Kapitel 8

Kombinieren von Assembler- und C-Modulen

(am Beispiel von ASM86 und iC86 Version 4.0)

Wie bereits gezeigt, stellt der Assembler Direktiven zur Verfügung, um Segmente zu kombinieren oder GROUPs zu bilden. Es existieren nun Regeln, die Auskunft darüber geben, **wie** Segmente zu kombinieren und möglicherweise zu GROUPs zusammenzufassen sind.

Zu diesem Zweck stellen höhere Programmiersprachen wie PL/M, FORTRAN und C einige **Modelle** zur Verfügung, die vom Anwenderprogrammierer ausgewählt werden können. Das ausgewählte Modell bestimmt darüber, welche Assembler-Segment- und GROUP-Anweisungen zu verwenden sind.

Weiterhin hat das Modell Auswirkungen auf das Prozedur-Interface, z.B. darauf, ob ein Short-Pointer (Offset) oder ein Long-Pointer (Base:Offset) an die Prozedur übergeben werden soll.

8.1 Das SMALL-Modell

Beim SMALL-Modell liegt die Code-Region in **einem einzelnen** physikalischen Segment und die Daten- zusammen mit der Stack-Region in einem **anderen**.

Dieses Modell wird benützt, wenn für die Programme nicht mehr als 64 KByte Code und nicht mehr als 64 Kbyte kombinierte Daten- und Stack-Region erforderlich sind.

Der Vorteil des SMALL-Modells ist, daß alle Pointer nur Offsets sind. Das Segment-Register CS ist nicht veränderlich, so daß sich bei JMP- und CALL-Befehlen nur der Instruction-Pointer IP ändert.

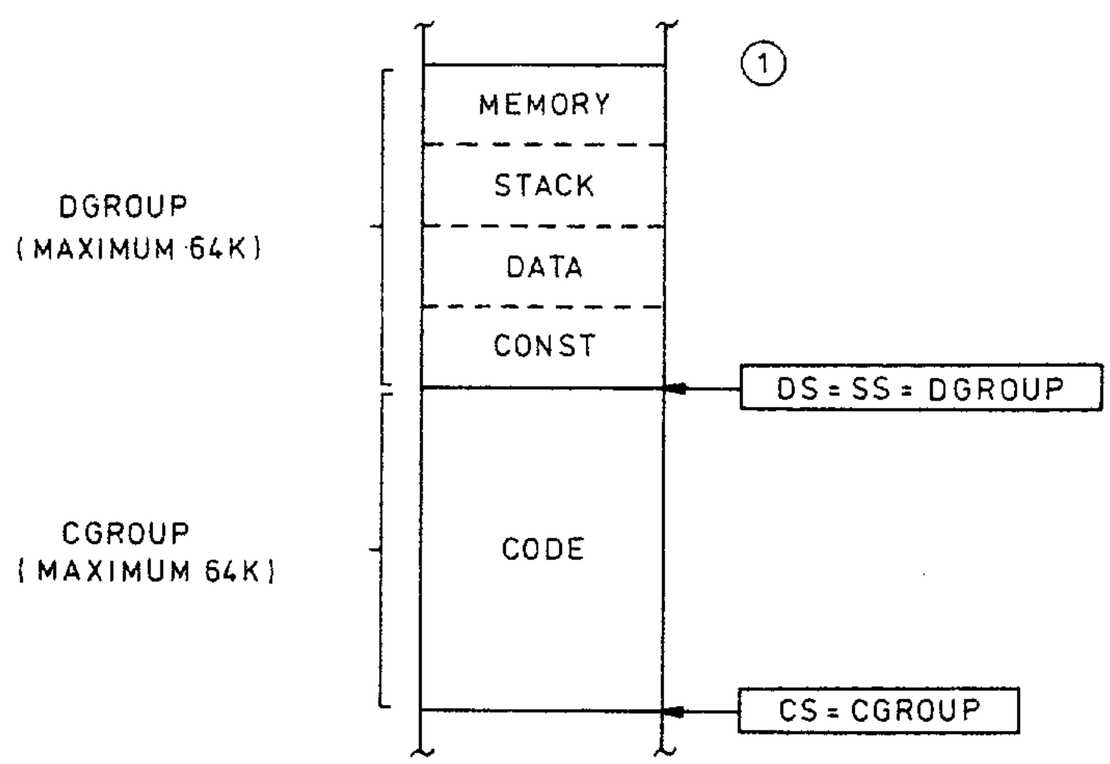
Weiterhin sind DS und SS ebenfalls fest und mit dem gleichen Wert initialisiert. Das bedeutet, daß nur Offsets notwendig sind, um eine Variablenadresse oder einen Eintrag im Stack zu spezifizieren.

Das SMALL-Modell bietet im Vergleich zu den anderen HLL-Modellen (High Level Langage) den komprimiertesten Code.

8.1.1 CGROUP und DGROUP

Der Code eines SMALL-Programms wird in einem Segment hinterlegt, das mit **CODE** bezeichnet wird. Die Daten gehören zu einem Segment mit dem Namen **DATA** und der Stack zu einem Segment mit dem Namen **STACK**.

Weiterhin stehen die Segmente **CONST** und **MEMORY** zur Verfügung. Sie können benützt werden, um spezielle Datenwerte aufzunehmen. Da diese Segmente selten von Assembler-Modulen benützt werden, sollen sie hier nicht weiter diskutiert werden. Mehr Informationen zur Plazierung von **Konstanten** siehe **Anhang**.



Das SMALL-Programm ist in zwei GROUPs unterteilt, für die folgende Regel gilt:

"Die Code-Region ist in einem physikalischen Segment und die Stack- und Daten-Region in einem anderen hinterlegt."

Dabei ist das Code-Segment nur Mitglied der **CGROUP**, also einer GROUP deren Basiswert in CS steht. Die MEMORY_STACK_DATA- und CONST-Segmente sind Mitglieder der **DGROUP**, einer GROUP deren Basiswert sowohl in DS als auch in SS steht. Das angegebene Bild ① zeigt die Speicherbelegung (Memory Layout) eines SMALL-Programms.

In Anlehnung an das 'Memory Layout' eines Small-Programms kann für ein Assembler-Programm folgendes 'Rahmenwerk' angegeben werden; es stimmt mit dem eines C-Small-Programms überein:

```
CGROUP GROUP CODE
DGROUP GROUP CONST, DATA, STACK, MEMORY

ASSUME CS:CGROUP, DS:DGROUP, SS:DGROUP

CONST SEGMENT PUBLIC 'CONST'
  <Hier können Programm-Konstanten definiert werden>
CONST ENDS

DATA SEGMENT PUBLIC 'DATA'
EXTRN   <externe Variablen>
  <Hier sind die Programm-Variablen zu definieren>
DATA ENDS

STACK SEGMENT STACK 'STACK'
  <Hier ist das DW-Statement zu benützen, um die Länge des
  'Run Time' Stack festzulegen>
STACK ENDS

MEMORY SEGMENT MEMORY 'MEMORY'
  <Hier können spezielle Programmdaten oberhalb der anderen
  Segmente definiert werden>
MEMORY ENDS

CODE SEGMENT PUBLIC 'CODE'
EXTRN <externe NEAR Labels wie z.B. Prozedur-Namen>
  <Hier wird der Programm-Code hinterlegt>
CODE ENDS

END <Hier kann für ein Hauptmodul die Startadresse angegeben
    werden>
```

Das 'Rahmenwerk' zeigt exakt, welche SEGMENT-Statements für CODE, DATA, STACK und eventuell CONST- und MEMORY-Segmente und welche GROUP- und ASSUME-Statements in einem Modul angegeben werden müssen.

Die Segment-**Align-Typ**-Attribute können sich vom PARA-'Default'-Typ unterscheiden. Allerdings müssen die **Combine-Typ**- und 'Class-Name'-Attribute gemäß dem 'Rahmenwerk' angegeben werden.

8.1.2 Register-Initialisierung und Offsets im Small-Modell

Ist der Hauptmodul eines Small-Programms in Assembler geschrieben, ist darauf zu achten, daß die Segment-Register und SP richtig initialisiert sind. Dabei muß CS zur CGROUP zeigen. Das bedeutet, die END-Anweisung im Hauptmodul muß als Startadresse einen Label im Code-Segment benützen, das in der CGROUP liegt.

Sowohl DS als auch SS müssen den DGROUP-Basiswert enthalten. Das SP-Register muß mit dem Offset der Stack-Regionspitze relativ zur DGROUP initialisiert werden. Das folgende Beispiel zeigt, wie ein Assembler-Hauptmodul die Register für ein Small-Programm initialisiert.

```
LOC  OBJ                    LINE     SOURCE

                               1             NAME    BEISPIEL_80
                               2
                               3     CGROUP GROUP CODE
                               4     DGROUP GROUP DATA,STACK
                               5
                               6     ASSUME CS:CGROUP,DS:DGROUP
                               7
----                           8     DATA SEGMENT PUBLIC 'DATA'
                               9
0000 00                       10     VAR_1   DB      0,1,2,3
0001 01
0002 02
0003 03
                              11
----                          12     DATA ENDS
                              13
----                          14     STACK SEGMENT STACK 'STACK'
                              15
0000 (4                       16             DW      4 DUP (?)
     ????
     )
0008                          17     TOP_STACK LABEL WORD
                              18
----                          19     STACK ENDS
                              20
----                          21     CODE SEGMENT PUBLIC 'CODE'
                              22
0000 B8----          R        23     START:  MOV     AX,DGROUP                       ;DS Register initialisieren
0003 8ED8                     24             MOV     DS,AX
0005 8ED0                     25             MOV     SS,AX                           ;SS Register initialisieren
0007 BC0800          R        26             MOV     SP,OFFSET DGROUP:TOP_STACK      ;SP Register initialisieren
000A BB0000          R        27             MOV     BX,OFFSET DGROUP:VAR_1          ;BX = Zeiger zum Feld VAR_1
                              28                     ;
                              29                     ;
----                          30     CODE ENDS
                              31
                              32             END     START
```

Da die Segment-Register die Basiswerte von GROUPs enthalten, ist es notwendig in SMALL-Modulen immer einen GROUP-Override-Operator zu benützen. So wird der Offset der Stack-Regionspitze durch den Ausdruck `OFFSET DGROUP:TOP_STACK` und nicht durch `OFFSET TOP_STACK` angegeben.

Ähnliches gilt für die Initialisierung des BX-Registers mit dem Offset von VAR_1. Da das DATA-Segment Mitglied der DGROUP ist, kann der Offset von VAR_1 nur durch den Ausdruck `OFFSET DGROUP:VAR_1` angegeben werden.

Offsets von Variablen liegen immer relativ zu DGROUP, während Offsets von Labels immer relativ zu CGROUP liegen. Wird im **Initialisierungsfeld** einer Datenzuweisung der Name einer Variablen oder eines Labels angegeben, so ist auch hier der GROUP-Override-Operator zu benützen. Das folgende Beispiel soll dies verdeutlichen.

```
LOC  OBJ                  LINE     SOURCE

                            1              NAME    BEISPIEL_81
                            2
                            3      CGROUP GROUP CODE
                            4      DGROUP GROUP DATA
                            5
                            6      ASSUME CS:CGROUP,DS:DGROUP
                            7
----                        8      DATA SEGMENT PUBLIC 'DATA'
                            9
0000 0000            E     10      NEAR_LABEL      DW      CGROUP:ZIEL      ①     ;Offset von ZIEL in CGROUP
                           11
----                       12      DATA ENDS
                           13
----                       14      CODE SEGMENT PUBLIC 'CODE'
                           15
                           16      EXTRN ZIEL:NEAR
                           17
0000 B8----          R     18      START:  MOV     AX,DGROUP                        ;DS Register initialisieren
0003 8ED8                  19              MOV     DS,AX
0005 BB0000          R     20              MOV     BX,OFFSET DGROUP:NEAR_LABEL      ;BX = Zeiger zur Variablen
                           21                                                       ;NEAR_LABEL
0008 FF27                  22              JMP     WORD PTR [BX]          ②         ;Sprung nach CGROUP:ZIEL
                           23                      ;
                           24                      ;
----                       25      CODE ENDS
                           26
                           27              END     START
```

```
LOC  OBJ                  LINE     SOURCE

                            1              NAME    BEISPIEL_82
                            2
                            3      CGROUP GROUP CODE
                            4
                            5      ASSUME CS:CGROUP
                            6
----                        7      CODE SEGMENT PUBLIC 'CODE'
                            8                      ;
                            9                      ;
0000 8BC3                  10      ZIEL:   MOV     AX,BX
                           11                      ;
                           12                      ;
----                       13      CODE ENDS
                           14
                           15              END
```

Die Variable NEAR_LABEL ist durch `DW CGROUP:ZIEL` ① für einen intrasegment-indirekten Sprungbefehl (`JMP WORD PTR [BX]` ②) mit dem Label

ZIEL initialisiert. Diese Zuweisung wird eine korrekte CGROUP-relative Zieladresse liefern. Im Gegensatz dazu wäre die Zuweisung `NEAR_LABEL DW ZIEL` **unkorrekt**.

8.1.3 Das Prozedur-Interface beim SMALL-Modell

Da sich beim SMALL-Modell das CS-Register nicht ändert, müssen alle Labels und Prozeduren als NEAR deklariert werden. Dies bedeutet, daß sich nur IP ändert, wenn JMP- und CALL-Befehle ausgeführt werden. Daher sind Rücksprungadressen Worte, die einen IP-Wert kennzeichnen.

Zeiger (Pointer) werden im SMALL-Modell ebenfalls durch Worte dargestellt. So ist ein Zeiger auf eine Variable (Variablen-Adresse) in der Daten-Region oder im Stack durch seinen Offsetwert relativ zu DGROUP gekennzeichnet. Die Adresse eines Labels in der Code-Region entspricht seinem Offsetwert relativ zu CGROUP. Damit ist ein Zeiger, der an eine Prozedur übergeben wird, ein Wort, das einen Offsetwert relativ zu DGROUP oder CGROUP darstellt.

Ergebnisse, die eine C-Funktion im SMALL-Modell liefert, werden in folgenden CPU-Registern hinterlegt:

- **char** in AL
- **int** in AX
- **long** in DX:AX
- **float** im 8087-Stack
- **double** im 8087-Stack
- **struct** Zeiger in AX
- **union** Zeiger in AX
- **pointer** in AX

8.1.4 Erweiterung des SMALL-Modells

Es gibt Anwendungen, bei denen eine 64 KByte große Code-Region und eine 64 KByte große kombinierte Daten- und Stack-Region nicht ausreichen. Gewöhnlich bedeutet dies, daß eines der anderen, weniger effizienten Modelle, benützt werden müssen.

Als weitere Modelle stehen zur Verfügung:

- Das COMPACT-Modell, viel Daten, wenig Code
- das MEDIUM-Modell und viel Code, wenig Daten
- das LARGE-Modell

Möchte man, trotz der Einschränkungen, auf das SMALL-Modell nicht verzichten, ist jedoch eine Erweiterung möglich.

Als Beispiel sei vorausgesetzt, daß ein Programm existiert, dessen Code in ein 64 KByte großes Segment paßt. Das Programm soll eine große Daten-Tabelle haben, so daß die 64 KByte große kombinierte Daten- und Stack-Region überschritten wird.

In diesem Fall kann in einem Assembler-Modul ein zusätzliches Datensegment **außerhalb** von DGROUP definiert werden. D.h. die Daten-Tabelle kann in ein eigenes Segment, z.B. in das EXTRA_DATEN-Segment gelegt werden.

Um die Eintragungen in diesem Segment zu erreichen, kann der Assembler-Code das ES-Register benützen. Wenn nun ein C-Code auf die Daten-Tabelle zugreifen möchte, kann er entsprechende Assembler-Routinen benützen. Das folgende Programm-Beispiel macht von dieser Möglichkeit Gebrauch.

8.1.5 Programmierbeispiel: Modulkombination im SMALL-Modell

Das folgende Kombinationsbeispiel wurde mit ASM86 und dem INTEL ANSI-C-Compiler iC86 durchgeführt.

Es sollen das **C-Modul B83** und das **Assembler-Modul BEISPIEL_84** nach dem SMALL-Modell zu **einem** Modul kombiniert werden. Dabei ruft das C-Modul die Assembler-NEAR-Prozedur erzeuge_eintrag auf und übergibt ihr einen Index-Wert und einen Zeiger auf einen Zeichen-String, der im C-Modul definiert ist. Diesen Zeichen-String liest die Assembler-Prozedur und speichert ihn zusammen mit dem Index-Wert in einer Daten-Tabelle im EXTRA_DATEN-Segment.

Anschließend ruft das C-Programm die Assembler-NEAR-Prozedur nachsehen_eintrag auf und übergibt ihr den gleichen Index-Wert, den sie vorher an die Prozedur erzeuge_eintrag übergeben hat. Die Prozedur nachsehen_eintrag schreibt den vorher übertragenen Zeichen-String zusammen mit dem Index-Wert in den Character-Vektor eintrag_puffer des C-Moduls zurück. Das C-Modul könnte nun anschließend überprüfen, ob die Übertragung des Zeichen-Strings erfolgreich war. Wie die beiden Module miteinander korrespondieren, wird nun in den folgenden Abschnitten erläutert.

C-Modul:

```
char string_1[]={'E','I','N','G','A','B'};
char index_wert;
char eintrag_puffer[8];
beispiel_83()
{
alien erzeuge_eintrag();
alien nachsehen_eintrag();

erzeuge_eintrag(index_wert,string_1);
nachsehen_eintrag(index_wert);
}
```

Da sich die C-Funktionen erzeuge_eintrag und nachsehen_eintrag nach der Standard-Konvention verhalten sollen, sind sie mit dem Attribut **alien** entsprechend gekennzeichnet.

```
LOC  OBJ                  LINE     SOURCE

                             1             NAME    BEISPIEL_84
                             2
                             3     CGROUP  GROUP   CODE
                             4     DGROUP  GROUP   DATA,STACK
                             5
                             6     ASSUME CS:CGROUP,DS:DGROUP,SS:DGROUP
                             7
----                         8     EXTRA_DATEN SEGMENT
                             9
0000 (256                   10         DB  256 DUP (8 DUP (0))                    ;Daten-Tabelle mit
      (8
     00
      )
     )
                            11                                                    ;256 Acht-Byte-Eintragungen
----                        12     EXTRA_DATEN ENDS
                            13
----                        14     DATA SEGMENT PUBLIC 'DATA'
                            15
```

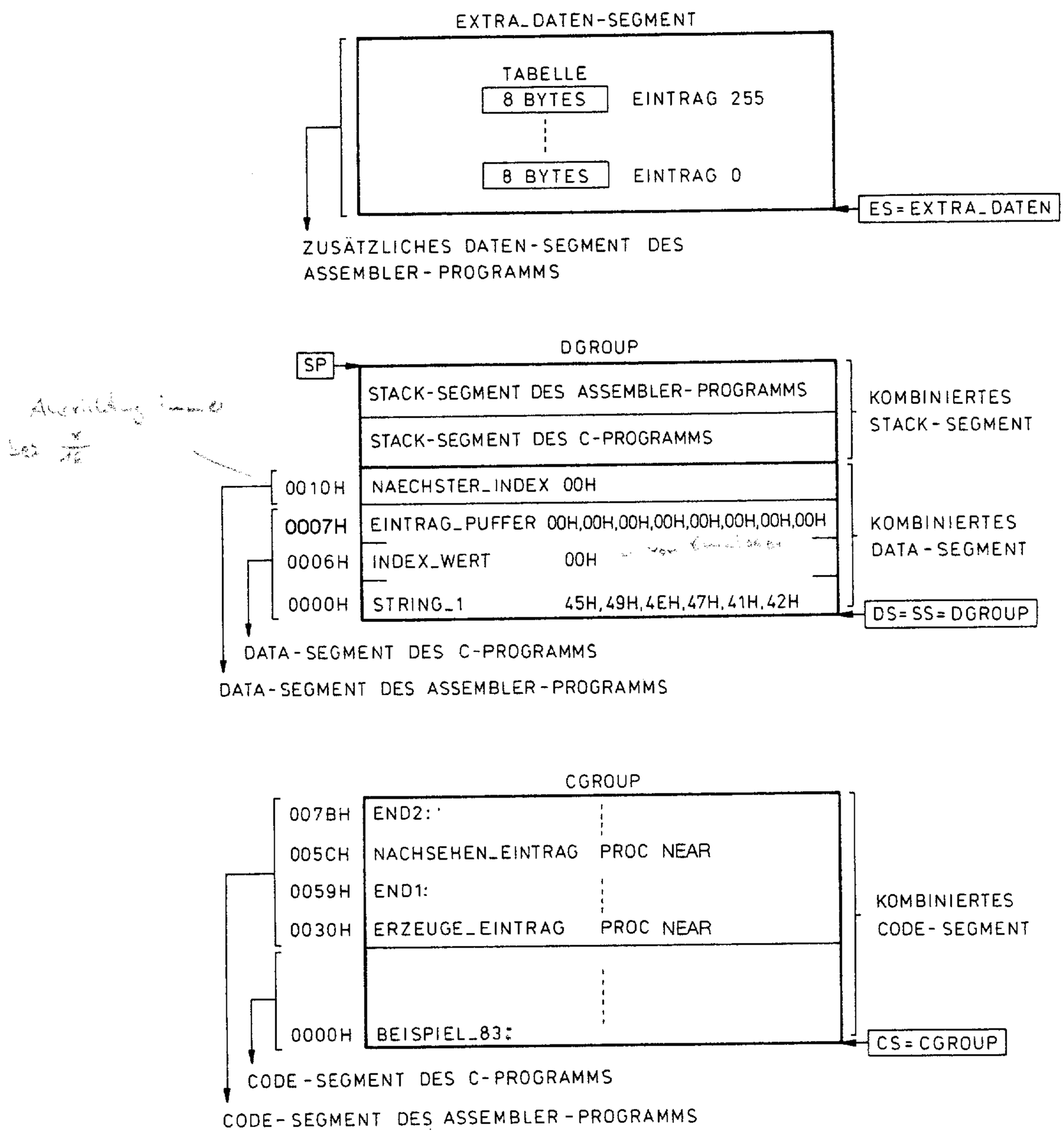
```
LOC  OBJ                  LINE     SOURCE

0000 00                     16     NAECHSTER_INDEX DB 0                          ;Index des naechsten Eintrags
                            17     EXTRN eintrag_puffer:BYTE
                            18
----                        19     DATA ENDS
                            20
----                        21     STACK SEGMENT STACK 'STACK'
0000 (4                     22         DW  4 DUP (?)
     ????
     )
----                        23     STACK ENDS
                            24
----                        25     CODE SEGMENT PUBLIC 'CODE'
                            26
                            27     PUBLIC erzeuge_eintrag,nachsehen_eintrag
                            28
0000                        29     erzeuge_eintrag PROC NEAR
                            30
0000 55                     31             PUSH    BP                            ;Prolog
0001 8BEC                   32             MOV     BP,SP
0003 B8----          R      33             MOV     AX,EXTRA_DATEN                ;ES=Basis des
0006 8EC0                   34             MOV     ES,AX                         ;   EXTRA_DATEN-Segments
0008 A00000          R      35             MOV     AL,NAECHSTER_INDEX            ;AX=augenblicklicher
000B B400                   36             MOV     AH,0                          ;   Indexwert
000D 8BF8                   37             MOV     DI,AX
000F C1E703                 38             SHL     DI,3                          ;DI=8*NAECHSTER_INDEX
                            39                                                   ;   Damit zeigt ES:DI zum
                            40                                                   ;   naechsten Eintrag
0012 8B5E06                 41             MOV     BX,[BP+6]                     ;BX=index_wert (C-Variable)
0015 26891D                 42             MOV     ES:[DI],BX                    ;index_wert im naechsten
                            43                                                   ;Eintrag der Daten-Tabelle
                            44                                                   ;speichern
0018 83C702                 45             ADD     DI,2                          ;ES:DI zeigt zum String-Teil
                            46                                                   ;des Eintrags
001B 8B7604                 47             MOV     SI,[BP+4]                     ;SI=Offset von string_1 (C-
                            48                                                   ;   Variable). DS:SI zeigt
                            49                                                   ;zum uebertragenden String
001E B90300                 50             MOV     CX,3                          ;Der String ist 3 Worte
                            51                                                   ;(6 Bytes) lang
0021 FC                     52             CLD                                   ;Direction-Flag=0 bedeutet
                            53                                                   ;SI und DI inkrementieren
0022 F3                     54             REP MOVS ES:WORD PTR[DI],DS:[SI]      ;Uebertrage String in die
0023 A5
                            55                                                   ;Daten-Tabelle
0024 FE060000        R      56             INC     NAECHSTER_INDEX               ;Naechsten Eintrag in der
                            57                                                   ;Daten-Tabelle indizieren
0028 5D                     58             POP     BP                            ;Epilog
0029 C20400                 59     END_1:  RET     4                             ;Entferne die Parameter
                            60                                                   ;index_wert und string_1 aus
                            61                                                   ;dem Stack und zurueck zum
                            62                                                   ;C-Programm
                            63     erzeuge_eintrag ENDP
                            64
002C                        65     nachsehen_eintrag PROC NEAR
                            66
002C 58                     67             POP     AX                            ;AX=Rueckspungadresse (IP)
002D 5B                     68             POP     BX                            ;BX (BL)=Parameter index_wert
                            69                                                   ;(0..255)
002E 50                     70             PUSH    AX                            ;Rueckspungadr. (IP) in den
                            71                                                   ;Stack zurueckschreiben
002F B8----          R      72             MOV     AX,EXTRA_DATEN                ;ES=Basis des
0032 8EC0                   73             MOV     ES,AX                         ;EXTRA_DATEN-Segments
0034 B700                   74             MOV     BH,0                          ;ES:BX zeigt auf  zurueckzu-
0036 C1E303                 75             SHL     BX,3                          ;lesenden Eintrag
0039 BF0000                 76             MOV     DI,0                          ;DI=Offset des ersten Worts
                            77                                                   ;   im Eintrag
003C B90400                 78             MOV     CX,4                          ;Es werden 4 Worte (8 Bytes)
                            79                                                   ;uebertragen
003F                        80     BEKOMMEN_EINTRAG:
003F 268B01                 81             MOV     AX,ES:[BX][DI]                ;Wort von  Daten-Tabelle im
                            82                                                   ;EXTRA_DATEN-Segment lesen
0042 89850000        E      83             MOV     WORD PTR eintrag_puffer[DI],AX ;Wort in  C-Character-Vektor
                            84                                                   ;eintrag_puffer uebertragen
0046 83C702                 85             ADD     DI,2                          ;Naechstes Wort adressieren
```

```
LOC  OBJ                LINE     SOURCE

0049 E2F4                 86              LOOP    BEKOMMEN_EINTRAG              ;Wiederhole, bis 8 Bytes
                          87                                                    ;(4 Worte) uebertragen sind
004B C3                   88     END_2:   RET                                  ;Zurueck zum C-Programm
                          89
                          90     nachsehen_eintrag ENDP
----                      91     CODE     ENDS
                          92              END
```

Das Assembler-Programm enthält die Befehle `SHL DI,3` und `SHL BX,3`. Diese beiden Befehle stehen nur beim 80186 und 80286 zur Verfügung.

Das Programm muß daher mit der Steuer-Anweisung MOD186, also `>ASM86 B84.SRC MOD186` übersetzt werden.

Nach der Kombination der beiden Module mit `>LINK86 B83.OBJ,B84.OBJ TO B84.LNK` ergibt sich folgendes Speicher-Layout:



Es ist zu erkennen, daß der Linker

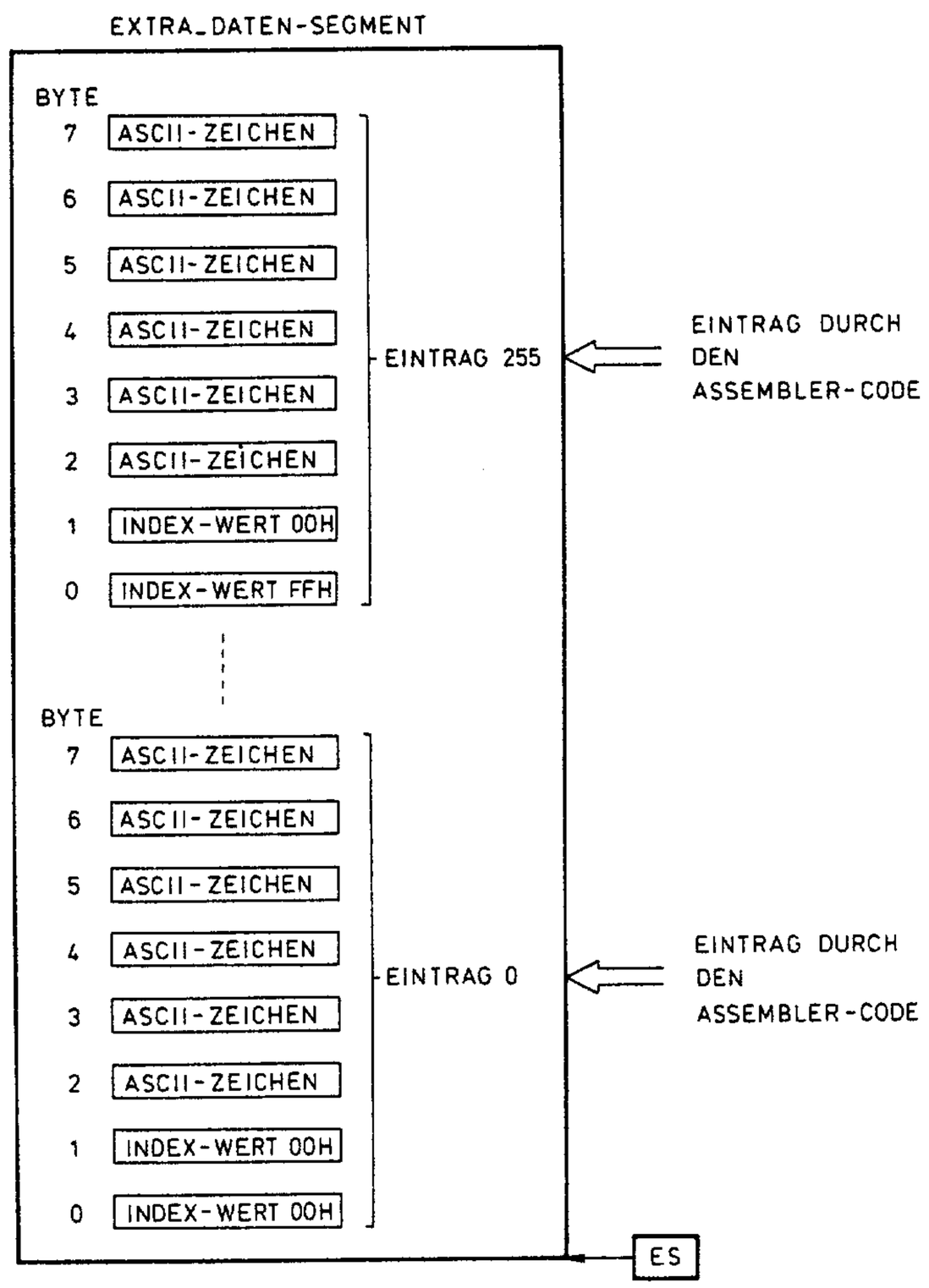
- die Code-Segmente der beiden Module kombiniert und als Ergebnis die CGROUP liefert,
- jeweils die DATA- und STACK-Segmente der beiden Module kombiniert und als Ergebnis die DGROUP liefert und schließlich
- außerhalb der DGROUP das zusätzliche EXTRA_DATEN-Segment zur Verfügung stellt.

Im Assembler-Programm ist zu sehen, daß das SMALL-Modell um das EXTRA_DATEN-Segment erweitert ist. Dieses Segment steht alleine, d.h. es ist nicht kombinierbar und gehört zu keiner GROUP.

Für den C-Code gibt es keine Möglichkeit, auf die Daten in diesem Segment direkt zuzugreifen. Doch wie gezeigt wird, haben diese Möglichkeit die Assembler-Prozeduren, ohne dabei das 'Prozedur-Interface' des SMALL-Modells zu verletzen.

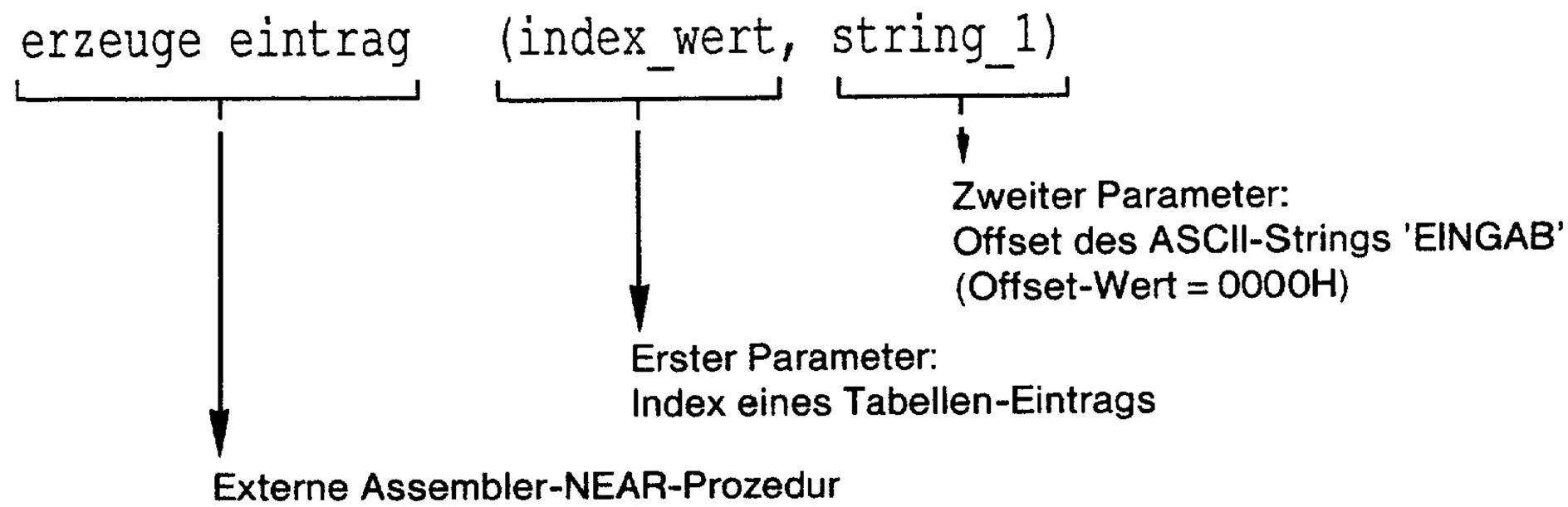
Das EXTRA_DATEN-Segment enthält eine Tabelle mit 256 verschiedenen Eintragungen und ist ES-relativ erreichbar. Jeder Eintrag besteht aus 8 Bytes, wobei der Assembler-Code in die ersten beiden Bytes einen passenden Index-Wert im Bereich von 0 bis 255 und in die folgenden 6 Bytes einen String mit 6 ASCII-Zeichen einträgt.

Diese Situation ist im folgenden Bild illustriert:



Sowohl den Index-Wert, als auch den ASCII-String liefert das C-Programm. Die Eintragung besorgt dann die Assembler-NEAR-Prozedur erzeuge_eintrag.

Das C-Programm übergibt die beiden Parameter an die externe, als PUBLIC deklarierte Assembler-NEAR-Prozedur mit folgendem Aufruf:

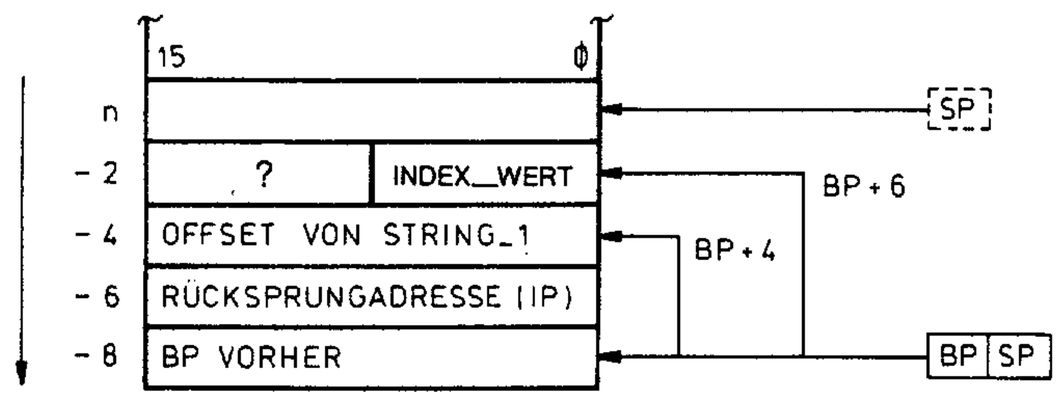


Diese Parameter werden nach Standard-Konvention (alien) von links nach rechts an den Stack übergeben, so daß sie die Assembler-Prozedur erzeuge_eintrag nach der BP-Methode bearbeiten kann.

Das Stack-Layout nach der Parameter-Übergabe und nach dem Assembler-Prolog

```
PUSH  BP                    ;Prolog
MOV   BP,SP
```

zeigt das folgende Bild:



Der erste Parameter index_wert, der als erstes Wort im neuen Eintrag der Daten-Tabelle gespeichert werden soll, liegt bei der Position [BP+6]. Der zweite Parameter string_1 liegt bei der Position [BP+4] und ist ein Offset-Wert relativ zur DGROUP, der im DATA-Segment des C-Programms den Beginn des 6-Byte-ASCII-Strings 'EINGAB' kennzeichnet. Die Position dieses Strings liegt bei Offset 0000H (siehe resultierendes Speicher-Layout).

Die Prozedur erzeuge_eintrag benützt ES, um die Tabelle im EXTRA_DATEN-Segment zu adressieren. Daher wird ES mit der Basis von EXTRA_DATEN geladen:

```
MOV     AX,EXTRA_DATEN              ;ES=Basis des
MOV     ES,AX                       ;   EXTRA_DATEN-Segments
```

Die folgende Code-Sequenz zerfällt in zwei Aufgaben. Als erstes wird AL mit dem Wert von NAECHSTER_INDEX geladen. Diese Variable ist im DATA-Segment des Assembler-Programms definiert und kann Werte im Bereich von 0..255 (FFH) annehmen.

Sie wird vom Assembler-Programm als Index benützt, um innerhalb der Daten-Tabelle fortlaufend die individuellen 8-Byte-Eintragungen zu selektieren.

```
DATA SEGMENT PUBLIC 'DATA'

NAECHSTER_INDEX DB 0 ①                    ;Index des naechsten Eintrags
EXTRN eintrag_puffer:BYTE

DATA ENDS
```

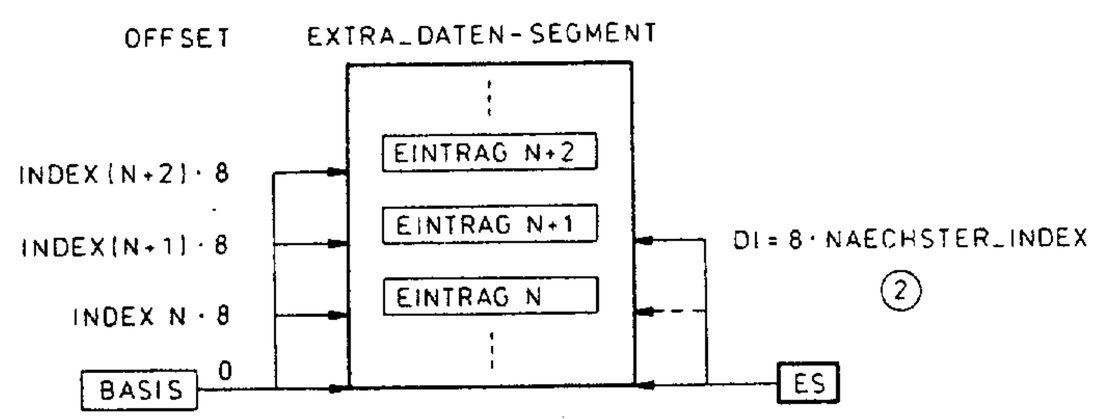
Initialisiert ist die Variable zunächst mit 0, um anzuzeigen, daß die Tabelle leer ist ①.

Nachdem mit

```
MOV     AL,NAECHSTER_INDEX              ;AX=augenblicklicher
MOV     AH,0                            ;   Indexwert
```

der Index eines bestimmten 8-Byte-Eintrags geladen worden ist, wird als nächstes in DI der Index-Wert mit 8 multipliziert. Damit zeigt ES:DI zum nächsten Eintrag in der Daten-Tabelle ②.

```
MOV     DI,AX
SHL     DI,3                            ;DI=8*NAECHSTER_INDEX
                                        ;   Damit zeigt ES:DI zum
                                        ;   naechsten Eintrag
```



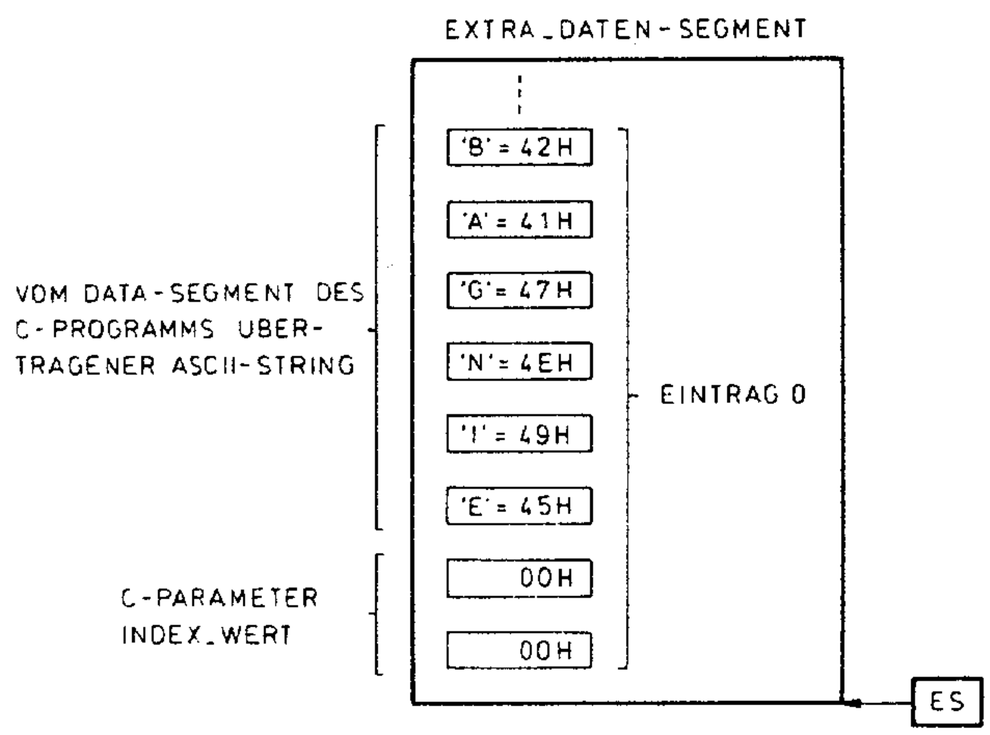
Die beiden Befehle

```
MOV     BX,[BP+6]                       ;BX=index_wert (C-Variable)
MOV     ES:[DI],BX                      ;index_wert im naechsten
                                        ;Eintrag der Daten-Tabelle
                                        ;speichern
```

speichern nun den ersten vom C-Programm übergebenen Parameter (index_wert) im ersten Wort des neuen Eintrags. Die nächsten fünf Befehle übertragen den 6-Byte-String 'EINGAB', der im C-Programm deklariert ist, in der Daten-Tabelle im EXTRA_DATEN-Segment:

```
ADD     DI,2                          ;ES:DI zeigt zum String-Teil
                                      ;des Eintrags
MOV     SI,[BP+4]                     ;SI=Offset von string_1 (C-
                                      ;Variable). DS:SI zeigt
                                      ;zum uebertragenden String
MOV     CX,3                          ;Der String ist 3 Worte
                                      ;(6 Bytes) lang
CLD                                   ;Direction-Flag=0 bedeutet
                                      ;SI und DI inkrementieren
REP MOVS ES:WORD PTR[DI],DS:[SI]      ;Uebertrage String in die
                                      ;Daten-Tabelle
```

Ist der bisher beschriebene Programm-Abschnitt mit dem C-Parameter index_wert=0 zum ersten mal ausgeführt worden, enthält der Eintrag 0 in der Daten-Tabelle folgende Werte:



Nach Ausführung von

```
INC     NAECHSTER_INDEX               ;Naechsten Eintrag in der
                                      ;Daten-Tabelle indizieren
```

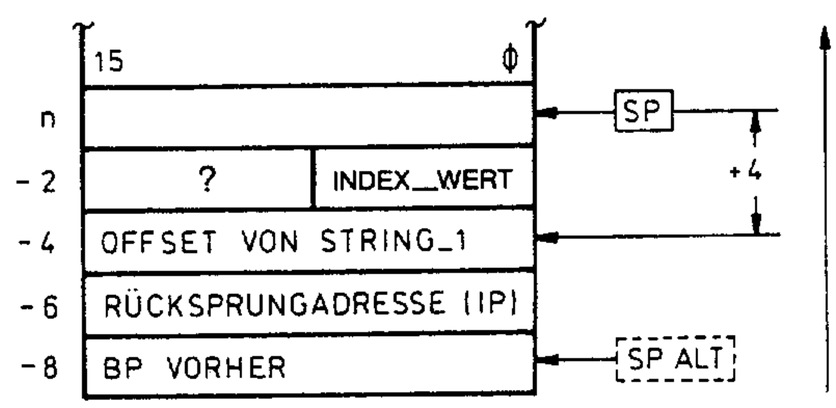
enthält nach dem erstmaligen Programm-Durchlauf die Assembler-Variable NAECHSTER_INDEX den Wert 1 und indiziert damit den nächsten Eintrag 1 in der Daten-Tabelle. Nach dem folgenden Programm-Durchlauf enthält NAECHSTER_INDEX den Wert 2. Auf diese Weise werden bei 256 Programm-Durchläufen alle Eintragungen in der Daten-Tabelle indiziert.

Im Epilog der Prozedur erzeuge_eintrag, entfernen die Befehle

```
        POP     BP                          ;Epilog
END_1:  RET     4                           ;Entferne die Parameter
                                            ;index_wert und string_1
                                            ;aus dem Stack und zurueck
                                            ;zum C-Programm
```

die Parameter index_wert und string_1 wieder aus dem Stack. Zusätzlich bewirkt `RET 4` die Rückkehr zum C-Programm.

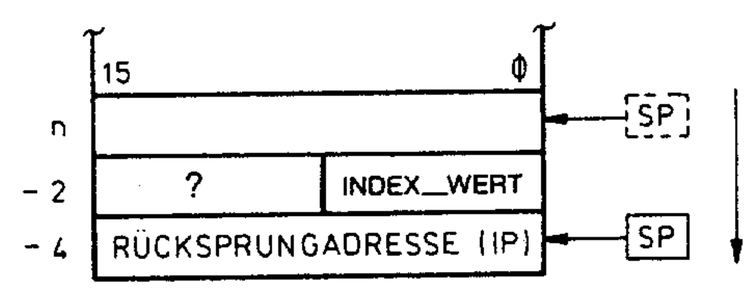
Die Stack-Situation am Ende des Prozedur-Epilogs zeigt das folgende Bild:



Nachdem die 8-Byte-Eintragungen im EXTRA_DATEN-Segment erfolgt sind, können sie vom C-Programm überprüft werden. Zu diesem Zweck stellt die Assembler-NEAR-Prozedur nachsehen_eintrag dem C-Code den letzten Eintrag zur Verfügung. Die Prozedur ist als PUBLIC deklariert und wird mit der folgenden C-Anweisung aufgerufen:

```
nachsehen_eintrag(index_wert)
```

Nach dem Aufruf der Assembler-Prozedur ergibt sich folgendes Stack-Layout:



Wie im Assembler-Listing zu sehen ist, benützt die Prozedur nachsehen_eintrag die POP-Methode, um auf den Parameter index_wert zuzugreifen:

```
POP     AX                          ;AX=Ruecksprungadresse (IP)
POP     BX                          ;BX (BL)=Parameter index_wert
                                    ;(0..255)
PUSH    AX                          ;Ruecksprungadresse (IP) in den
                                    ;Stack zurueckschreiben
```

Diese drei Befehle retten zunächst die Rücksprungadresse (IP) nach AX und den Parameter index_wert nach BX. Der eigentliche Index-Wert steht dann in BL. Anschließend wird die Rücksprungadresse in den Stack zurückgeschrieben.

Es wird wiederum ES benützt, um den Eintrag im EXTRA_DATEN-Segment zu erreichen. Daher wird ES mit der Basis des EXTRA_DATEN-Segments geladen:

```
MOV     AX,EXTRA_DATEN              ;ES=Basis des
MOV     ES,AX                       ;EXTRA_DATEN-Segments
```

Der Offset für den letzten Eintrag ergibt sich, wenn der Index-Wert mit 8 multipliziert wird:

```
MOV     BH,0                        ;ES:BX zeigt auf den zurueckzu-
SHL     BX,3                        ;lesenden Eintrag
```

Der Eintrag wird nun an den 8-Byte-Character-Vektor eintrag_puffer des C-Programms übergeben. Sie ist daher im DATA-Segment des Assembler-Moduls als externe (EXTRN) Byte-Variable vereinbart.

Vollständiges Assembler-DATA-Segment:

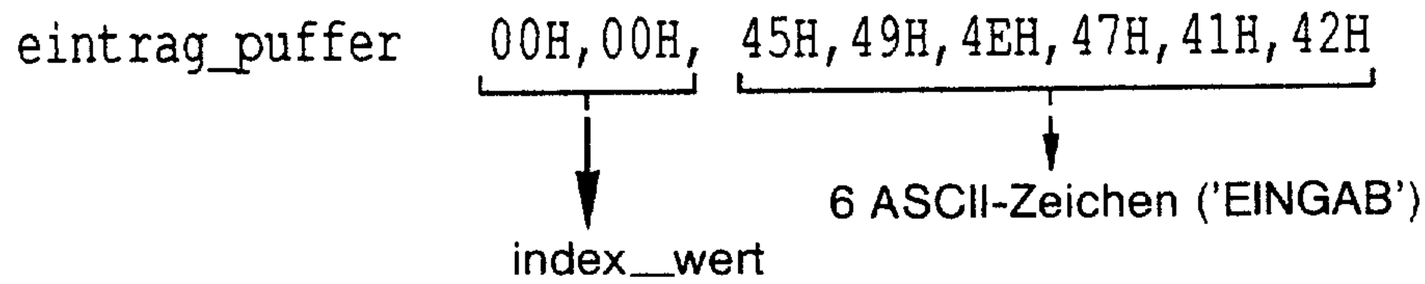
```
DATA SEGMENT PUBLIC 'DATA'

NAECHSTER_INDEX DB 0                ;Index des naechsten Eintrags
EXTRN eintrag_puffer:BYTE

DATA ENDS
```

Auf der Assembler-Ebene wird das DI-Register als Offset benützt, um sowohl den Eintrag im EXTRA_DATEN-Segment (Quelle), als auch den Character-Vektor eintrag_puffer des C-Programms (Ziel) zu adressieren.

```
        MOV     DI,0                            ;DI=Offset des ersten Worts
                                                ;   im Eintrag
        MOV     CX,4                            ;Es werden 4 Worte (8 Bytes)
                                                ;uebertragen
BEKOMMEN_EINTRAG:
        MOV     AX,ES:[BX][DI]                    ;Wort von Daten-Tabelle im
                                                  ;EXTRA_DATEN-Segment lesen
        MOV     WORD PTR eintrag_puffer[DI],AX    ;Wort in den C-Character-
                                                  ;Vektor eintrag_puffer
        ADD     DI,2                              ;Naechst. Wort adressieren
        LOOP    BEKOMMEN_EINTRAG                  ;Wiederhole, bis 8 Bytes
                                                  ;(4 Worte) uebertragen
END_2:  RET                                       ;Zurueck zum C-Programm
```

Der angegebene Assembler-Code zeigt, wie die komplexen Adreß-Ausdrücke ES: [BX] [DI] und WORD PTR eintrag_puffer [DI], AX eine schnelle Wortübertragung ermöglichen. Für die Übertragung des gesamten Eintrags ist lediglich mit `ADD DI, 2` der Offset-Wert in DI zu aktualisieren.

Ist der Eintrag 0 übertragen worden, enthält der Character-Vektor eintrag_puffer des C-Programms folgende Werte:



Wenn nach der Kombination der beiden Module B83 (C-Modul) und BEISPIEL_84 (Assembler-Modul) das resultierende Programm auf einer Ziel-Maschine im PROM laufen soll, muß ein zusätzlicher Assembler-Code die Register DS,SS und SP initialisieren. In der Regel übernimmt die Initialisierung dieser Register der Code des Haupt-Moduls. Das folgende Assembler-Listing zeigt hierfür ein Beispiel:

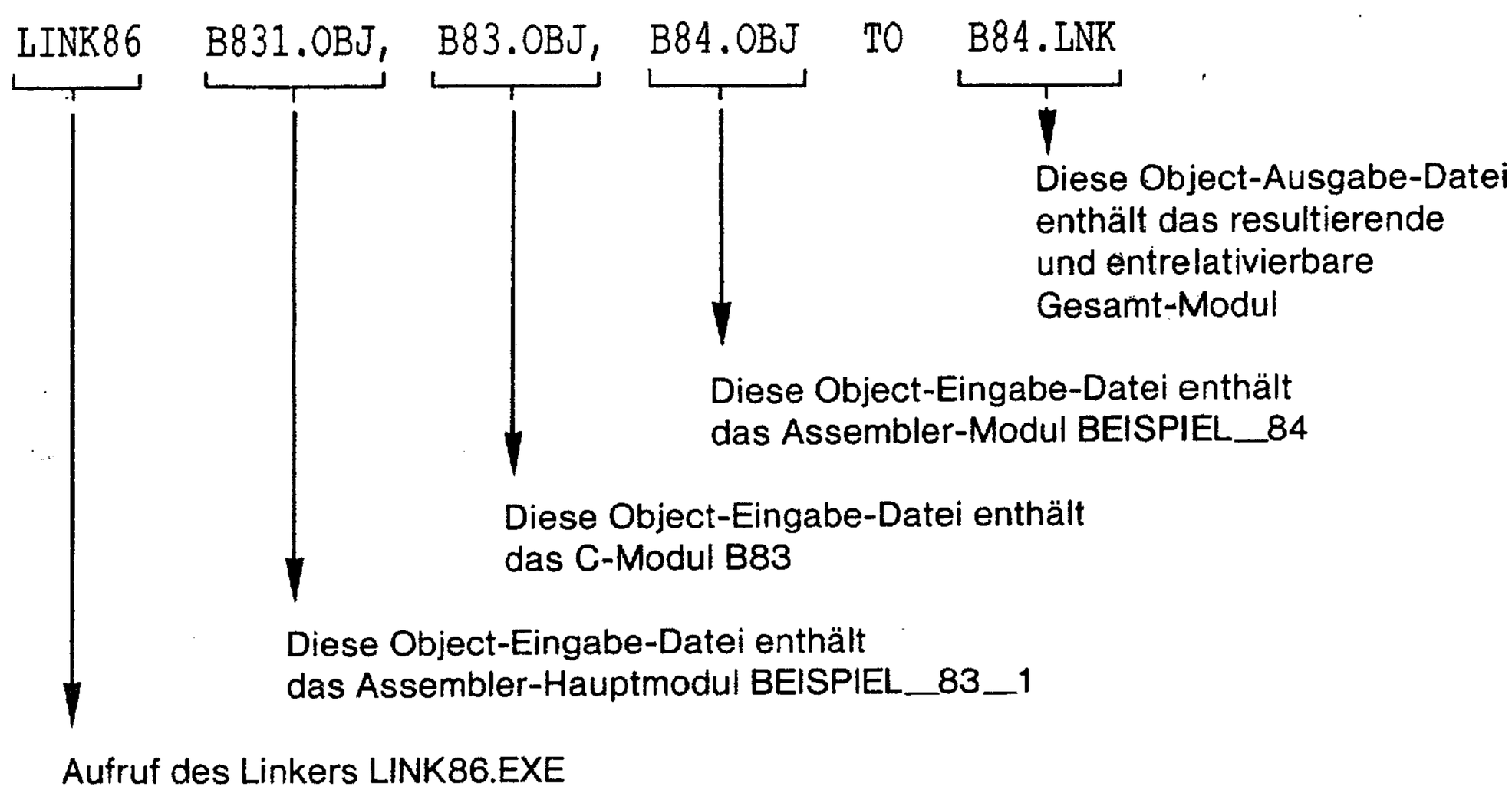
```
LOC  OBJ                  LINE     SOURCE

                            1              NAME BEISPIEL_B83_1
                            2
                            3     CGROUP  GROUP   CODE
                            4     DGROUP  GROUP   DATA,STACK
                            5
                            6     ASSUME CS:CGROUP,DS:DGROUP,SS:DGROUP
                            7
----                        8     DATA SEGMENT PUBLIC 'DATA'                ;Dummy-Segment
                            9
----                       10     DATA ENDS
                           11
----                       12     STACK SEGMENT STACK 'STACK'
0000 (20                   13             DW  20 DUP (?)
```

```
LOC  OBJ                  LINE     SOURCE

     ????
     )
0028                        14     TOP_STACK LABEL WORD
----                        15     STACK ENDS
                            16
----                        17     CODE SEGMENT PUBLIC 'CODE'
                            18     PUBLIC START
                            19     EXTRN beispiel_83_:NEAR
0000 B8----          R      20     START:  MOV AX,DGROUP                    ;DS- und SS-Register initialisieren
0003 8ED8                   21             MOV DS,AX
0005 8ED0                   22             MOV SS,AX
0007 BC2800          R      23             MOV SP,OFFSET DGROUP:TOP_STACK  ;SP-Register initialisieren
000A E80000          E      24             CALL    beispiel_83             ;Aufruf der externen C-Funktion
000D F4                     25             HLT
----                        26     CODE ENDS
                            27     END START
```

Wie zu erkennen ist, enthält das Haupt-Modul BEISPIEL_83_1 in Übereinstimmung mit dem SMALL-Modell die notwendigen GROUP- und Segment-Definitionen. Dabei werden im CODE-Segment die Register DS,SS und SP in der üblichen Weise initialisiert. Ist dies erledigt, ruft `CALL beispiel_83` die externe C-Funktion beispiel_83() auf.

Hinweis: Im Beispiel ist der Befehl, der als erstes nach einem System-RESET ausgeführt werden soll, mit dem Label START: markiert. Damit der INTEL-Locater LOC86.EXE die physikalische Position dieses Befehls erkennen kann, ist START als PUBLIC gekennzeichnet.

Um ein ablauffähiges Programm zu bekommen, sind zunächst die beteiligten Module in der richtigen Reihenfolge zu binden. Dies erledigt der INTEL-Linker LINK86.EXE mit dem folgenden Kommando:



Da der Linker lediglich die zusammengehörigen Segmente der Eingabe-Module kombiniert, muß nun der INTEL-Locater LOC86.EXE die physikalischen Positionen der Segmente und GROUPS festlegen. Das folgende Locater-Kommando zeigt hierfür ein Beispiel:

```
>LOC86 B84.LNK TO B84.LOC AD(SM(CODE(0FC000H), DATA(200H), &
>>STACK(300H), EXTRA_DATEN(400H)), GR(DGROUP(200H))) START(START)
```

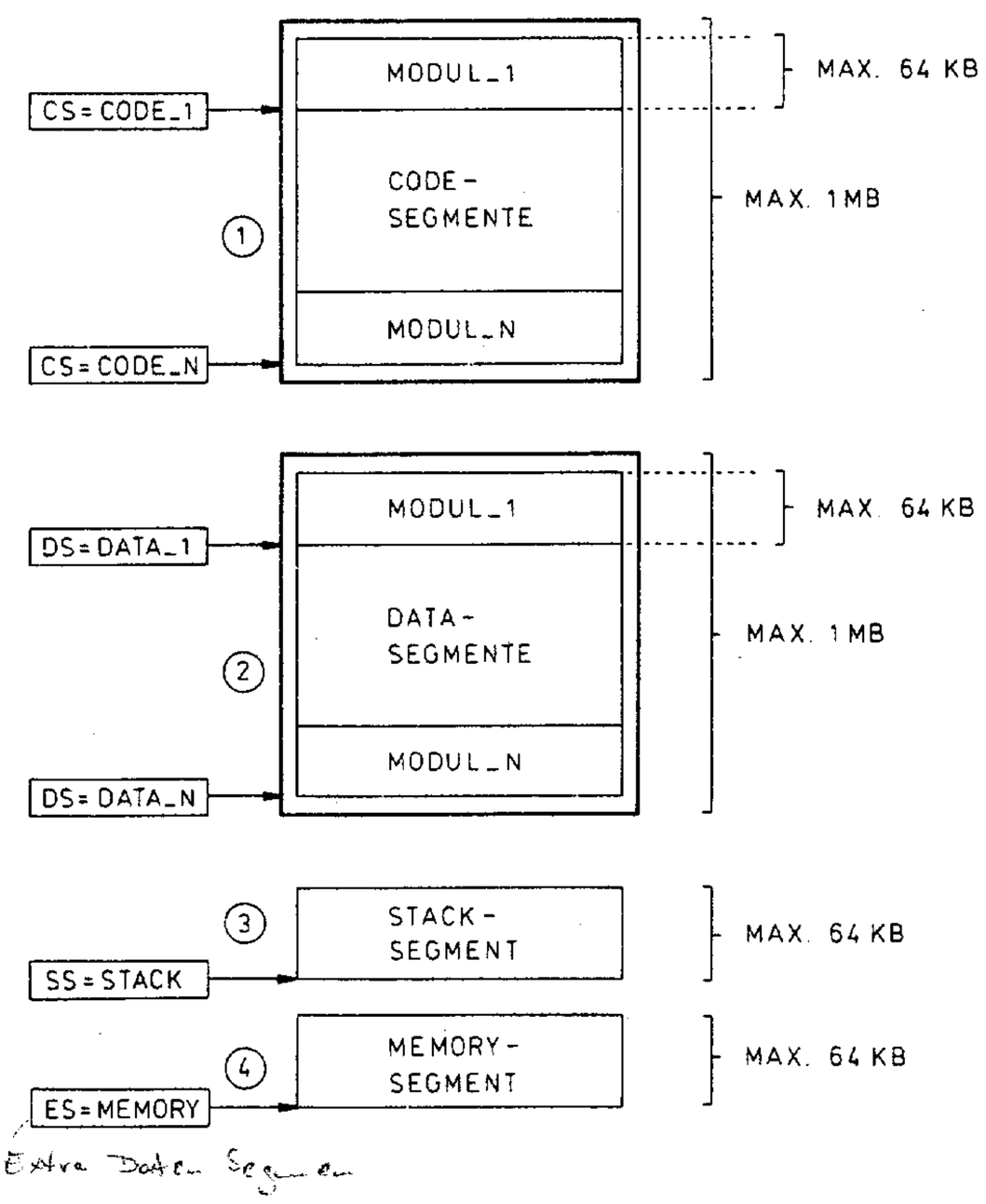
LOC86	Aufruf des Locaters LOC86.EXE
B84.LNK	OBJECT-Eingabe-Datei enthält entrelativierbares Modul
B84.LOC	OBJECT-Ausgabe-Datei enthält entrelativiertes und damit ablauffähiges Programm
AD	Schlüsselwort für ADDRESSES
SM	Schlüsselwort für SEGMENTS
CODE	Name des Code-Segments (SMALL-Modell)
(0FC000H)	Physikalische Basis-Adresse des CODE-Segments
DATA	Name des Daten-Segments (SMALL-Modell)
(200H)	Physikalische Basis-Adresse des DATA-Segments
&	Kommando-Verlängerung
>>	Rückmeldung nach Kommando-Verlängerung
STACK	Name des Stack-Segments (SMALL-Modell)
(300H)	Physikalische Basis-Adresse des STACK-Segments
EXTRA_DATEN	Name des zusätzlichen Daten-Segments (Segment-Erweiterung des SMALL-Modells)
(400H)	Physikalische Basis-Adresse des EXTRA_DATEN-Segments
GR	Schlüsselwort für GROUPS
DGROUP	Schlüsselwort für kombinierte Daten-/Stack-Regionen
(200H)	Physikalische Basis-Adresse der DGROUP
START	Teilt dem Locater die Start-Position des ersten Befehls nach RESET mit
(START)	PUBLIC-Symbol (Start-Label) des Assembler-Hauptmoduls

Der Locater LOC86.EXE liefert ein entrelativiertes und damit ablauffähiges Programm und bewahrt es in der Datei B84.LOC auf. Wenn nun mit dem Code des Programms ein PROM programmiert wird, kann das Programm anschließend auf einer Ziel-Hardware laufen und dort mit geeigneten Emulatoren getestet werden.

8.2 Das LARGE-Modell

Dieses Modell wird für eine große Menge von Code und Daten benützt. Wie im folgenden Bild zu sehen ist, können eine Vielzahl von Modulen existieren (Modul_1..Modul_N), wobei sich die lokalen Code- und Daten-Segmente der **individuellen** Module in jeweils maximal 64 KBytes aufhalten. Es gibt jedoch nur ein **einziges** Stack-Segment mit einer maximalen Größe von 64 KBytes.

Im LARGE-Modell sind weder die Code- noch die Daten-Segmente kombinierbar. Im Gegensatz dazu lassen sich die lokalen Stack-Segmente der individuellen Module mit dem Combine-Typ **STACK** kombinieren.



Im LARGE-Modell enthält

- das CS-Register die Basis-Adresse der lokalen Code-Region ①,
- das DS-Register die Basis-Adresse der lokalen Daten-Region ②,
- das SS-Register die Basis-Adresse des gemeinsamen Stack-Segments aller Module ③ und
- das ES-Register die Basis-Adresse einer zusätzlichen lokalen Daten-Region ④.

Da jedes Modul sein eigenes Code-Segment hat, benützen Inter-Modul-Aufrufe die lange Form des CALL-Befehls. Dies bedeutet, daß sowohl CS als auch IP umgeladen werden. Aus diesem Grund müssen externe PUBLIC-Prozeduren vom Typ FAR sein. Als Rücksprung-Adressen werden dann automatisch komplette CS:IP-Zeiger im Stack hinterlegt.

Prozeduren, die sich im gleichen lokalen Code-Segment aufhalten wie die aufrufenden CALL-Befehle, können vom Typ NEAR sein.

Da jedes Modul seine eigene lokale Daten-Region hat, muß eine Prozedur, die von einem anderen Modul aufgerufen worden ist, den **DS**-Wert des Aufrufers in den Stack retten (PUSH DS). Dies gilt **auch** für **ES**, wenn die Module noch eigene Zusatz-Daten-Segmente enthalten sollten (PUSH ES).

Die aufgerufene Prozedur muß nun DS oder ES **umladen**, um die eigenen lokalen Variablen zu adressieren. Bevor aber die Rückkehr zum Aufrufer erfolgt, muß die augenblicklich aktive Prozedur die DS- oder ES-Werte des vorher verlassenen Moduls in die entsprechenden Register zurückschreiben (POP ES, POP DS).

Sollte die aufgerufene PUBLIC-Prozedur keine lokalen Variablen in ihrem Daten- oder Zusatz-Daten-Segment benützen, kann sie sich das Retten und Zurückschreiben von DS oder ES sparen.

Wie bereits erwähnt, benützen alle Module das gleiche Stack-Segment. Daher ändert sich der Inhalt des **SS**-Registers bei Inter-Modul-Aufrufen **nicht**.

In Anlehnung an das Speicher-Layout eines LARGE-Programms, ist für ein Assembler-Programm folgendes 'Rahmenwerk' definiert:

LARGE-Modell-Segmentierung:

```
NAME     <Modul-Name>
ASSUME   CS:CODE,DS:DATA,SS:STACK
EXTRN    <Externe Variablen>
DATA     SEGMENT   'DATA'
         <Hier sind die lokalen Programm-Variablen
         zu definieren>
DATA     ENDS
STACK    SEGMENT  STACK  'STACK'
         <Hier ist das DW-Statement zu benützen, um
         die Länge des 'Run Time'-Stack festzulegen>
STACK    ENDS
MEMORY   SEGMENT  MEMORY 'MEMORY'
         <Hier können zusätzliche Programm-Daten
         oberhalb der anderen Segmente definiert werden>
MEMORY   ENDS
EXTRN    <Externe FAR-Labels, wie z.B Prozedur-Namen>
CODE     SEGMENT   'CODE'
         <Hier wird der lokale Programm-Code aufbewahrt>
CODE     ENDS
END      <Hier kann zur Definition eines Hauptmoduls
         die Startadresse des Programms angegeben werden>
```

Das 'Rahmenwerk' zeigt exakt, welche SEGMENT- und ASSUME-Statements für die Code-, Daten-, Stack- und Memory-Segmente eines Moduls angegeben werden müssen und wo die EXTRN-Direktiven zu plazieren sind. Als Align-Typ ist PARA vordefiniert und kann geändert werden.

Da im LARGE-Modell viele Module mit individuellen nicht-kombinierbaren Code- und Daten-Regionen existieren, müssen in allen zusätzlichen Modulen die im 'Rahmenwerk' angegebenen lokalen Segment-Namen wie DATA, MEMORY und CODE durch anwenderdefinierte Namen ersetzt werden.

Hinweis: Wenn das MEMORY-Segment **nicht oberhalb** der anderen Segmente, sondern irgendwo im 1-MByte-Adreßraum plaziert werden soll, kann das **Attribut MEMORY** weggelassen werden.

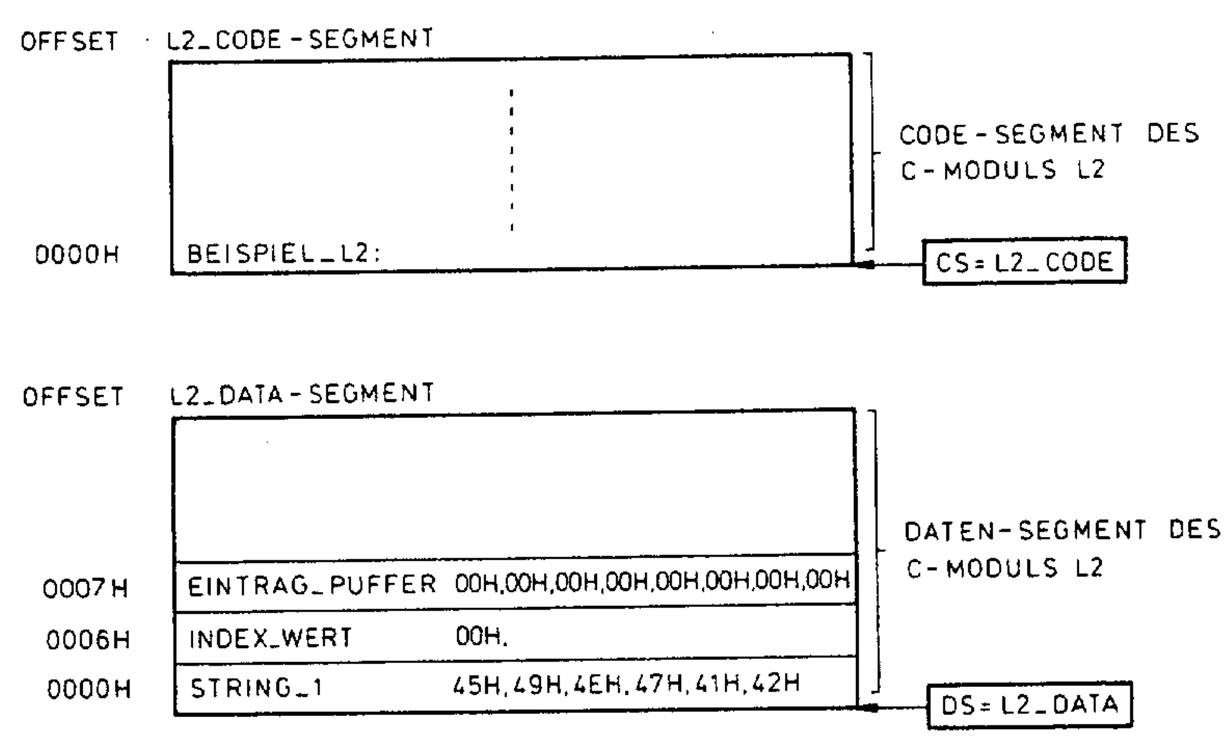
8.2.1 Programmierbeispiel: Modul-Kombination im LARGE-Modell

Die im Abschnitt 8.1.5 beschriebene Modul-Kombination im SMALL-Modell soll nun auf das LARGE-Modell übertragen werden. Dabei hält sich das gleiche C-Programm jetzt im Modell L2 auf und ist nach dem LARGE-Modell übersetzt. Auch hier wurden ASM86 und iC86 verwendet.

C-Modul:

```
char string_1[]={'E','I','N','G','A','B'};
char index_wert;
char eintrag_puffer[8];
beispiel_l2()
{
alien erzeuge_eintrag();
alien nachsehen_eintrag();

erzeuge_eintrag(index_wert,string_1);
nachsehen_eintrag(index_wert);
}
```

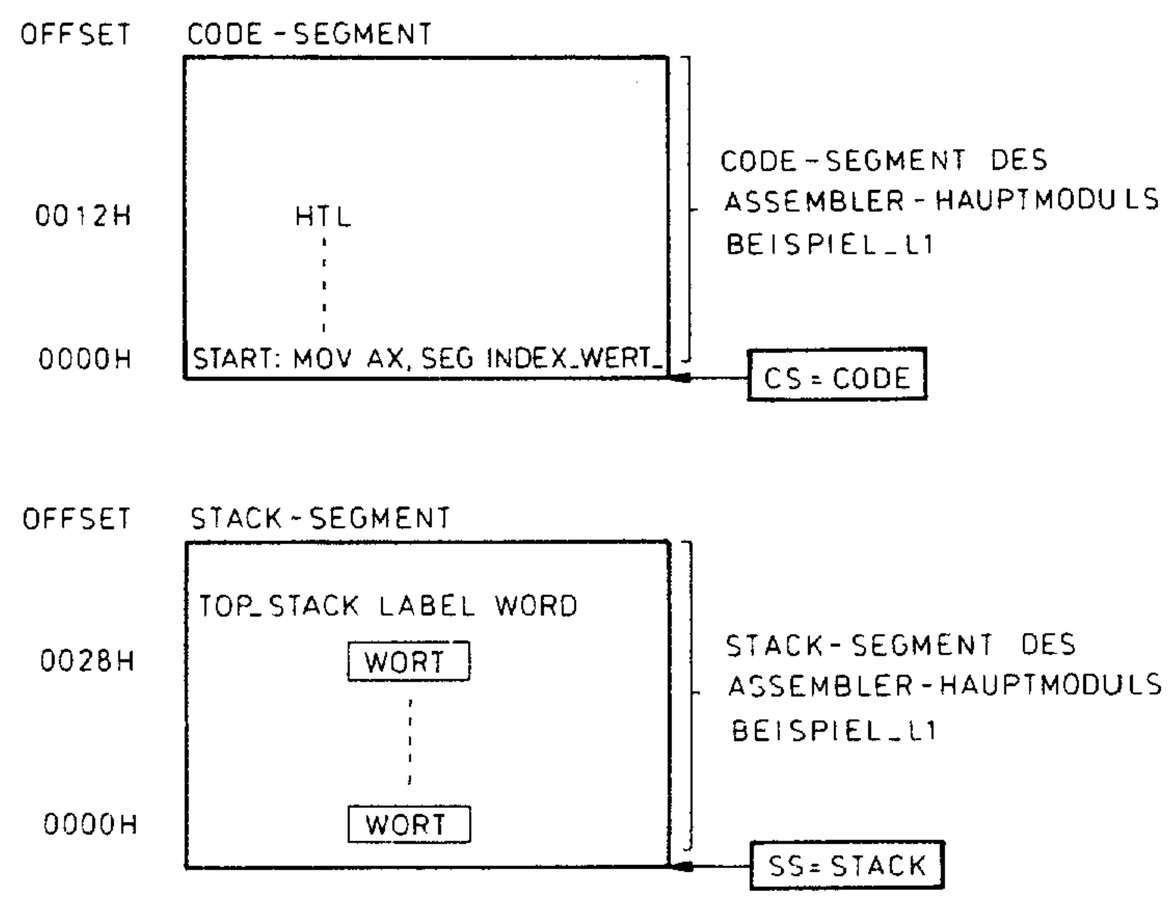
Der C-Compiler generiert eigene Segment-Namen, um die Code- und Daten-Region eines Anwender-Programms zu kennzeichnen. Zu diesem Zweck nimmt er den vom Anwender vergebenen **Modulnamen** (im Beispiel L2) und erweitert ihn um die Bezeichnungen **_CODE** und **_DATA**. Für das C-Programm bedeutet dies, daß der C-Compiler den Code im **L2_CODE**-Segment und die Daten im **L2_DATA**-Segment aufbewahrt. Unter diesen Voraussetzungen ergibt sich folgendes Speicher-Layout:



Da auch im LARGE-Modell das resultierende Gesamt-Modul auf einer Ziel-Maschine im PROM laufen soll, ist es erforderlich, die Register DS,SS und SP für das C-Modul zu initialisieren. Dies übernimmt das folgende **Assembler-Hauptmodul**:

```
LOC  OBJ                    LINE     SOURCE

                               1             NAME BEISPIEL_L1
                               2
                               3     ASSUME CS:CODE,SS:STACK
                               4
----                           5     STACK SEGMENT STACK 'STACK'
0000 (20                       6             DW  20 DUP (?)
     ????
     )
0028                           7     TOP_STACK LABEL WORD
----                           8     STACK ENDS
                               9
                              10     EXTRN beispiel_l2:FAR,index_wert:BYTE
                              11
----                          12     CODE SEGMENT 'CODE'
                              13     PUBLIC START
0000 B8----          E        14     START:  MOV AX,SEG index_wert          ;AX=Selektor des C-Programm-
                              15                                             ;   Daten-Segments
0003 8ED8                     16             MOV DS,AX                       ;DS initialisieren
0005 B8----          R        17             MOV AX,STACK
0008 8ED0                     18             MOV SS,AX                       ;SS initialisieren
000A BC2800          R        19             MOV SP,OFFSET TOP_STACK         ;SP-Register initialisieren
000D 9A0000----      E        20             CALL    beispiel_l2            ;Aufruf der externen C-Funktion
0012 F4                       21             HLT
----                          22     CODE ENDS
                              23          END START
```

Dieses Modul liefert folgendes Speicher-Layout:



Für das Haupt-Modul besteht das Problem, daß es das Daten-Segment L2_DATA des C-Programms nicht kennt. Um aber den Selektor des C-Programm-Daten-Segments zu bekommen, benützt das Modul den Assembler-SEG-Operator zusammen mit der C-Variablen index_wert (auch string_1 oder eintrag_puffer sind möglich).

Dieser Operator liefert den Selektor des Segments, in dem sich die Variable index_wert aufhält. Für das angegebene Kombinations-Beispiel ist dies der Selektor des Daten-Segments L2_DATA.

Hinweis: Die Initialisierung von DS im Hauptmodul ist **nicht** zwingend notwendig, da dies der C-Compiler iC86 Version 4.0 mit `MOV CX,L2_DATA` und `MOV DS,CX` selbst erledigt. Erforderlich ist die Initialisierung von DS nur dann, wenn auch das Hauptmodul auf die C-Variablen zugreift.

Hat das Haupt-Modul die Register DS,SS oder SP initialisiert, ruft es mit `CALL beispiel_l2` die externe C-FAR-Prozedur beispiel_l2 auf.

Das C-LARGE-Modul L2 korrespondiert nun mit dem folgenden Assembler-Modul BEISPIEL_L3. Dieses Modul ist in Übereinstimmung mit den Konventionen des LARGE-Modells aufbereitet und leistet das gleiche wie das Modul BEISPIEL_84 im SMALL-Modell.

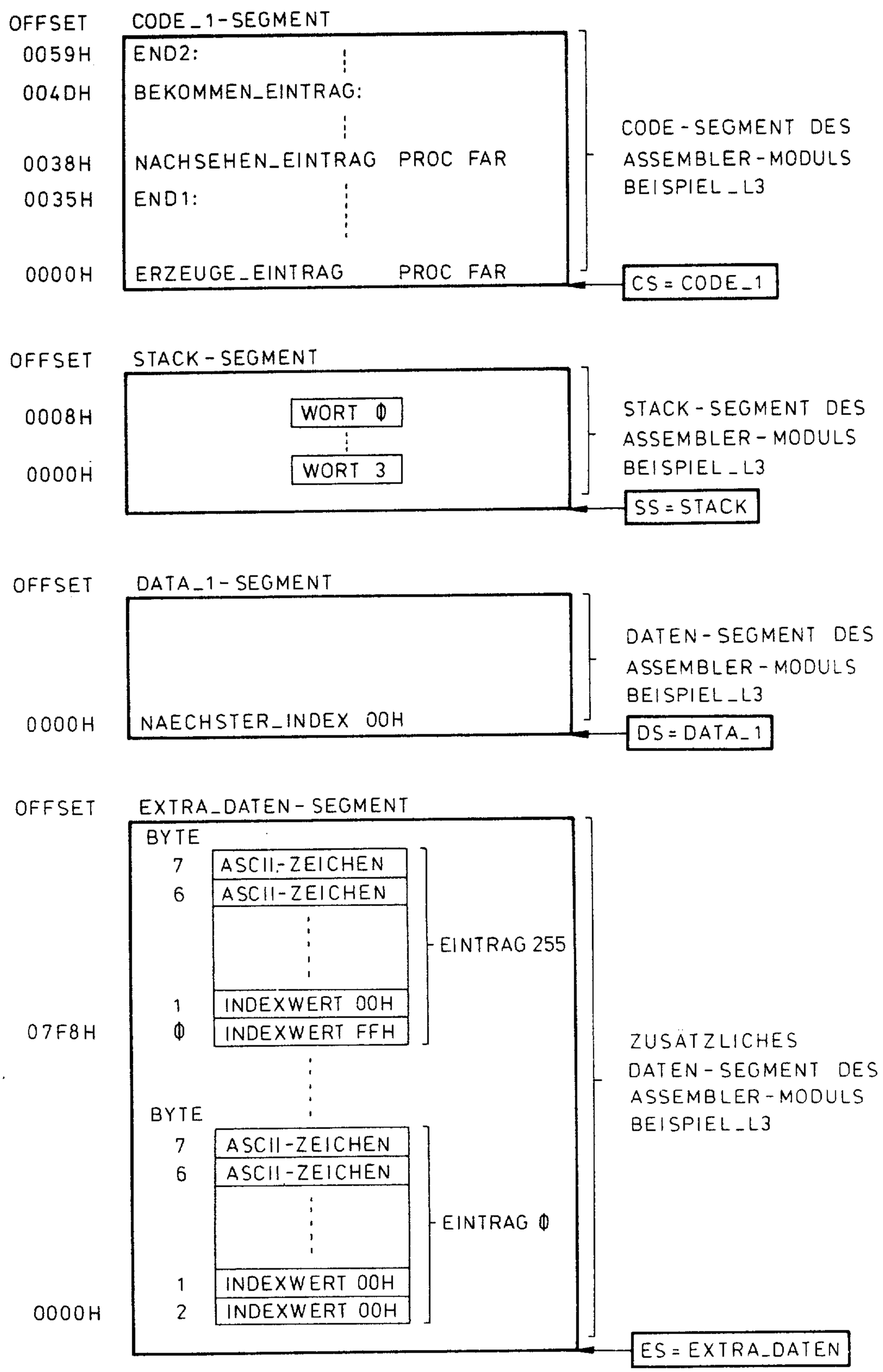
Assembler-Modul:

```
LOC  OBJ                    LINE     SOURCE

                              1             NAME    BEISPIEL_L3
                              2
                              3     ASSUME CS:CODE_1,DS:DATA_1,SS:STACK
                              4
----                          5     EXTRA_DATEN SEGMENT 'MEMORY'
                              6
0000 (256                     7         DB  256 DUP (8 DUP (0))                    ;Daten-Tabelle mit
      (8
     00
      )
     )
                              8                                                    ;256 Acht-Byte-Eintragungen
----                          9     EXTRA_DATEN ENDS
                             10
                             11     EXTRN eintrag_puffer:BYTE
                             12
----                         13     DATA_1 SEGMENT 'DATA'
                             14
0000 00                      15     NAECHSTER_INDEX DB 0                          ;Index des naechsten Eintrags
                             16
----                         17     DATA_1 ENDS
                             18
----                         19     STACK SEGMENT STACK 'STACK'
0000 (4                      20         DW  4 DUP (?)
     ????
     )
----                         21     STACK ENDS
                             22
----                         23     CODE_1 SEGMENT 'CODE'
                             24
                             25     PUBLIC erzeuge_eintrag,nachsehen_eintrag
                             26
0000                         27     erzeuge_eintrag PROC FAR
0000 55                      28             PUSH    BP                            ;Prozedur-Prolog
```

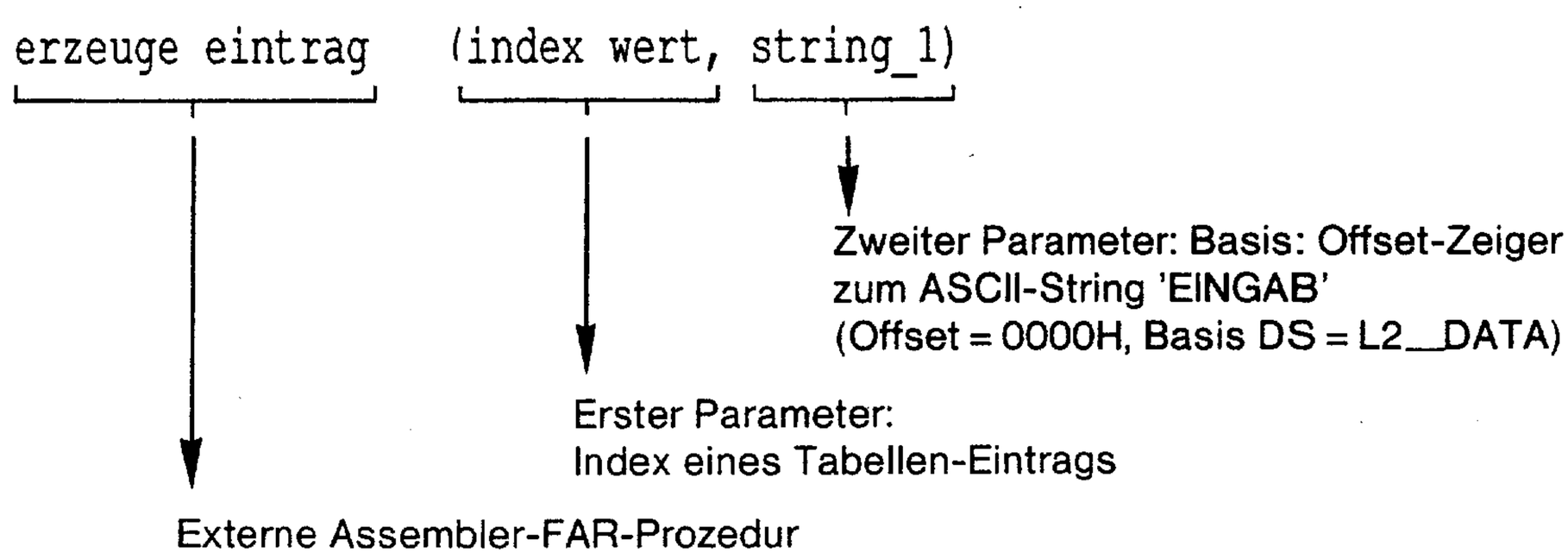
```
LOC  OBJ                  LINE     SOURCE

0001 8BEC                   29            MOV     BP,SP
0003 1E                     30            PUSH    DS                          ;Selektor des C-Programm-
                            31                                                ;Daten-Segments retten
0004 B8----        R        32            MOV     AX,DATA_1                   ;AX=Selektor des
                            33                                                ;   DATA_1-Segments
0007 8ED8                   34            MOV     DS,AX                       ;DS initialisieren
0009 B8----        R        35            MOV     AX,EXTRA_DATEN              ;ES=Basis des
000C 8EC0                   36            MOV     ES,AX                       ;   EXTRA_DATEN-Segments
000E A00000                 37            MOV     AL,NAECHSTER_INDEX          ;AX=augenblicklicher
0011 B400                   38            MOV     AH,0                        ;   Indexwert
0013 8BF8                   39            MOV     DI,AX
0015 C1E703                 40            SHL     DI,3                        ;DI=8*NAECHSTER_INDEX
                            41                                                ;   Damit zeigt ES:DI zum
                            42                                                ;   naechsten Eintrag
0018 8B5E0A                 43            MOV     BX,[BP+10]                  ;BX=index_wert (C-Variable)
001B 26891D                 44            MOV     ES:[DI],BX                  ;index_wert im naechsten
                            45                                                ;Eintrag der Daten-Tabelle
                            46                                                ;speichern
001E 83C702                 47            ADD     DI,2                        ;ES:DI zeigt zum String-Teil
                            48                                                ;des Eintrags
0021 C57606                 49            LDS     SI,DWORD PTR [BP+6]         ;SI=Offset von string_1
                            50                                                ;DS=Basis von string_1
                            51                                                ;(C-Variable). DS:SI zeigt
                            52                                                ;zum uebertragenden String
0024 B90300                 53            MOV     CX,3                        ;Der String ist 3 Worte
                            54                                                ;(6 Bytes) lang
0027 FC                     55            CLD                                 ;Direction-Flag=0 bedeutet
                            56                                                ;SI und DI inkrementieren
0028 F3                     57            REP MOVS ES:WORD PTR[DI],DS:[SI]    ;Uebertrage String in die
0029 A5
                            58                                                ;Daten-Tabelle
002A B8----        R        59            MOV     AX,DATA_1                   ;AX=Selektor des
                            60                                                ;   DATA_1-Segments
002D 8ED8                   61            MOV     DS,AX                       ;DS initialisieren
002F FE060000               62            INC     NAECHSTER_INDEX             ;Naechsten Eintrag in der
                            63                                                ;Daten-Tabelle indizieren
0033 1F                     64            POP     DS                          ;Selektor des C-Programm-
                            65                                                ;Daten-Segments zurueckschr.
0034 5D                     66            POP     BP                          ;Prozedur-Epilog
0035 CA0600                 67    END_1:  RET     6                           ;Entferne die Parameter
                            68                                                ;index_wert (2 Bytes) und
                            69                                                ;string_1 (4 Bytes) aus
                            70                                                ;dem Stack und zurueck zum
                            71                                                ;C-Programm
                            72    erzeuge_eintrag ENDP
                            73
0038                        74    nachsehen_eintrag PROC FAR
                            75
0038 5E                     76            POP     SI                          ;SI=Ruecksprungadresse (IP)
0039 5F                     77            POP     DI                          ;DI=Ruecksprungadresse (CS)
003A 5B                     78            POP     BX                          ;BX (BL)=Parameter index_wert
                            79                                                ;(0..255)
003B 57                     80            PUSH    DI                          ;Ruecksprungadresse (CS) und
003C 56                     81            PUSH    SI                          ;Ruecksprungadr. (IP) in den
                            82                                                ;Stack zurueckschreiben
003D B8----        R        83            MOV     AX,EXTRA_DATEN              ;ES=Basis des
0040 8EC0                   84            MOV     ES,AX                       ;EXTRA_DATEN-Segments
0042 B700                   85            MOV     BH,0                        ;ES:BX zeigt auf den zurueck-
0044 C1E303                 86            SHL     BX,3                        ;zulesenden Eintrag
0047 BF0000                 87            MOV     DI,0                        ;DI=Offset des ersten Worts
                            88                                                ;   im Eintrag
004A B90400                 89            MOV     CX,4                        ;Es werden 4 Worte (8 Bytes)
                            90                                                ;uebertragen
004D                        91    BEKOMMEN_EINTRAG:
                            92    ASSUME DS:SEG eintrag_puffer
004D 268B01                 93            MOV     AX,ES:[BX][DI]              ;Wort von der Daten-Tabelle
                            94                                                ;im EXTRA_DATEN-Segment lesen
0050 89850000      E        95            MOV     WORD PTR eintrag_puffer[DI],AX ;Wort in  C-Character-Vektor
                            96                                                ;eintrag_puffer uebertragen
0054 83C702                 97            ADD     DI,2                        ;Naechstes Wort adressieren
0057 E2F4                   98            LOOP    BEKOMMEN_EINTRAG            ;Wiederhole, bis 8 Bytes
                            99                                                ;(4 Worte) uebertragen sind
0059 CB                    100    END_2:  RET                                ;Zurueck zum C-Programm
```

```
LOC  OBJ                    LINE     SOURCE

                             101
                             102     nachsehen_eintrag ENDP
----                         103     CODE_1  ENDS
                             104             END
```

Wie zu erkennen ist, liefert BEISPIEL_L3 folgendes Speicher-Layout:



Das C-Programm ruft zunächst die externe, als PUBLIC deklarierte Assembler-FAR-Prozedur erzeuge_eintrag mit folgender Anweisung auf:

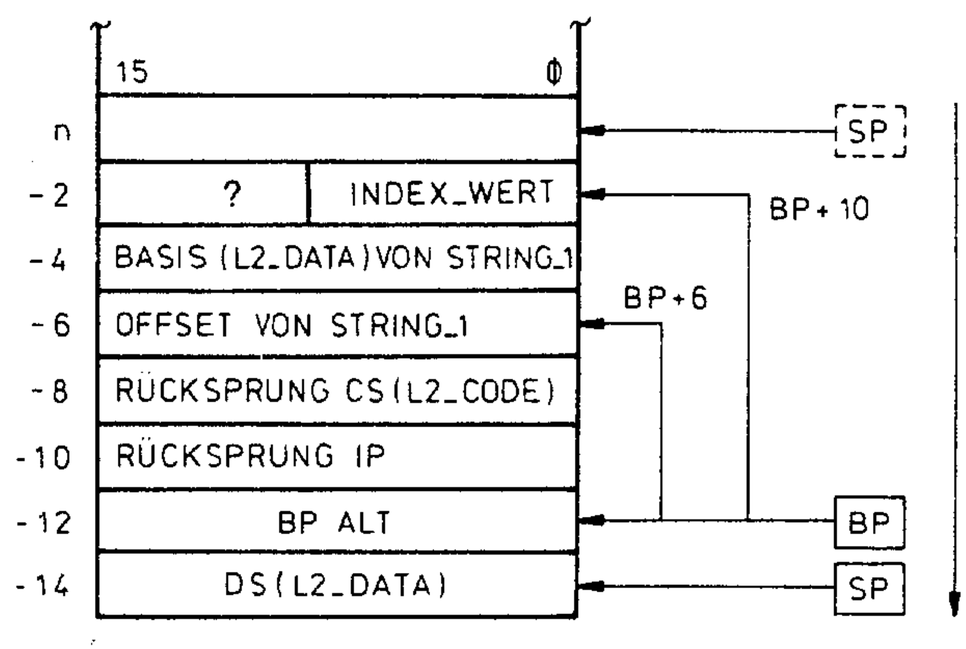


Nach der Standard-Konvention (alien) werden alle Parameter von links nach rechts an den Stack übergeben. Da die Prozedur erzeuge_eintrag auf die Variable NAECHSTER_INDEX im lokalen DATA_1-Segment DS-relativ zugreift (`MOV AL,NAECHSTER_INDEX`), ist es notwendig, den DS-Wert des C-Moduls (DS zeigt zum L2_DATA-Segment) in den Stack zu retten.

Hat erzeuge_eintrag die Befehle

```
PUSH    BP                          ;Prozedur-Prolog
MOV     BP,SP
PUSH    DS                          ;Selektor des C-Programm-
                                    ;Daten-Segments retten
```

ausgeführt, ergibt sich folgendes Stack-Layout:



Der erste Parameter index_wert, der als erstes Wort im neuen Eintrag der Daten-Tabelle im EXTRA_DATEN-Segment gespeichert werden soll, liegt bei der Position [BP+10]. Der zweite Parameter string_1 liegt bei der Position [BP+6] und ist ein kompletter Basis:Offset-Zeiger, der zum 6-Byte-ASCII-String 'EINGAB' im L2_DATA-Segment des C-Programms zeigt.

Wie bereits erwähnt, benützt erzeuge_eintrag die Variable NAECHSTER_INDEX im lokalen DATA_1-Segment. Daher wird als nächstes DS **umgeladen** mit:

```
MOV     AX,DATA_1                       ;AX=Selektor des
                                        ;   DATA_1-Segments
MOV     DS,AX                           ;DS initialisieren
```

Um den nächsten Eintrag der Daten-Tabelle im lokalen EXTRA_DATEN-Segment zu erreichen, wird anschließend genauso wie im SMALL-Modell die folgende Befehls-Sequenz ausgeführt:

```
MOV     AX,EXTRA_DATEN                  ;ES=Basis des
MOV     ES,AX                           ;   EXTRA_DATEN-Segments
MOV     AL,NAECHSTER_INDEX              ;AX=augenblicklicher
MOV     AH,0                            ;   Indexwert
MOV     DI,AX
SHL     DI,3                            ;DI=8*NAECHSTER_INDEX
                                        ;   Damit zeigt ES:DI zum
                                        ;   naechsten Eintrag
```

Bevor der 6-Byte-String 'EINGAB' des C-Programms in die Daten-Tabelle des lokalen EXTRA_DATEN-Segments übertragen wird, speichern folgende Befehle den index_wert im ersten Wort des neuen Eintrags:

```
MOV     BX,[BP+10]                      ;BX=index_wert (C-Variable)
MOV     ES:[DI],BX                      ;index_wert im naechsten
                                        ;Eintrag der Daten-Tab. speichern
```

Die String-Übertragung erledigt dann die folgende Befehlssequenz:

```
ADD     DI,2                            ;ES:DI zeigt zum String-Teil
                                        ;des Eintrags
LDS     SI,DWORD PTR [BP+6]             ;SI=Offset von string_1
                                        ;DS=Basis von string_1
                                        ;(C-Variable). DS:SI zeigt
                                        ;zum uebertragenden String
MOV     CX,3                            ;Der String ist 3 Worte
                                        ;(6 Bytes) lang
CLD                                     ;Direction-Flag=0 bedeutet
                                        ;SI und DI inkrementieren
REP MOVS ES:WORD PTR[DI],DS:[SI]        ;Uebertrage String in die
                                        ;Daten-Tabelle
```

Im Gegensatz zum SMALL-Modell, ist es im LARGE-Modell erforderlich, daß zur Adressierung des Strings 'EINGAB' (C-Variable) der komplette Basis:Offset-Zeiger auf string_1 an die Assembler-FAR-Prozedur erzeuge_eintrag übergeben werden muß. Dies erledigt der Befehl `LDS SI,DWORD PTR [BP+6]`.

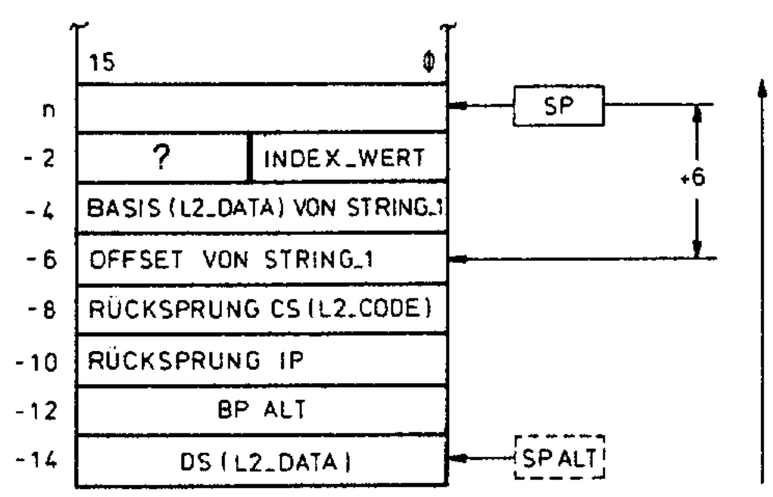
Nach seiner Ausführung zeigt DS zum externen L2_DATA-Segment des C-Programms und SI liefert den Offset von string_1. Das Register DS ist also an dieser Stelle **umgeladen** worden. Da aber vor dem Verlassen der Prozedur die lokale Variable NAECHSTER_INDEX inkrementiert wird, muß DS zum lokalen DATA_1-Segment zeigen. Daher ist die folgende Befehlssequenz notwendig:

```
MOV     AX,DATA_1               ;AX=Selektor des
                                ;   DATA_1-Segments
MOV     DS,AX                   ;DS initialisieren
INC     NAECHSTER_INDEX         ;Naechsten Eintrag in der
                                ;Daten-Tabelle indizieren
```

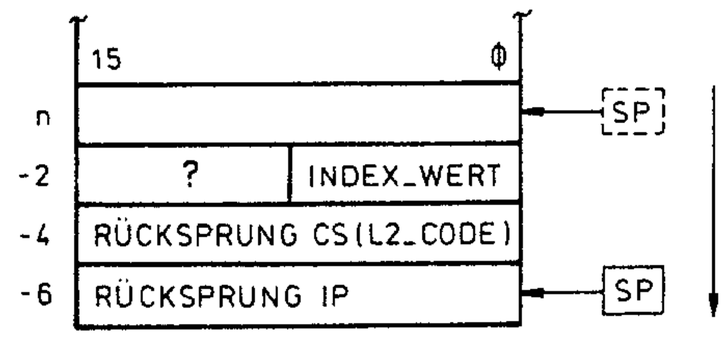
Im Epilog der Prozedur muß der im Prolog gerettete DS-Wert des C-Programm-Daten-Segments (L2_DATA) wieder in das DS-Register zurückgeschrieben und die Parameter aus dem Stack entfernt werden. Dies erledigen die Befehle

```
POP     DS                      ;Selektor des C-Programm-Daten-
                                ;Segments zurueckschreiben
        POP     BP              ;Prozedur-Epilog
END_1:  RET     6               ;Entferne die Parameter
                                ;index_wert (2 Bytes) und
                                ;string_1 (4 Bytes) aus
                                ;dem Stack und zurueck zum
                                ;C-Programm
```

Die Stack-Situation am Ende des Prozedur-Prologs zeigt folgendes Bild:



Nach der Rückkehr ins C-Programm, wird mit `nachsehen_eintrag (index_wert)` die externe Assembler-FAR-Prozedur nachsehen_eintrag aufgerufen und die Parameter index_wert an den Stack übergeben. Nach dem Aufruf der Assembler-Prozedur ergibt sich folgendes Stack-Layout:



Genauso wie im SMALL-Modell benützt die Prozedur nachsehen_eintrag die POP-Methode, um auf den Parameter index_wert zuzugreifen. Da aber im LARGE-Modell ein kompletter 32-Bit-Rücksprung-Zeiger CS:IP im Stack hinterlegt wird, benützt nachsehen_eintrag die Register SI und DI für die Zwischenspeicherung der 32-Bit-Rücksprungadresse. Den index_wert überträgt die Prozedur in das Register BX. Dies alles erledigt die folgende Befehlssequenz:

```
POP     SI                          ;SI=Ruecksprungadresse (IP)
POP     DI                          ;DI=Ruecksprungadresse (CS)
POP     BX                          ;BX (BL)=Parameter index_wert
                                    ;(0..255)
PUSH    DI                          ;Ruecksprungadresse (CS) und
PUSH    SI                          ;Ruecksprungadresse (IP) in den
                                    ;Stack zurueckschreiben
```

Da die Prozedur nachsehen_eintrag auf das lokale DATA_1-Segment **nicht** zugreift, muß sie den augenblicklichen DS-Wert (DS zeigt zum L2_DATA-Segment des C-Programms) auch **nicht** retten.

Sie benützt aber genauso wie im SMALL-Modell ES, BX und DI, um den zurückzulesenden Eintrag zu markieren. Die folgende Befehlssequenz ist daher die gleiche wie im SMALL-Modell:

```
MOV     AX,EXTRA_DATEN              ;ES=Basis des
MOV     ES,AX                       ;EXTRA_DATEN-Segments
MOV     BH,0                        ;ES:BX zeigt auf den zurueckzu-
SHL     BX,3                        ;lesenden Eintrag
MOV     DI,0                        ;DI=Offset des ersten Worts
                                    ;   im Eintrag
MOV     CX,4                        ;4 Worte (8 Bytes) uebertragen
```

Für die Übertragung eines 8-Byte-Eintrags **von** der Daten-Tabelle im lokalen EXTRA_DATEN-Segment **in** den C-Character-Vektor eintrag_puffer im externen L2_DATA-Segment benützt die Assembler-Prozedur die folgenden Befehle:

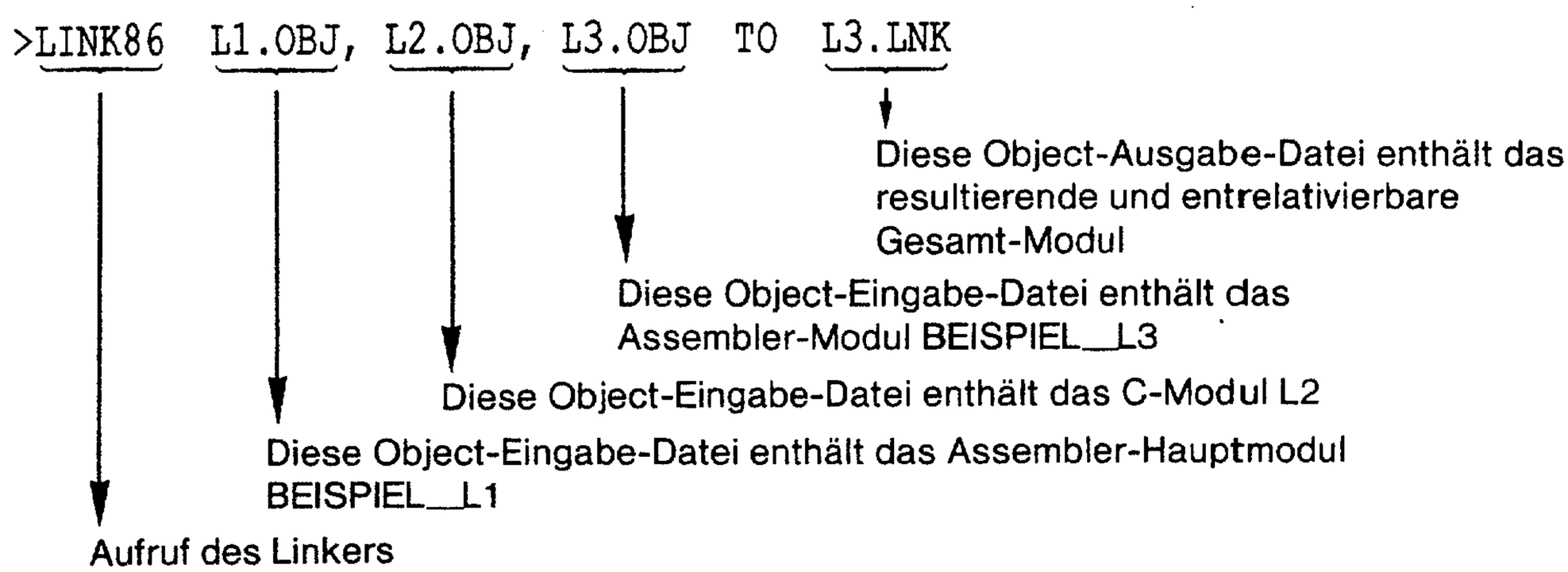
```
BEKOMMEN_EINTRAG:
ASSUME DS:SEG eintrag_puffer
        MOV     AX,ES:[BX][DI]                  ;Wort von Daten-Tabelle im
                                                ;EXTRA_DATEN-Segment lesen
        MOV     WORD PTR eintrag_puffer[DI],AX  ;Wort in C-Character-Vektor
                                                ;eintrag_puffer uebertr.
        ADD     DI,2                            ;Naechst. Wort adressieren
        LOOP    BEKOMMEN_EINTRAG                ;Wiederhole, bis 8 Bytes
                                                ;(4 Worte) uebertragen
END_2:  RET                                     ;Zurueck zum C-Programm
```

Wie zu erkennen ist, wird für die komplette Übertragung eines 8-Byte-Eintrags der Befehl `MOV WORD PTR eintrag_puffer_[DI],AX` 4 mal ausgeführt. Dabei adressiert der Prozessor die Variable eintrag_puffer DS-relativ. Da aber im Augenblick DS mit dem lokalen DATA_1-Segment verbunden ist, kann der Assembler die Variable eintrag_puffer **nicht** erreichen.

Um nun dem Assembler mitzuteilen, in welchem Segment sich die Variable eintrag_puffer aufhält, kann der **SEG**-Operator in einer neuen **ASSUME**-Anweisung benützt werden. Dabei teilt `ASSUME DS:SEG eintrag_puffer` dem Assembler mit, daß DS auf das Segment zeigt, in dem die Variable eintrag_puffer enthalten ist. Im Beispiel ist dies das C-Programm-Daten-Segment L2_DATA. Da DS nicht umgeladen werden mußte, kann nun die Übertragung des 8-Byte-Eintrags problemlos erfolgen.

Die oben beschriebenen Korrespondenzen kann der Prozessor nur realisieren, wenn ihm ein ablauffähiges Programm zur Verfügung steht. Aus diesem Grund müssen die beteiligten Module in der richtigen Reihenfolge gebunden werden.

Dies erledigt der INTEL-Linker LINK86.EXE mit folgendem Kommando:



Da der Linker die physikalischen Positionen der Segmente nicht festlegt und damit kein ablauffähiges Ausgabe-Modul liefert, ist anschließend ein Locater-Lauf erforderlich. Das folgende Locater-Kommando zeigt hierfür ein Beispiel:

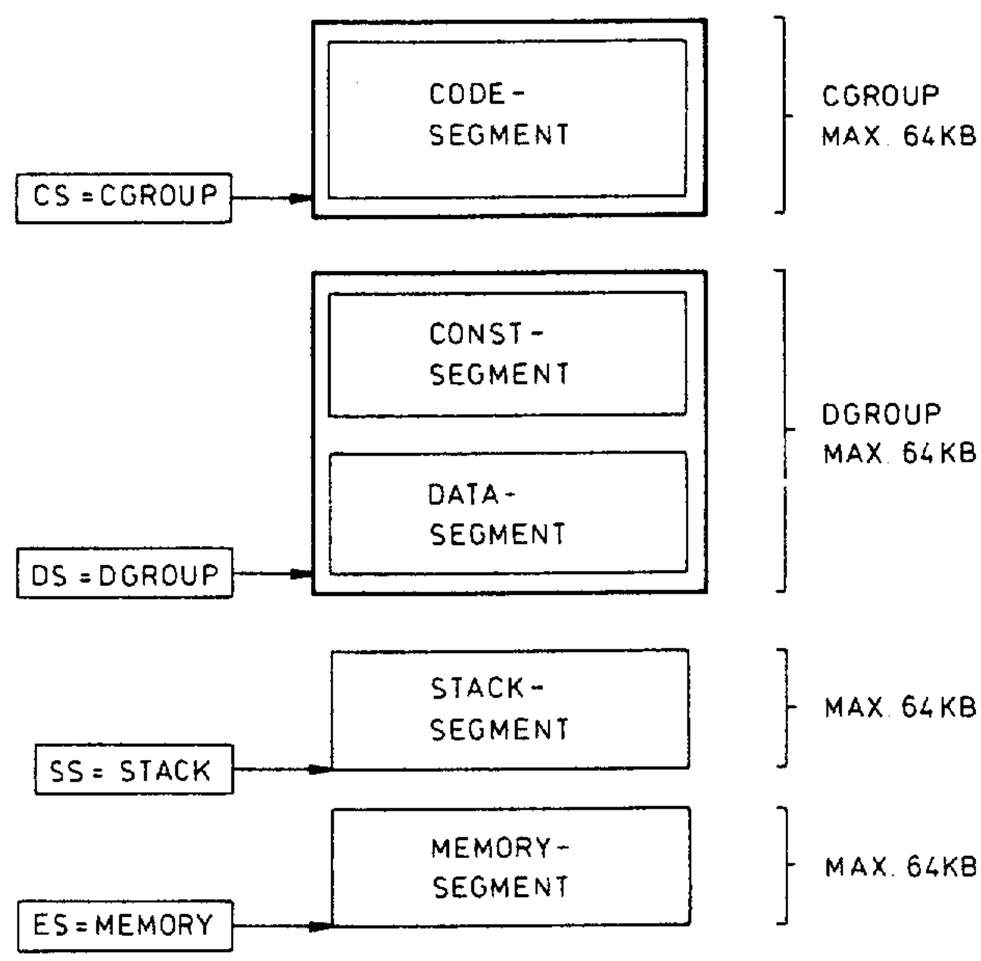
```
>LOC86 L3.LNK TO L3.LOC AD(SM(CODE(0FC000H),L2_CODE(0FC100H),&
>>CODE_1(0FC200H),L2_DATA(100H),DATA_1(200H),STACK(300H),&
>>EXTRA_DATEN(400H))) START(START)
```

LOC86	Aufruf des Locaters LOC86.EXE
L3.LNK	OBJECT-Eingabe-Datei enthält das entrelativierbare Modul
L3.LOC	OBJECT-Ausgabe-Datei enthält entrelativiertes und damit ablauffähiges Programm
AD	Schlüsselwort für ADDRESSES
SM	Schlüsselwort für SEGMENTS
CODE	Name des Code-Segments von Modul BEISPIEL_L1
(0FC000H)	Physikalische Basis-Adresse des CODE-Segments
L2_CODE	Name des Code-Segments von Modul L2 (C-Modul)
(0FC100H)	Physikalische Basis-Adresse des L2_CODE-Segments
&	Kommando-Verlängerung
>>	Rückmeldung nach Kommando-Verlängerung
CODE_1	Name des Code-Segments von Modul BEISPIEL_L3
(0FC200H)	Physikalische Basis-Adresse des CODE_1-Segments
L2_DATA	Name des Daten-Segments von Modul L2 (C-Modul)
(100H)	Physikalische Basis-Adresse des L2_DATA-Segments
(DATA_1)	Name des Daten-Segments von Modul BEISPIEL_L3
(200H)	Physikalische Basis-Adresse des DATA_1-Segments

STACK	Name des Stack-Segments (LARGE-Modell)
(300H)	Physikalische Basis-Adresse des STACK-Segments
&	Kommando-Verlängerung
>>	Rückmeldung nach Kommando-Verlängerung
EXTRA_DATEN	Name des zusätzl. Daten-Segments im Modul BEISPIEL_L3
(400H)	Physikalische Basis-Adresse des EXTRA_DATEN-Segments
START	Teilt Locater die Start-Position des 1. Befehls nach RESET mit
(START)	PUBLIC-Symbol (Start-Label) des Assembler-Hauptmoduls

8.3 Das COMPACT-Modell

Dieses Modell unterscheidet sich nur geringfügig vom SMALL-Modell. Wie im folgenden Bild zu sehen ist, hält sich das CODE-Segment in der CGROUP (maximal 64 KBytes) auf und die DATA- und CONST-Segmente halten sich in der DGROUP (maximal 64 KBytes) auf. Die STACK- und MEMORY-Segmente gehören allerdings zu **keiner** Group. Sie stehen **alleine** und besetzen jeweils maximal 64 KBytes des Speichers.



Beim COMPACT-Modell ändern sich die Segment-Register nicht:

- Das CS-Register enthält die Basis-Adresse der CGROUP,
- das DS-Register enthält die Basis-Adresse der DGROUP und
- das SS-Register enthält die Basis-Adresse des STACK-Segments.
- das MEMORY-Segment kann über das ES-Register adressiert werden

In Anlehnung an das Speicher-Layout eines COMPACT-Programms ist für ein Assembler-Programm folgendes 'Rahmenwerk' definiert:

COMPACT-Modell der Segmentierung:

```
 NAME  <Modul-Name>
*CGROUP  GROUP  *CODE
*DGROUP  GROUP  *CONST,*DATA
 ASSUME  CS:*CGROUP,DS:*DGROUP,SS:STACK
*CONST   SEGMENT  PUBLIC  'CONST'
       <Hier können Programm-Konstanten definiert werden>
*CONST   ENDS
*DATA    SEGMENT  PUBLIC  'DATA'
 EXTRN <externe Variablen>
       <Hier können die Programm-Variablen definiert werden>
*DATA    ENDS
 STACK   SEGMENT  STACK   'STACK'
       <Hier ist das DW-Statement zu benützen, um die Länge
       des 'Run Time'-Stack festzulegen>
 STACK   ENDS
 MEMORY  SEGMENT  MEMORY  'MEMORY'
       <Hier können spezielle Programm-Daten oberhalb der
       anderen Segmente definiert werden>
 MEMORY  ENDS
*CODE    SEGMENT  PUBLIC  'CODE'
 EXTRN <Externe NEAR-Labels, wie z.B Prozedur-Namen>
       <Hier wird der Programm-Code aufbewahrt>
*CODE    ENDS
 END   <Hier kann zur Definition eines Haupt-Moduls die
        Startadresse des Programms angegeben werden>
```

Hinweis: * bedeutet, daß auch andere Namen benützt werden können

Das 'Rahmenwerk' zeigt exakt, welche SEGMENT-Statements für die CODE-, DATA-, STACK-, CONST- und MEMORY-Segmente und welche GROUP- und ASSUME-Statements in einem Assembler-Modul angegeben werden müssen. Als Align-Typ ist PARA vordefiniert und kann geändert werden.

Da die CODE-Segmente der individuellen Module zur CGROUP kombiniert werden, sollten alle Prozeduren den Typ **NEAR** haben. Die Offsets der Variablen sind DGROUP-relativ.

Daher muß der 'Group Override'-Operator **DGROUP:** zusammen mit dem **OFFSET**-Operator benützt werden, wenn die Position einer Variablen innerhalb der DGROUP bestimmt werden soll.

8.3.1 Stack-Layout nach der Parameter-Übergabe

Das folgende C-Programm-Fragment soll im COMPACT-Modell übersetzt werden:

```
int  wpara;
char bpara;
int  *ppara;
beispiel()
{
alien dreiparameter();
drei_parameter(wpara,bpara,ppara);
}
```

Wie zu sehen ist, ruft das C-Programm die Funktion drei_parameter auf, die als Assembler-**NEAR**-Prozedur in einem anderen Modul existieren soll.

Dabei übergibt das C-Programm an die Assembler-Prozedur den **Word**-Parameter wpara (Typ int), den **Byte**-Parameter bpara (Typ char) und den Zeiger (*ppara), der auf eine **Integer**-Variable zeigt.

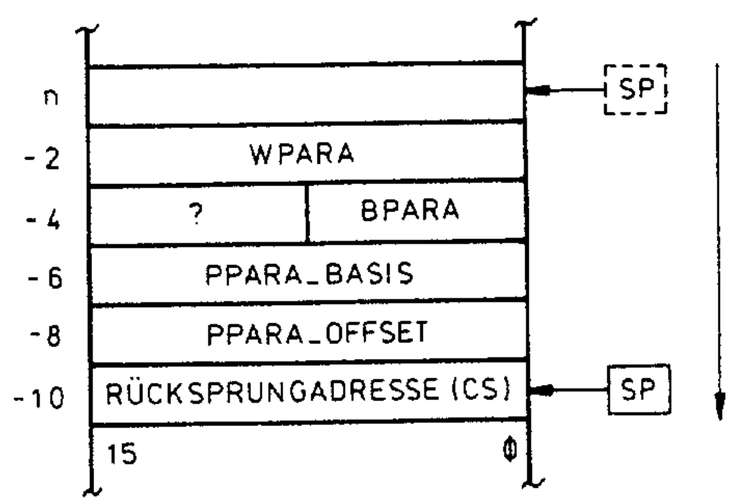
Wegen des alien-Attributs werden die Parameter nach der Standard-Konvention von links nach rechts an den Stack übergeben.

Der Code, den der C-Compiler im COMPACT-Modell für `drei_parameter(wpara,bpara,ppara)` liefert, sieht im wesentlichen folgendermaßen aus:

C-Code (COMPACT-Modell):

```
        .
        .
PUSH    wpara                   ;Word-Parameter wpara (Typ int)
PUSH    bpara                   ;Byte-Parameter bpara (Typ char)
PUSH    ppara_basis             ;Basis-Komponente des Long-Pointer-
                                ;Parameters *ppara
PUSH    ppara_offset            ;Offset-Komponente des Long-Pointer-
                                ;Parameters *ppara
CALL    drei_parameter          ;Aufruf der Assembler-NEAR-Prozedur
        .
        .
```

Zugehöriges Stack-Layout (Compact-Modell):



Wie im C-Code zu sehen ist, setzt der C-Compiler für die Übergabe des Zeiger-Parameters ppara an den Stack die folgenden beiden Befehle ab `PUSH ppara_basis` und `PUSH ppara_offset`.

Dies bedeutet, daß ein kompletter **Basis:Offset**-Zeiger an den Stack übergeben wird. Diese Tatsache läßt die Adressierung von Variablen zu, die sich außerhalb der DGROUP in einem zusätzlichen Daten-Segment aufhalten.

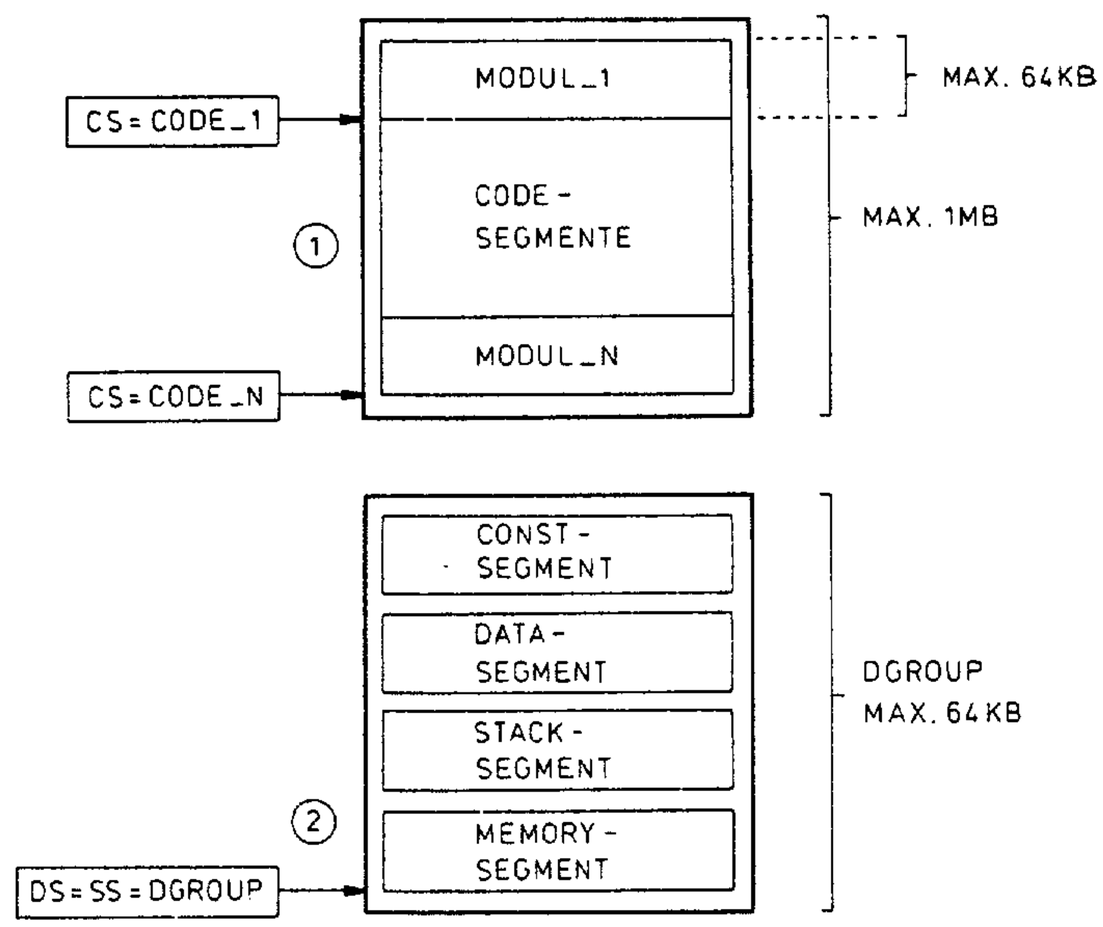
8.4 Das MEDIUM-Modell

Dieses Modell ist eine weitere Variante des SMALL-Modells. Wie im folgenden Bild zu sehen ist, halten sich genauso wie im SMALL-Modell das CONST-, DATA-, STACK- und MEMORY-Segment in der DGROUP auf. Allerdings gibt es **keine** CGROUP.

Dies bedeutet, daß jedes Modul sein eigenes **nicht-kombinierbares** Code-Segment hat. Das MEDIUM-Modell erlaubt daher eine große Menge von Programm-Code. Bei Inter-Modul-Aufrufen benützt das MEDIUM-Modul die **lange** Form des CALL-Befehls.

Dies hat zur Folge, daß sowohl CS als auch IP umgeladen werden. Aus diesem Grund müssen externe PUBLIC-Prozeduren vom Typ **FAR** sein. Als Rücksprungadressen werden dann automatisch komplette CS:IP-Zeiger im Stack hinterlegt.

Da jedes Modul sein eigenes Code-Segment hat, haben lokale Prozeduren den Typ **NEAR**. Das folgende Bild zeigt das komplette Speicher-Layout des MEDIUM-Modells:



Wie zu erkennen ist, enthält das CS-Register die Basis-Adresse des lokalen Code-Segments ① und das DS- und SS-Register die Basis-Adresse der DGROUP ②.

Genauso wie im SMALL-Modell, läßt sich das MEDIUM-Modell um ein zusätzliches Daten-Segment erweitern. In diesem Fall enthält das ES-Register die Basis-Adresse des lokalen Extra-Daten-Segments.

In Anlehnung an das Speicher-Layout eines MEDIUM-Programms ist für ein Assembler-Programm folgendes 'Rahmenwerk' definiert:

MEDIUM-Modell der Segmentierung:

```
NAME   <Modul-Name>
DGROUP  GROUP  CONST,DATA,STACK,MEMORY
ASSUME  CS:*CODE,DS:DGROUP,SS:DGROUP
CONST   SEGMENT  PUBLIC  'CONST'
      <Hier können Programm-Konstanten definiert werden>
CONST   ENDS
DATA    SEGMENT  PUBLIC  'DATA'
EXTRN <externe Variablen>
      <Hier können die Programm-Variablen definiert werden>
DATA    ENDS
STACK   SEGMENT  STACK   'STACK'
      <Hier ist das DW-Statement zu benützen, um die Länge
      des 'Run Time'-Stack festzulegen>
STACK   ENDS
MEMORY  SEGMENT  MEMORY  'MEMORY'
      <Hier können spezielle Programm-Daten oberhalb der
      anderen Segmente definiert werden>
MEMORY  ENDS
 EXTRN <Externe FAR-Labels, wie z.B Prozedur-Namen>
*CODE    SEGMENT   'CODE'
      <Hier wird der Programm-Code aufbewahrt>
*CODE    ENDS
 END   <Hier kann zur Definition eines Haupt-Moduls die
      Startadresse des Programms angegeben werden>
```

Hinweis: * bedeutet, daß auch andere Namen benützt werden können

Das 'Rahmenwerk' zeigt exakt, welche SEGMENT-Statements für die CODE-, DATA-, STACK-, CONST- und MEMORY-Segmente und welche GROUP- und ASSUME-Statements in einem Assembler-Modul angegeben werden müssen. Als Align-Typ ist PARA vordefiniert und kann geändert werden.

Die Offsets der Variablen sind DGROUP-relativ. Daher muß der 'Group Override'-Operator **DGROUP:** zusammen mit dem **OFFSET**-Operator benützt werden, wenn die Position einer Variablen innerhalb der DGROUP bestimmt werden soll.

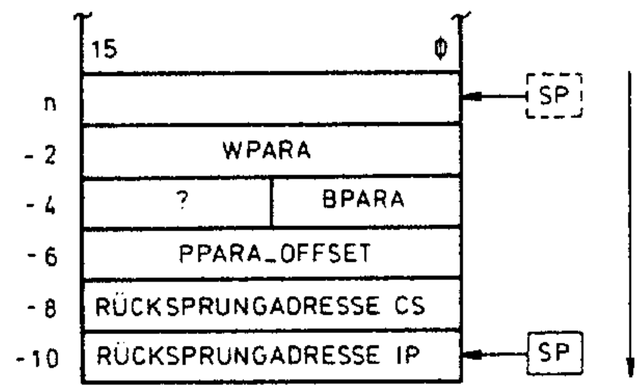
8.4.1 Stack-Layout nach der Parameter-Übergabe

Das folgende C-Programm-Fragment soll im MEDIUM-Modell übersetzt werden:

```
int  wpara;
char bpara;
int  *ppara;
beispiel()
{
alien dreiparameter();
drei_parameter(wpara,bpara,ppara);
}
```

Wie zu sehen ist, ruft das C-Programm die Funktion drei_parameter auf, die als Assembler-**FAR**-Prozedur in einem anderen Modul existieren soll. Dabei übergibt das C-Programm an die Assembler-Prozedur den **Word**-Parameter wpara (Typ int), den **Byte**-Parameter bpara (Typ char) und den Zeiger (*ppara), der auf eine **Integer**-Variable zeigt. Wegen des alien-Attributs werden die Parameter nach der Standard-Konvention von links nach rechts an den Stack übergeben. Der Code, den der C-Compiler im MEDIUM-Modell für `drei_parameter(wpara,bpara,ppara)` liefert, sieht im wesentlichen folgendermaßen aus:

C-Code (MEDIUM-Modell):

```
        .
        .
PUSH    wpara                   ;Word-Parameter wpara (Typ int)
PUSH    bpara                   ;Byte-Parameter bpara (Typ char)
PUSH    ppara_offset            ;Offset-Komponente des Short-Pointer-
                                ;Parameters *ppara
CALL    drei_parameter          ;Aufruf der Assembler-FAR-Prozedur
        .
        .
```

Zugehöriges Stack-Layout (MEDIUM-Modell):