Assembler in MODUL2 mit `ASSUME DS:SEG VAR` mitgeteilt, daß DS auf das DATEN-Segment in MODUL1 zeigt. Da sich VAR in diesem Segment aufhält, kann nun der Assembler die Variable problemlos erreichen.

Damit VAR auch vom Prozessor DS-relativ erreicht werden kann, muß vor der Ausführung des Befehls `MOV AX,VAR` das DS-Register mit dem Selektor des externen DATEN-Segments initialisiert werden. Dies erledigen die folgenden beiden Befehle:

```
MOV AX,SEG VAR        ;DS mit dem Selektor des
MOV DS,AX             ;externen DATEN-Segments
                      ;initialisieren
```

In MODUL2 können viele lokale Daten-Segmente (DATEN_1..DATEN_N) existieren, die alle nicht mit dem DATEN-Segment in MODUL1 kombinierbar sind. Daher hält sich VAR nach dem Binden in keinem dieser Segmente auf. Aus diesem Grund muß die Anweisung `EXTRN VAR:WORD` außerhalb jeden SEGMENT/ENDS-Paares plaziert werden.

5.1.5 GROUPS

Oft ist es wünschenswert **verschiedene** Segmente relativ zu einem gemeinsamen Registerwert zu adressieren. So könnte zum Beispiel die Daten- und Stack-Region eines Programms klein genug sein, um in ein physikalisches Segment zu passen.

Werden die beiden Regionen so kombiniert, daß sie relativ zu einer gemeinsamen Basisposition adressierbar sind, dann könnten die DS- und SS-Register mit identischen Werten geladen werden.

Um nun Variablen im Datenbereich, bzw. bestimmte Positionen im Stack zu adressieren, sind nur noch **Offsets** relativ zur gemeinsamen Basis notwendig. Die Kombination dieser unterschiedlichen Programm-Regionen, wird durch Anwendung der **GROUPS** erreicht.

Eine **GROUP** ist ein kombinierter Satz von Programm-Regionen, die relativ zu einer gemeinsamen Basisposition adressierbar sind. Die individuellen Regionen innerhalb einer GROUP, wie z.B. DATEN_PROG und STACK_PROG sind benachbart.

In der Tat, ist eine GROUP eine "Kombination von Kombinationen", da eine GROUP sich aus individuellen Regionen zusammensetzt, die ihrerseits aus individuellen logischen Segmenten bestehen.

Das GROUP-Statement teilt dem Assembler (MASM und ASM86, **nicht** ASM286; siehe Kapitel 13) mit, daß zwei oder mehrere unterschiedliche Regionen kombiniert werden sollen.

Syntax des GROUP-Statements:

```
Group Name   GROUP   Segmentname [,Segmentname] ..
```

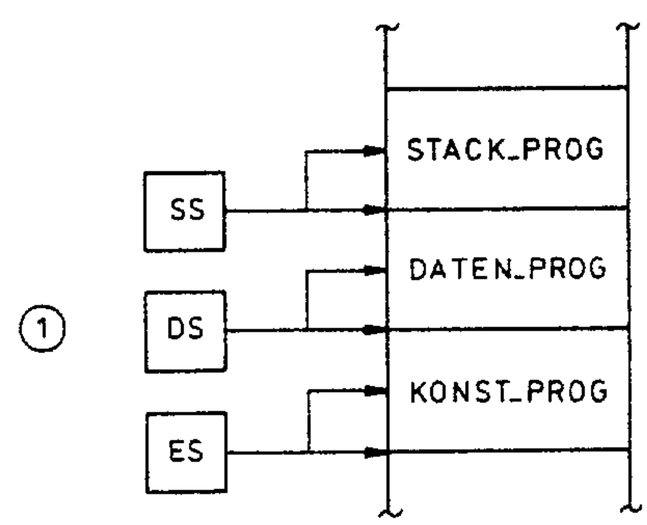
- **Group Name** ist der metasprachliche Ausdurck für ein GROUP.
- **GROUP** ist ein Schlüsselwort und muß angegeben werden.
- **Segmentname** sind die metasprachlichen Ausdrücke bereits kombinierter logischer Segmente, die zu GROUPS kombiniert werden sollen.

Ein einfaches Beispiel soll das Konzept der GROUPS erläutern. Es sei vorausgesetzt, daß drei unterschiedliche Daten-Regionen existieren:

- **Read-Only-Daten** (Konstante) in KONST_PROG Segmenten (Combine-Typ PUBLIC)
- **Read-Write-Daten** (Variablen) in DATEN_PROG-Segmenten (Combine-Typ PUBLIC)
- einen **Stack**, der aus STACK_PROG-Segmenten zusammengesetzt ist. (Combine-Typ STACK)

Weiterhin soll gelten, daß alle drei Regionen in ein 64 KByte physikalisches Segment passen.

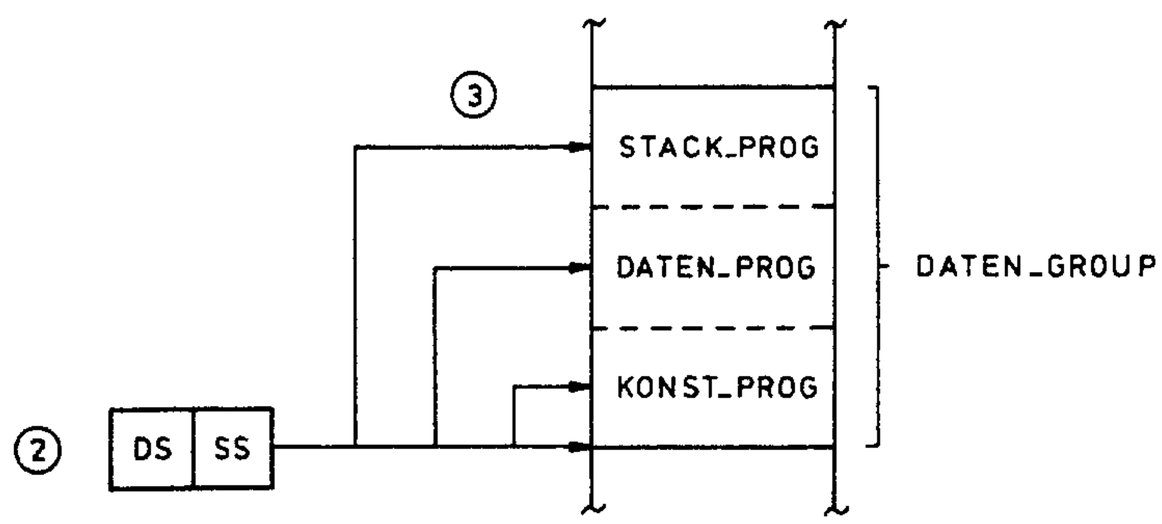
Wenn keine GROUPS verwendet werden, wird jede Region, also KONST_PROG, DATEN_PROG und STACK_PROG, ihre eigene Basis haben.



Für den Fall daß Datenzugriffe erforderlich sind, kann ohne Probleme DS für DATEN_PROG, ES für KONST_PROG und SS für STACK_PROG benützt werden ①.

Soll jedoch die **Adresse** einer Variablen gespeichert werden, sind zwei Worte notwendig, nämlich die Basis- und die Offsetkomponente der Variablenadresse.

Es sei nun vorausgesetzt, daß die Regionen STACK_PROG, DATEN_PROG und KONST_PROG eine gemeinsame Basis haben, also zu einer GROUP kombiniert sind.



In diesem Fall ergeben sich folgende Vereinfachungen:

- Die DS- und SS-Register können mit dieser Basis geladen werden ②, so daß alle drei Regionen mit Hilfe der verfügbaren Adressierungsarten und unter Berücksichtigung der Default-Segment-Register erreichbar sind ③.
- Der Offset einer Variablen kann nun benützt werden, um ihre Adresse zu kennzeichnen, zumal bekannt ist, daß der Wert in SS und DS der Basisteil der Adresse ist.

Folgende Anweisung kombiniert die Regionen STACK_PROG, DATEN_PROG und KONST_PROG zu einer GROUP:

```
DATEN_GROUP   GROUP   KONST_PROG, DATEN_PROG, STACK_PROG
```

GROUP Name Schlüsselwort Segmentnamen

Ähnlich wie ein **Segmentname**, der sich auf die Basis kombinierter logischer Segmente bezieht, wird der **GROUP-Name** benützt, um einen Bezug zur Basis der GROUP herzustellen.

Hinweis: Die Basis einer GROUP ist im Allgemeinen nicht identisch mit der Basis eines Segments innerhalb der GROUP.

Werden GROUPS benützt, müssen die Segment-Register mit dem GROUP-Namen und **nicht** mit Segmentnamen initialisiert werden ① (siehe folgendes Programmbeispiel). Die GROUP-Namen müssen auch in der ASSUME-Anweisung für Segment-Register, die die Basis einer GROUP enthalten, angegeben werden ②.

```
LOC  OBJ                    LINE     SOURCE

                              1              NAME    BEISPIEL_49
                              2
                              3      DATEN_GROUP GROUP DATEN_PROG,STACK_PROG
                              4
                              5      ASSUME CS:CODE_PROG,DS:DATEN_GROUP        ②
                              6
----                          7      DATEN_PROG SEGMENT WORD 'RAM_REGION'
                              8
0000 (1                       9      VAR     DW      1 DUP (?)
     ????
     )
                             10
----                         11      DATEN_PROG ENDS
                             12
----                         13      STACK_PROG SEGMENT WORD 'RAM_REGION'
                             14
0000 (5                      15              DW      5 DUP (?)
     ????
     )
                             16
----                         17      STACK_PROG ENDS
                             18
----                         19      CODE_PROG SEGMENT BYTE 'ROM_REGION'
                             20
0000 B8----          R       21      START:  MOV     AX,DATEN_GROUP          ;DS Register und
0003 8ED8                    22              MOV     DS,AX                   ;SS Register mit gemeinsamer    ①
0005 8ED0                    23              MOV     SS,AX                   ;Basis initialisieren
                             24
----                         25      CODE_PROG ENDS
                             26
                             27              END     START
```

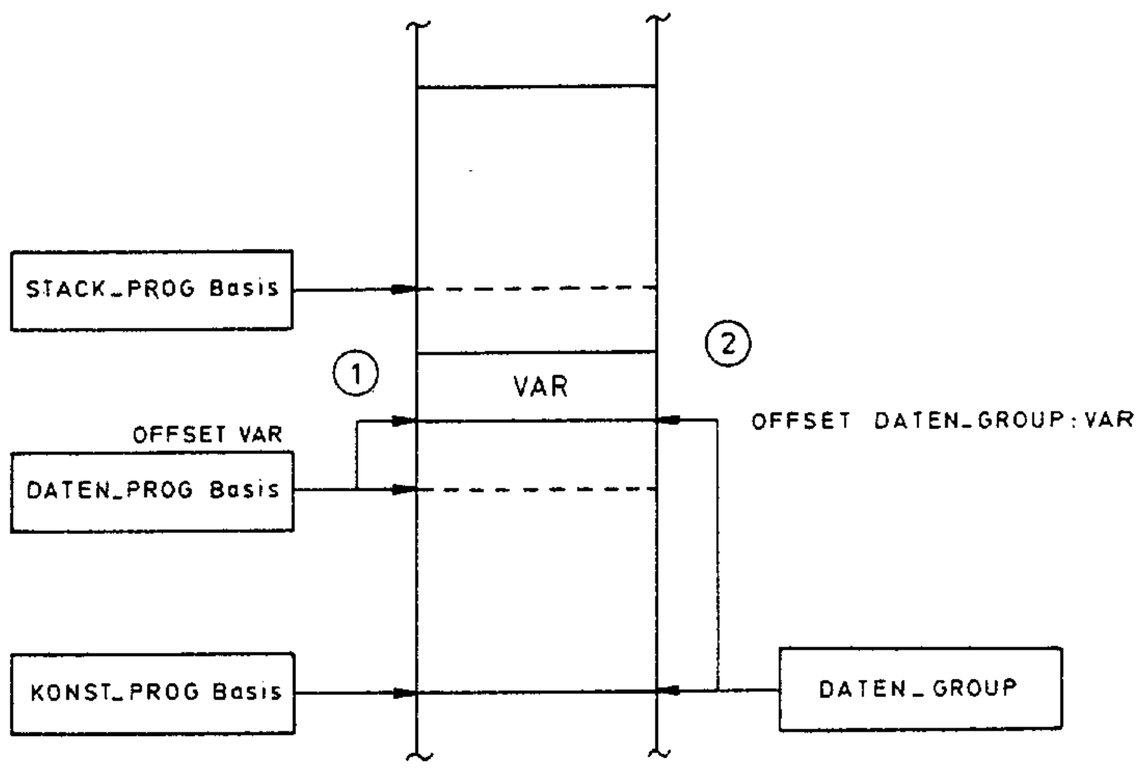
Hinweis: Im angegebenen Programmierbeispiel ist `ASSUME SS:NOTHING` vorausgesetzt.

Der Abstand einer Variablen oder eines Labels zur jeweiligen Segmentbasis und der Abstand zur GROUP-Basis sind verschieden. Der OFFSET-Operator liefert den Abstand einer Variablen oder eines Labels zur jeweiligen Segmentbasis.

Wenn jedoch GROUPS benützt werden, wird der Abstand zur GROUP-Basis gebraucht. Dies wird spezifiziert, wenn zum Symbol, auf das zugegriffen werden soll, ein sogenannter **Group Override** hinzugefügt wird.

Der Group Override ist durch den **GROUP-Namen**, gefolgt von einem Doppelpunkt (:) vor dem Namen einer Variablen oder eines Labels gekennzeichnet.

Beispiel: VAR ist eine Variable im DATEN_PROG-Segment, wobei DATEN_PROG ein Segment in der GROUP DATEN_GROUP ist.



Ein Zugriff zur Variablen VAR relativ zur Basis des DATEN_PROG-Segments ist mit `OFFSET VAR` möglich ①. Soll aber ein Zugriff zur Variablen VAR relativ zur Basis von DATEN_GROUP erfolgen, so ist dies mit `OFFSET DATEN_GROUP: VAR` möglich (DATEN_GROUP = Group Override) ②. Im angegebenen Programmierbeispiel existieren die logischen Segmente CODE_PROG, DATEN_PROG und STACK_PROG.

```
LOC  OBJ                    LINE     SOURCE

                              1              NAME    BEISPIEL_50
                              2
                              3      DATEN_GROUP GROUP DATEN_PROG,STACK_PROG      ①
                              4
                              5      ASSUME CS:CODE_PROG,DS:DATEN_GROUP
                              6
----                          7      DATEN_PROG SEGMENT WORD 'RAM_REGION'
                              8
0000 (1                       9      VAR     DW      1 DUP (?)
     ????
     )
                             10
----                         11      DATEN_PROG ENDS
                             12
----                         13      STACK_PROG SEGMENT WORD 'RAM_REGION'
                             14
0000 (5                      15              DW      5 DUP (?)
     ????
     )
000A                         16      TOP_STACK LABEL WORD
                             17
----                         18      STACK_PROG ENDS
                             19
----                         20      CODE_PROG SEGMENT BYTE 'ROM_REGION'
                             21
0000 B8----          R       22      START:  MOV     AX,DATEN_GROUP                  ;DS Register und
0003 8ED8                    23              MOV     DS,AX                           ;SS Register mit gemeinsamer ②
0005 8ED0                    24              MOV     SS,AX                           ;Basis initialisieren
0007 BC0A00          R       25              MOV     SP,OFFSET DATEN_GROUP:TOP_STACK ;SP Register initialisieren ③
                             26
----                         27      CODE_PROG ENDS
                             28
                             29              END     START
```

Die Segmente DATEN_PROG und STACK_PROG sind zur GROUP DATEN_GROUP zusammengefaßt ①.

Die gemeinsame Basis der beiden Segmente ist DATEN_GROUP, d.h. das DS-Register und das SS-Register werden mit DATEN_GROUP initialisiert ②.

Da die Segmente DATEN_PROG und STACK_PROG zu einer GROUP kombiniert werden, ist das SP-Register mit dem Offset relativ zur Basis der GROUP zu laden. Zum Label TOP_STACK ist daher als **Group Override** der Name der GROUP, nämlich DATEN_GROUP hinzuzufügen.

Der Befehl `MOV SP, OFFSET DATEN_GROUP: TOPSTACK` lädt dann das SP-Register mit dem Offset von TOP_STACK ③.