4 „Tasks“ und Status-Wechsel

„Tasks“ sind aktuell zur Ausführung kommende Befehlssequenzen, während man „Programme“ ein Reservoir an Befehlssequenzen nennen kann, die momentan nicht ausgeführt werden und auch in nicht ausführbarer Form vorliegen können, z. B. als Quellcode.

Ein Programm wird zur Task, sobald es für die Ausführung geeignet ist, etwa, indem sein Quellcode mit einem Compiler, einem Assembler oder einem Programmlader in eine ausführbare Form umgewandelt, und zusätzlich das Betriebssystem davon unterrichtet wurde, daß eine Task für Installation und Ausführung bereitsteht.

Der Unterschied zwischen Programmen und Tasks ist am Beispiel von Multitasking-Systemen am besten zu verstehen, in denen zwei oder noch mehr „Tasks“ ein und dasselbe „Programm benützen.

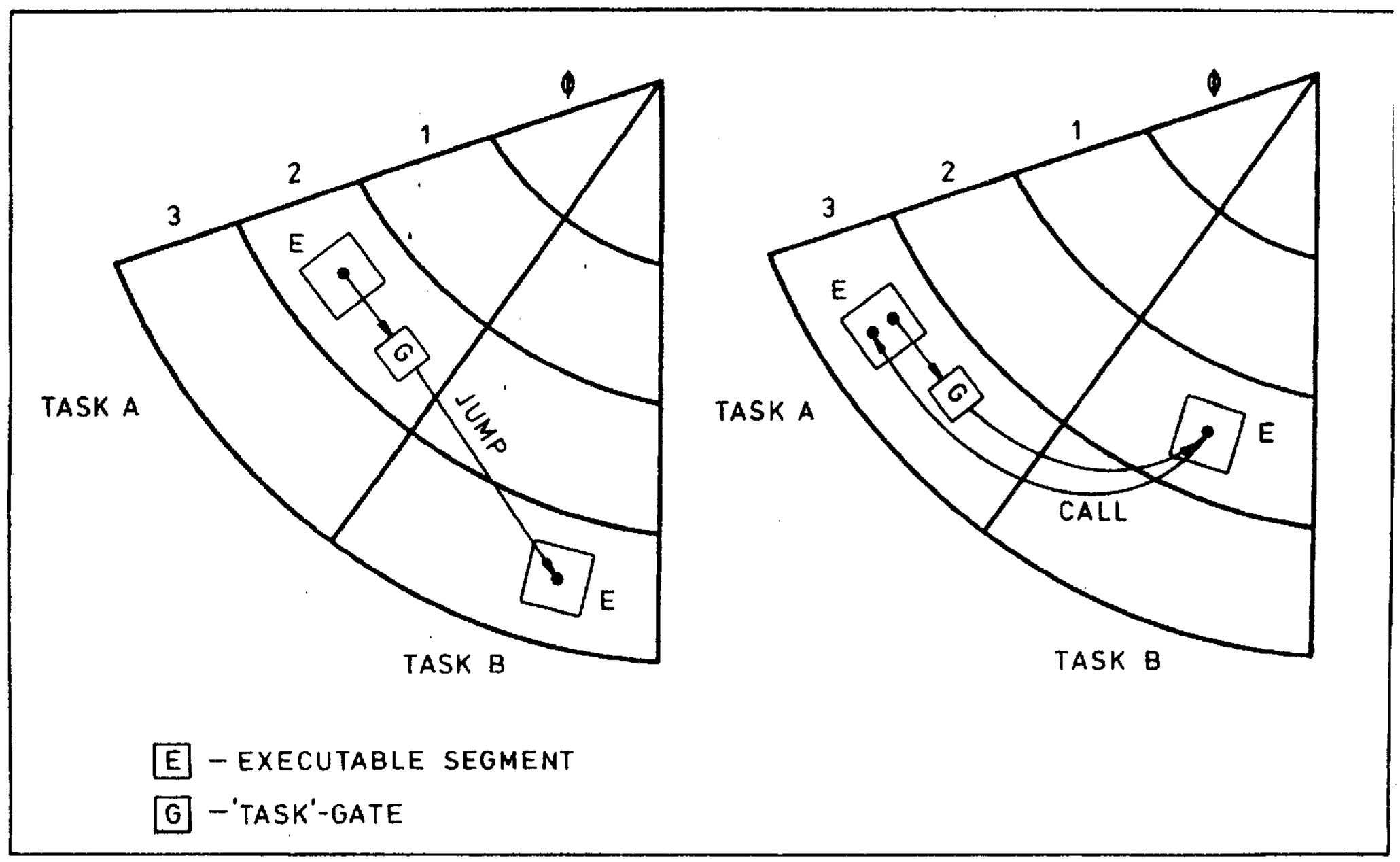
Nehmen wir den Fall eines Editor-Programms. Obwohl jede Editor-Task dasselbe Editor-Programm benützt, liefert sie dennoch unterschiedliche Ergebnisse, da jede Editor-Task das Editor-Programm mit unterschiedlichen Eingaben versorgt.

In einem 80486-System können viele Tasks mit der CPU in Verbindung gebracht werden, aber es wird nur jeweils eine Task zu einem bestimmten Zeitpunkt von der CPU ausgeführt.

Um die CPU von der Auführung der einen Task in die Ausführung einer anderen umzuschalten, ist eines der folgenden Ereignisse erforderlich:

- ein Interrupt
- ein „Inter-Task“-CALL-Befehl
- ein „Inter-Task“-JMP-Befehl
- ein IRET-Befehl

Das folgende Bild illustriert einige Task-Wechsel mit Hilfe der Befehle FAR JMP und FAR CALL. Dabei haben diese Befehle die Möglichkeit, verschiedene Task-Wechsel-Mechanismen zu aktivieren. In beiden Beispielen benützen sie spezielle Gates, sogenannte **„Task-Gates“**, deren Funktion später in diesem Kapitel behandelt wird.



Das Umschalten von einer Task auf eine andere ist die wichtigste Funktion eines Mult-User- oder Multitasking-Betriebssystems.

Da Task-Wechsel in einem Realzeit-Multitasking-System sehr oft vorkommen, ist in der 80486-CPU eine spezielle Hochgeschwindigkeits-Hardware enthalten, die diese Operation unterstützt.

4.1 Der Task-Status

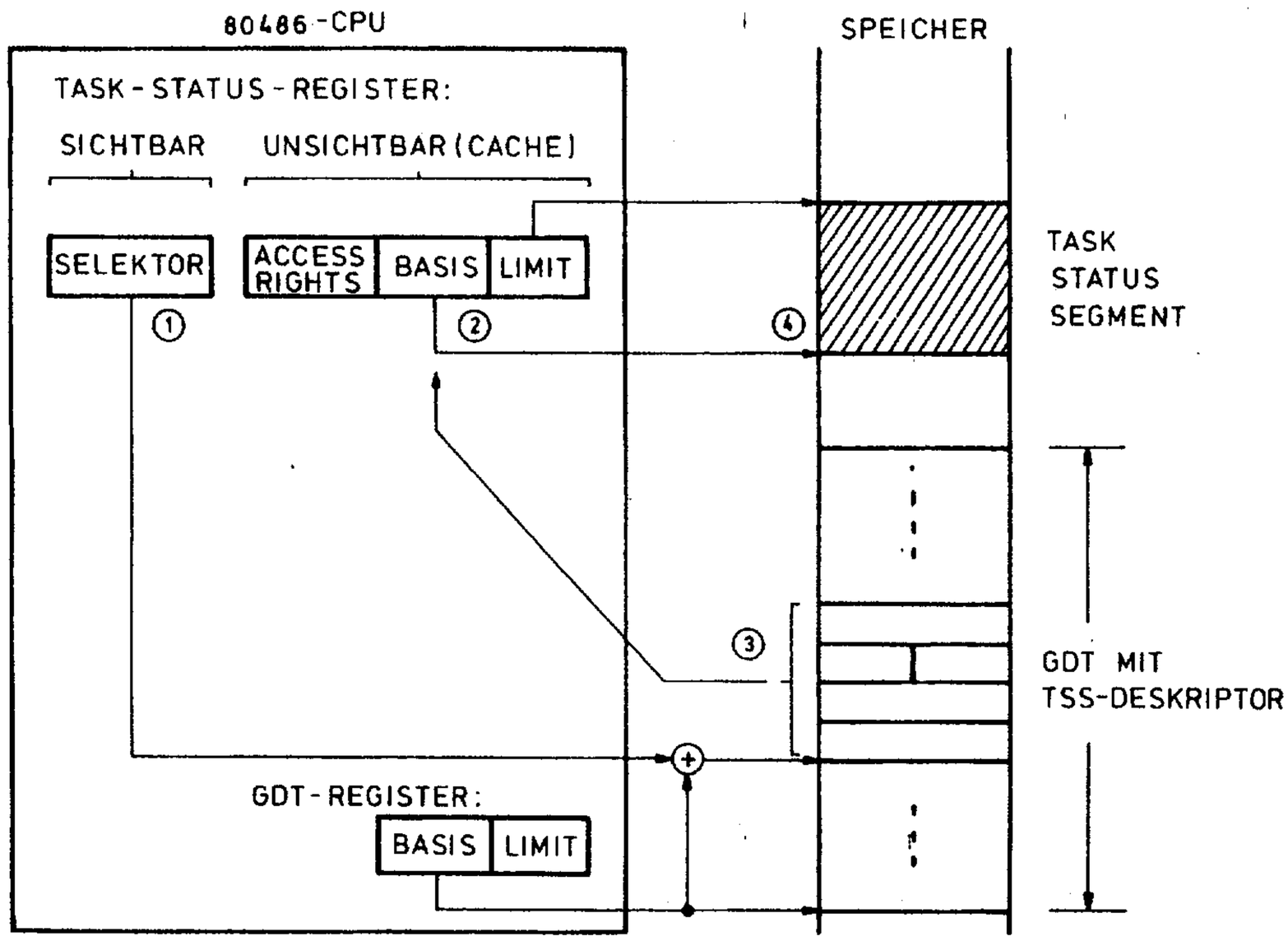
Der Status einer Task ist durch die Inhalte der CPU-Register definiert, die von der Task benützt werden. Die Architektur des 80486 enthält einen speziellen Segmenttyp, das sogenannte ***T***ask-***S***tatus-***S***egment (TSS), um den Status einer Task zu speichern.

Das Task-Status-Segment ist wie die GDT, die LDT oder die IDT ein ***System-Segment,*** das von einem speziellen Deskriptor beschrieben wird. Da dieser sogenannte TSS-Deskriptor jederzeit erreichbar sein muß, darf er sich ***nur*** in der ***GDT*** aufhalten.

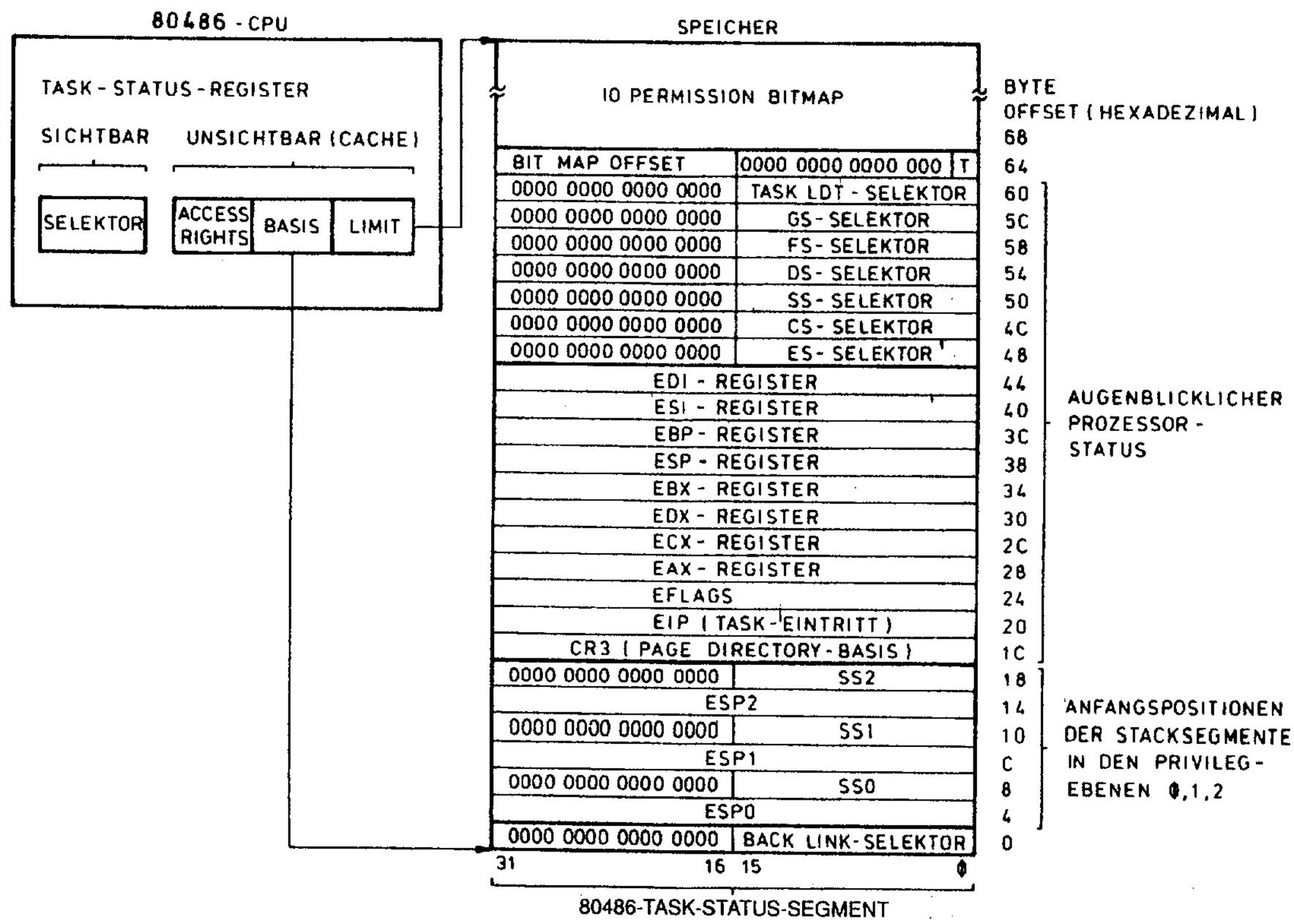
In einem Multitasking-System, existiert eine Vielzahl von Tasks, die durch individuelle TSS-Deskriptoren spezifiziert sind. Da nur jeweils eine Task aktiv ist, wird die Position des zugehörigen Task-Status-Segments im CPU-***T***ask-***R***egister TR aufbewahrt. Wie das folgende Bild zeigt, besteht das Task-Register aus zwei Teilen, und zwar:

- ***dem „sichtbaren" Teil,*** den die Task erreichen kann ①. Er enthält einen Selektor, um in der GDT den Deskripor des augenblicklichen Task-Status-Segments auszuwählen.
- ***dem „unsichtbaren" Teil,*** der das aktuelle Task-Status-Segment beschreibt ②. Immer dann, wenn sich der „sichtbare" Teil des Task-Status-Registers ändert, lädt

die CPU automatisch den vom Selektor ausgewählten TSS-Deskriptor in den „unsichtbaren" Teil des Registers ③. Damit ist die Basisadresse und die Segmentobergrenze de augenblicklichen Task-Status-Segments spezifiziert ④.



Damit die 80486-CPU bei einem Task-Wechsel den Task-Status automatisch umspeichern kann, benötigt sie für das Task-Status-Segment ein spezielles Format. Das folgende Bild illustriert ein Task-Status-Segment einschließlich der mit ihr verbundenen Hardware-Struktur.



Ein Task-Status-Segment ist irgendwo im linearen Adreßraum aufbewahrt und enthält alle Informationen, die der Prozessor benötigt, um eine Task zu managen. Wie das Format dieses Segments zeigt, besteht es aus mindestens 26 Feldern zu je 32 Bits, die in zwei Klassen unterteilt sind:

Dynamische Felder:

Sie enthalten

- die allgemeinen Register EAX, ECX, EDX, EBX, ESP, EBP, ESI und EDI.
- Die Segment-Register ES, CS, SS, DS, FS und GS.
- Das Flag-Register EFLAGS.
- Den „Instruction Pointer" EIP und
- den „BACK LINK"-Selektor zum Task-Status-Segment der aufrufenden Task.

Statische Felder:

Sie enthalten

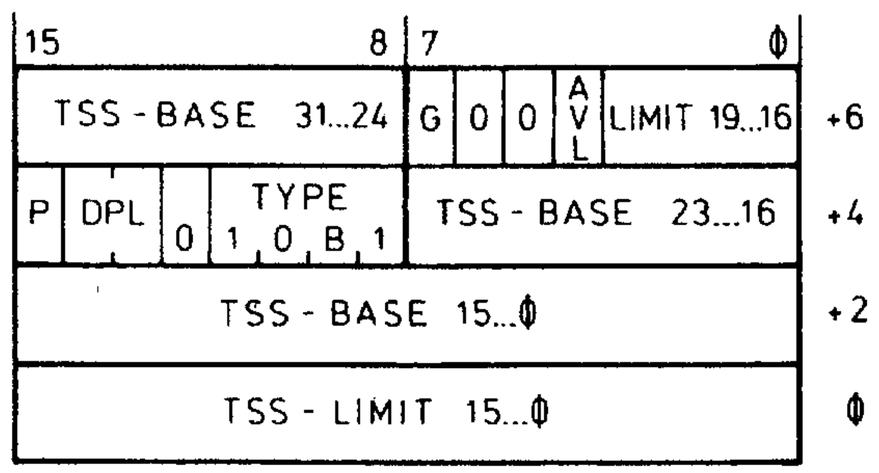
- den Task-LDT-Selektor zur LDT der augenblicklichen Task.
- Das Register CR3, das die Basisadresse der „Page Directory"-Tabelle für die augenblickliche Task enthält. Dieses Feld ist nur dann aktiv, wenn der Paging-Mechanismus eingeschaltet ist.
- Die Zeiger SS0:ESP0, SS1:ESP1 und SS2:ESP2, die die Anfangspositionen der Stack-Segmente in den Privileg-Ebenen 0–2 festlegen.
- Das T-Bit (Debug Trab Bit), das bei einem Task-Wechsel eine sogenannte „Debug Exception" bewirkt (mehr Informationen hierzu im Kapital „Debug-Unterstützung").
- den I/O Map Offset.

Die Bedeutungen dieser Felder werden im Laufe der nächsten Abschnitte dieses Kapitels näher erläutert.

4.2 Task-Status-Segment-Deskriptoren

Wie bereits erwähnt, werden zur Spezifikation von Task-Status-Segmenten TSS-Deskriptoren benützt. Das komplette Layout eines Task-Status-Segment-Deskriptors zeigt das folgende Bild.

TSS - DESKRIPTOR:



Genauso wie ein Daten- oder Code-Segment-Deskriptor, enthält der TSS-Deskriptor ein Basisadreß-Feld und ein Limit-Feld. Der Limit-Wert muß mindestens 0067H (103) sein, damit die für ein Task-Status-Segment erforderliche Minimalmenge von Informationen gespeichert werden kann. Der Versuch, in eine Task zu wechseln, deren TTS-Limit-Wert kleiner als 103 ist, stellt eine Ausnahmesituation dar und führt zur Unterbrechung des Programms.

Das P-Bit (Present) gibt an, ob dieser Deskriptor gültige Informationen enthält. Dabei bedeutet P = 1 ja und P = 0 nein. Sollte ein Task-Wechsel auf ein nicht-präsentes Task-Status-Segment (P = 0) versucht werden, stellt auch dies eine Ausnahmesituation dar.

Die Deskriptor-Privileg-Ebene DPL überwacht die Benützung des Task-Status-Segments durch einen JMP- oder CALL-Befehl. So kann DPL ein Programm daran ***hindern,*** über ein Task-Status-Segment einen Task-Wechsel auszulösen.

Task-Status-Segment-Deskriptoren können ***zwei*** Zustände haben:

- Ruhestand
- aktiver Zustand

Diese Zustände werden im Type-Feld des Access-Bytes spezifiziert, wobei für die Unterscheidung das B-Bit (Busy-Bit) benützt wird:

- Type-Code = B markiert die „Task" als aktiv (Busy-Bit = 1),
- Type-Code = 9 markiert die „Task" als inaktiv (Busy-Bit = 0).

Die Unterscheidung ist notwendig, da eine aktive Task ***nicht*** aufgerufen werden darf.

4.3 Initialisierung des Task-Registers

Um das Task-Register zu initialisieren bzw. seinen Inhalt zu ändern, gibt es zwei Möglichkeiten:

- ***Durch den LTR-Befehl (Load Task Register).*** Dieser Befehl wird benützt, um dem Task-Register während der Systeminitialisierung seinen Anfangswert zu geben.
- ***Durch eine der Task-Wechsel-Operationen.*** Diese werden später in diesem Kapitel beschrieben.

An dieser Stelle soll zunächst der LTR-Befehl betrachtet werden:

- **LTR ew** ;Load Task Register

Der Befehl bewirkt, daß das Task-Register vom Operanden ew geladen wird. Dabei enthält ew den Selektor eine Task-Status-Segments und spezifiziert entweder ein Word-Quellregister oder einen Word-Speicheroperanden.
Ist der Befehl ausgeführt, enthält der „unsichtbare" Teil des Task-Registers den Deskriptor eines Task-Status-Segments, wobei dieses dann als „Busy" markiert wird (Type-Code = B).

Der LTR-Befehl erscheint nur in Prozeduren der Privileg-Ebene 0 in der Betriebssystem-Software und wird ***nicht*** von Anwenderprogrammen benützt.

Sollte er im „Real Address Mode" ausgeführt werden, stellt dies eine Ausnahmesituation dar, die zur Programmunterbrechung führt.

4.3.1 **Progammierbeispiel:** Task-Register laden

Das folgende ASM386/486-Programm-Fragment zeigt, wie der LTR-Befehl benützt werden kann.

```
LOC        OBJ            LINE  SOURCE
                            1        NAME      B16
                            2
                            3  ;Definiere das Layout fuer einen Deskriptor
                            4
--------                    5  DESK            STRUC
00000000                    6  LIMIT_0_15      DW      0          ;Segment-Obergrenze (Bits 0..15)
00000002                    7  BASIS_0_15      DW      0          ;Basisadresse (Bits 0..15)
00000004                    8  BASIS_16_23     DB      0          ;Basisadresse (Bits 16..23)
00000005                    9  ZUGRIFF         DB      0          ;Access Right Byte
00000006                   10  GRAN            DB      0          ;Flags und Segment-Obergrenze
                           11                                     ;(Bits 16..19)
00000007                   12  BASIS_24_31     DB      0          ;Basisadresse (Bits 24..31)
--------                   13  DESK            ENDS
                           14
  0001                     15  PE              EQU     1          ;Protection Mode freigeben
                           16
----                       17  INITIALISIERUNG SEGMENT ER USE16   ;Ausfuehrbares und lesbares Segment
                           18
000                        19                  ORG     0          ;Die EPROM-GDT Adresse=FFFF0000H
                           20
0000 (8                    21  NULL_DESK       DB SIZE DESK DUP (?)  ;Luecke und Null Deskriptor
     ??
     )
0008 (8                    22  GDT_ALIAS_DESK DB SIZE DUP(?)     ;Alias-Deskriptor der GDT
     ??
     )
0010 (8                    23  IDT_ALIAS_DESK DB SIZE DESK DUP(?)  ;Alias-Deskriptor der IDT
     ??
     )
0018 (8                    24  TSS_DESK       DB SIZE DESK DUP (?)  ;Deskriptor des Task-Status-Segments
     ??
     )
                           25
0020 660F20C0              26                  MOV     EAX,CR0    ;EAX bekommt augenblicklichen
                           27                                     ;CR0-Status
     0024 660D01000000     28                  OR      EAX,PE     Setze PE-Bit
     002A 660F22C0         29                  MOV     CR0,EAX    ;Schalte in den Protected Mode
     002E EB00             30                  JMP     SHORT ENTFERNE ;Real Address Mode-Befehl aus der
     0030                  31  ENTFERNE                           ;Instruction Queue entfernen
0030 8D061800          R  32                  LEA     AX,CS:TSS_DESK ;AX enhaelt den Selektor des
                           33                                     ;TSS-Deskriptors
     0034 0F00D8           34                  LTR     AX         ;Deskriptor in das Tals-Register laden
                           35
     ----                  36  INITIALISIERUNGS ENDS
*** WARNING #377 IN 36, (PASS 2) SEGMENT CONTAINS PRIVILEGED INSTRUCTION(S)
                           37
                           38                  END
ASSEMBLY COMPLETE, 1 WARNING, NO ERRORS.
```

Es ist ersichtlich, daß der Task-Status-Segment-Deskriptor (TSS_DESK) in der EPROM-GDT enthalten ist.

Der Selektor dieses Deskriptors wird vom Befehl

- ***LEA AX, CS: TSS_DESK***

in das AX-Register geladen.

Anschließend benützt der Befehl

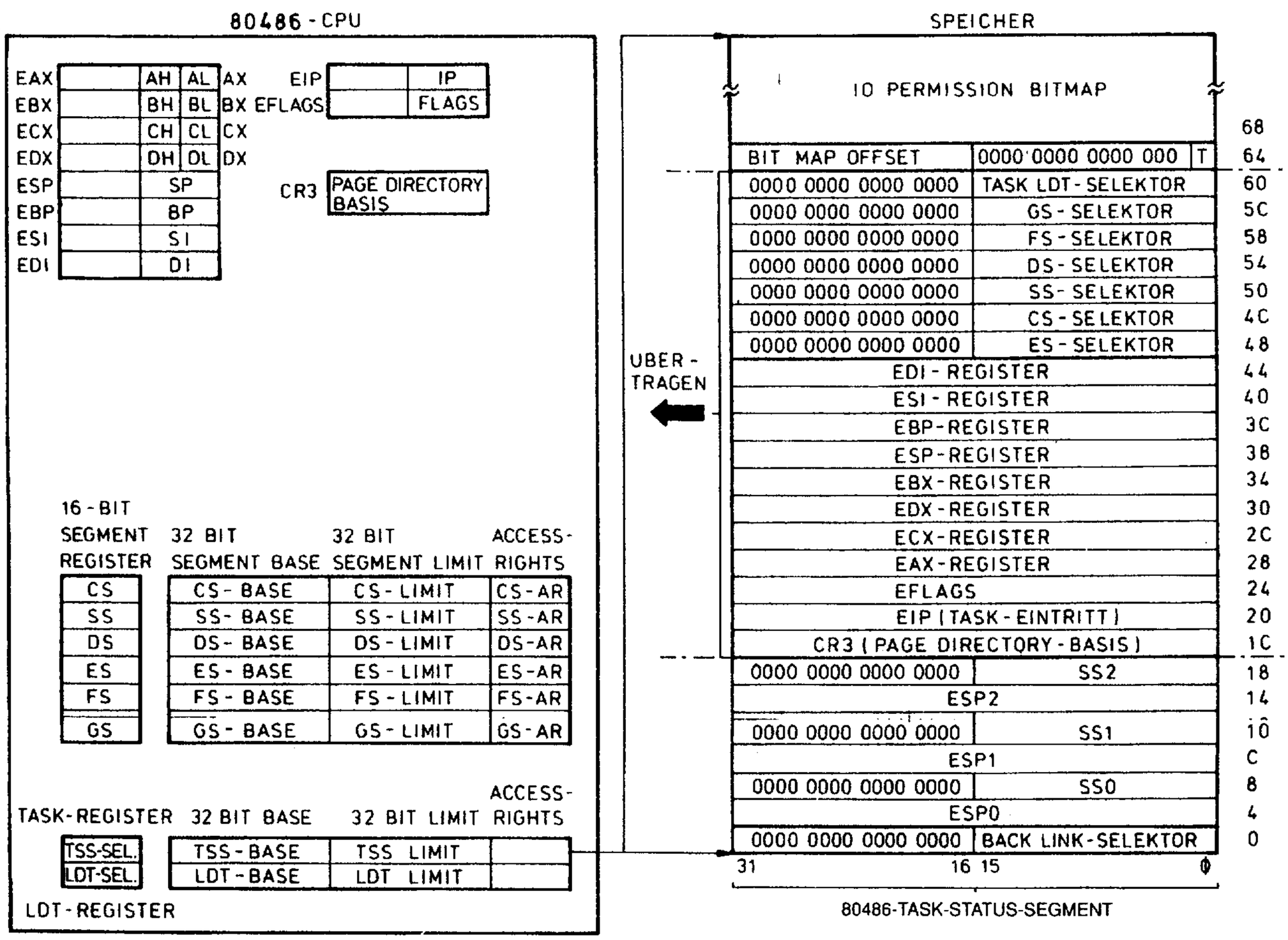
- ***LTR AX***

das AX-Register, um das Task-Status-Register der CPU mit dem TSS-Deskriptor zu initialisieren.

Da LTR nur im „Protected Mode" erkannt wird, mußte vor seiner Ausführung der Prozessor in diese Betriebsart umgeschaltet werden.

Nachdem das Task-Register initialisiert worden ist, werden die Eintragungen im ausgewählten Task-Status-Segment benützt, um die zugehörige Task zu aktivieren. Hierzu wird der augenblickliche Task-Status in die passenden CPU-Register übertragen.

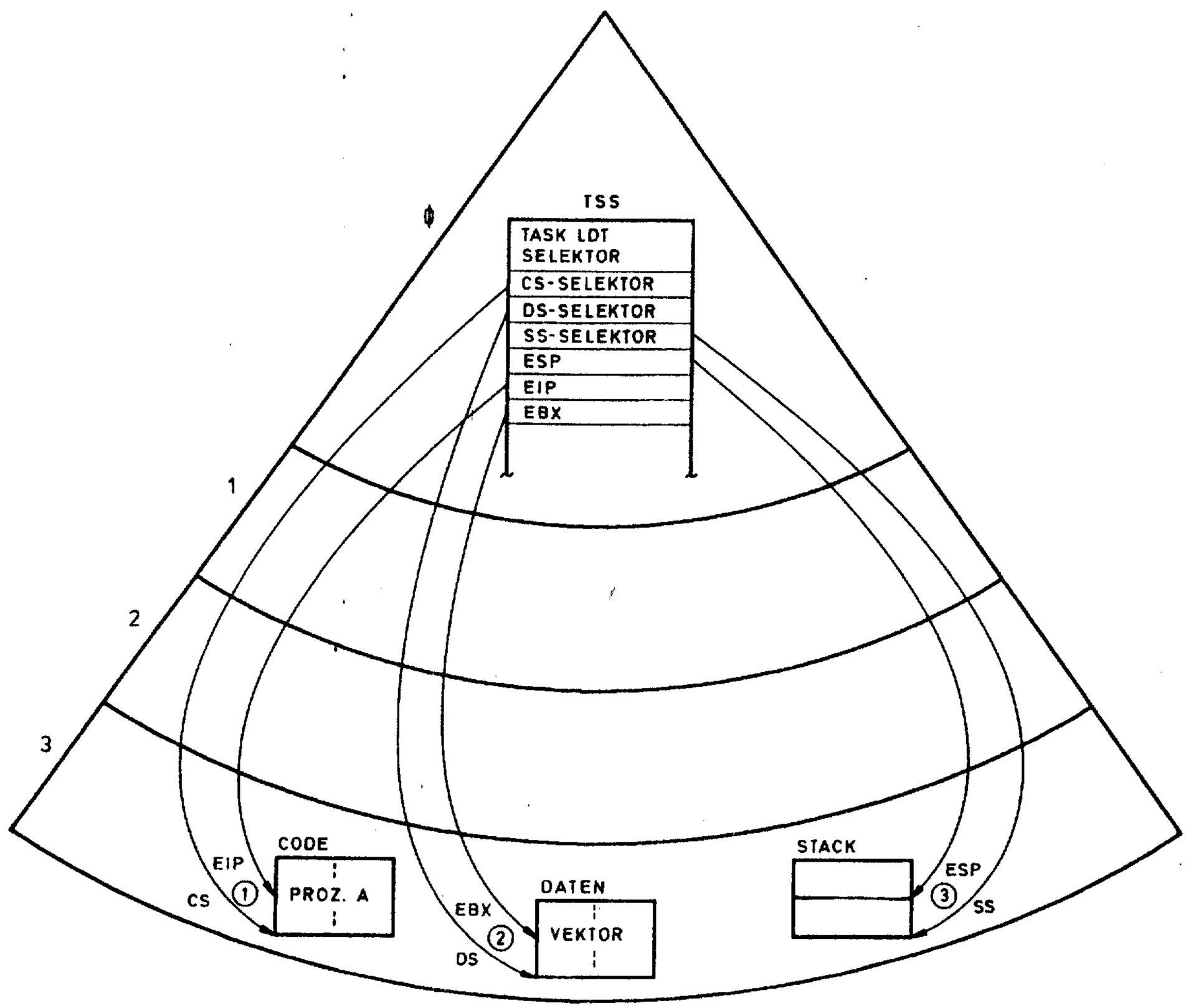
Das folgende Bild soll dies verdeutlichen:



Es ist ersichtlich, daß folgende Task-Status-Registerfelder in die CPU übertragen werden: TASK-LDT-Selektor, GS, FS, DS, SS, CS, ES, EDI, ESI, EBP, ESP, EBX, EDX, ECX, EAX, EFLAGS, EIP und CR3.

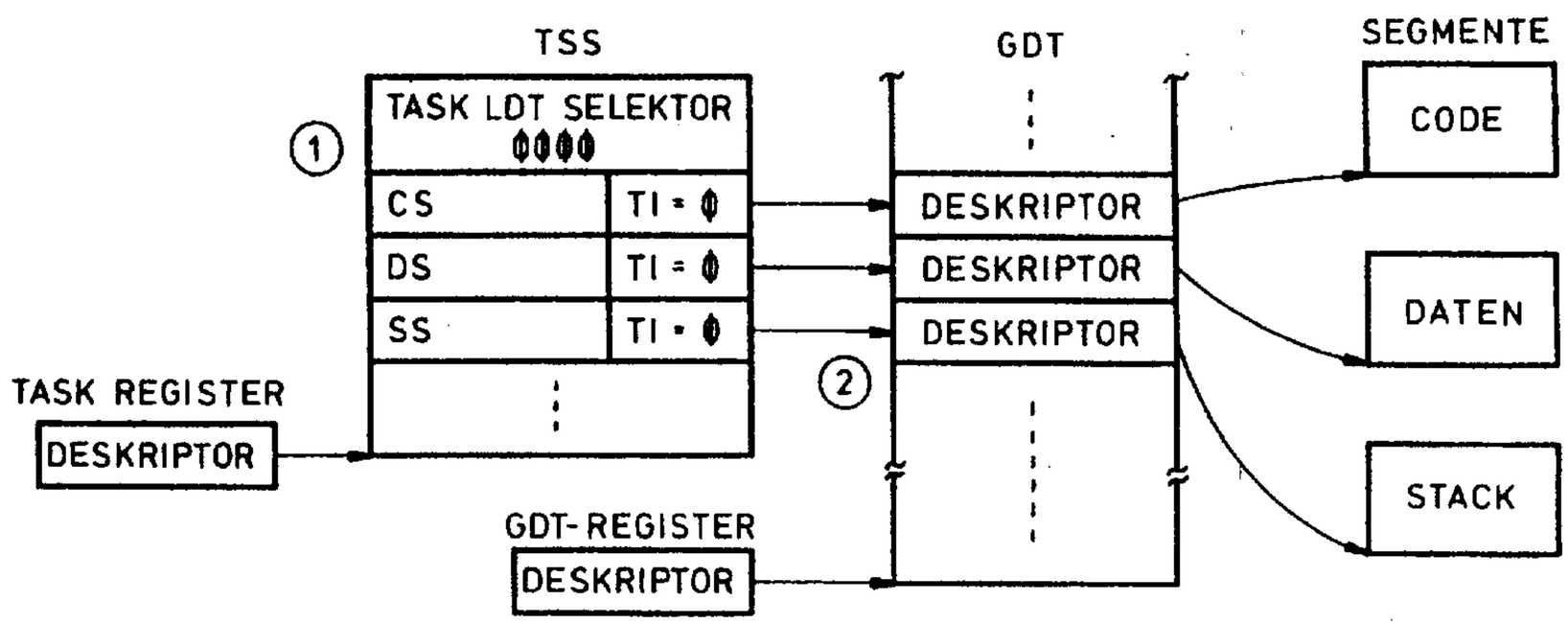
4.4 Starten einer Task

Ist der eben beschriebene Ladevorgang abgeschlossen, beginnt die CPU mit der Ausführung der Task. Wie das folgende Bild zeigt, benützt die CPU z. B. die TSS-Register-Felder CS, DS, SS, EBX, ESP und EIP, um eine Task in der Privileg-Ebene 3 auszuführen.

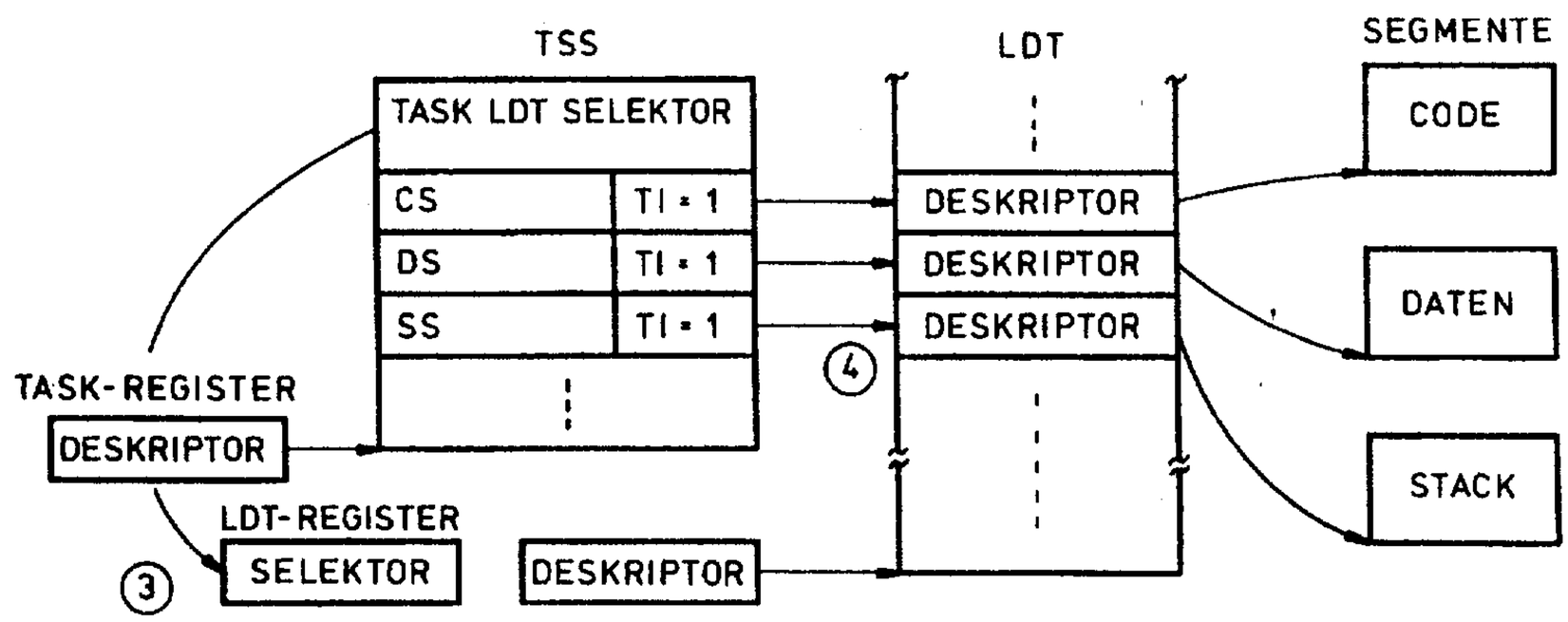


Dabei kennzeichnet CS:EIP die Anfangsposition der Prozedur A im Code-Segment ①, DS:EBX die Anfangsposition eines Vektors im Daten-Segment ② und SS:ESP die Anfangsposition des Stack-Segments ③.

Im vorigen Beispiel ist vorausgesetzt, daß die CPU den Task-LDT-Selektor im Task-Status-Segment ***nicht*** benützt, da dieses Feld einen Null-Sektor enthält ①. Dieser Null-Sektor markiert die LDT als „nicht vorhanden". Aus diesem Grund zeigen die Selektoren CS, DS und SS wegen TI = 0 zur GDT, um die entsprechenden Deskriptoren der Code-, Daten- und Stack-Segmente auszuwählen ②.



Benützt die CPU den Task-LDT-Selektor im Task-Status-Segment, bedeutet dies, daß das LDT-Register mit einem Deskriptor initialisiert wird, der die LDT der augenblicklichen Task beschreibt ③.
In diesem Fall zeigen die Selektoren CS, DS, und SS wegen TI = 1 zur LDT der aktiven Task, um die entsprechenden Deskriptoren der Code-, Daten- und Stack-Segmente auszuwählen ④.



4.5 Task-LDT-Selektor und -Deskriptor

Da das komplette Layout eines LDT-Deskriptors bisher noch nicht gezeigt worden ist, soll dies an dieser Stelle nachgeholt werden:

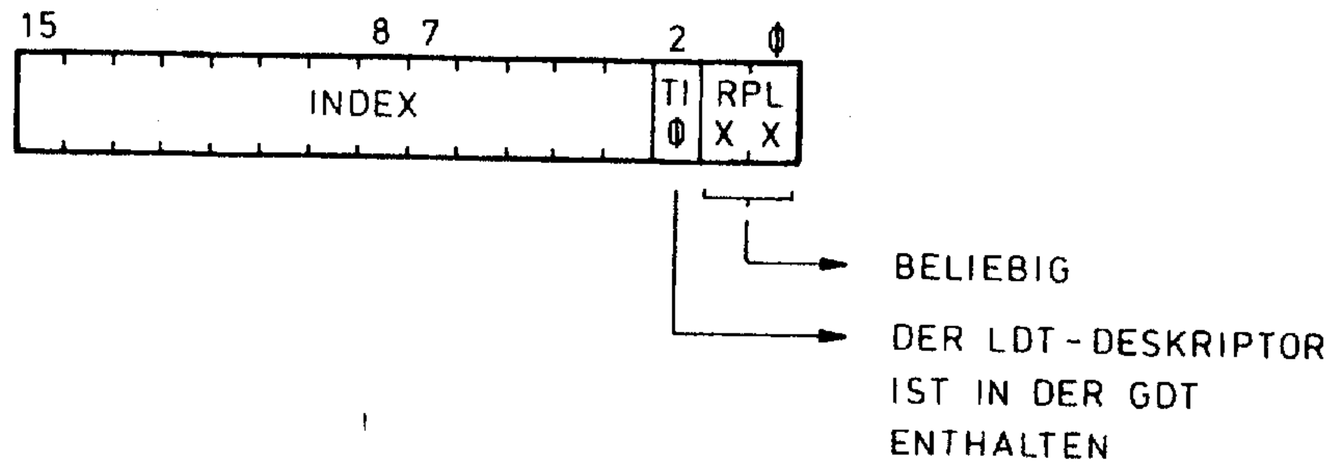
LDT - DESKRIPTOR

LDT-BASE 31...24					G	0	0	AVL	LIMIT 19...16
P	DPL X X		0	TYPE 0 0 1 0	LDT-BASE 23...16				
LDT-BASE 15...0									
LDT-LIMIT 15...0									

Der Deskriptor beschreibt ein System-Segment, wobei der Wert 2 im Type-Feld des Deskriptors angibt, daß es sich um eine lokale Deskriptor-Tabelle handelt.

Das DPL-Feld wird ***nicht*** benützt und kann daher mit einem beliebigen Wert initialisiert werden. Der Grund hierfür liegt in der Tatsache, daß sich die LDT genauso wie die GDT

oder die IDT als speicher-residente Tabelle auf ***keiner*** Privileg-Ebene aufhält und daher keine Privileg-Kennummer hat.
Da beim Zugriff auf einen LDT-Deskriptor keine Privileg-Überprüfung durchgeführt wird, kann auch das RPL-Feld eines Task-LDT-Selektors einen beliebigen Wert haben.



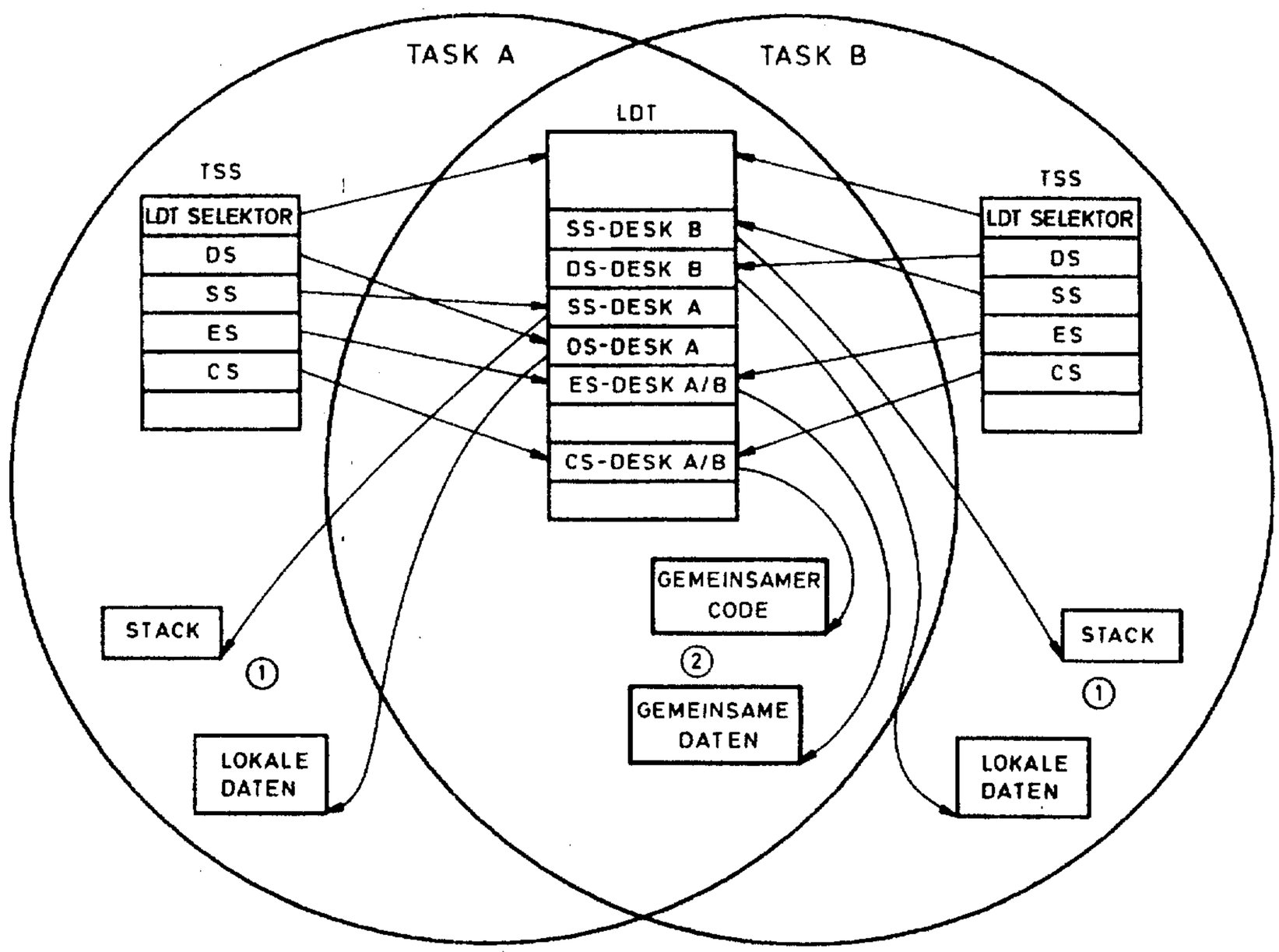
HINWEIS: Eine Privileg-Überprüfung wird dann durchgeführt, wenn die speicherresidenten Tabellen zu Manipulationszwecken als Alias-Segmente (les- und schreibbare Daten-Segmente) definiert sind.

Das P-Bit (Present) gibt an, ob der LDT-Deskriptor gültige Informationen enthält. Dabei bedeutet P = 1 ja und P = 0 nein.

4.6 Gemeinsamer linearer Adreßraum für mehrere Tasks

1. Segment-Arrangement über eine gemeiname LDT

Wollen zwei oder mehrere Tasks die gleichen Segmente erreichen, dann ist dies realisierbar, wenn sie die ***gleiche*** LDT benützen. Das folgende Bild illustriert ein solches Arrangement:



Es existieren die beiden Tasks A und B, die durch individuelle Task-Status-Segmente (TSS) definiert sind.
Beide Status-Segmente enthalten im LDT-Feld identische Selektoren, so daß beide Tasks den gleichen LDT-Deskriptor in der GDT benützen.

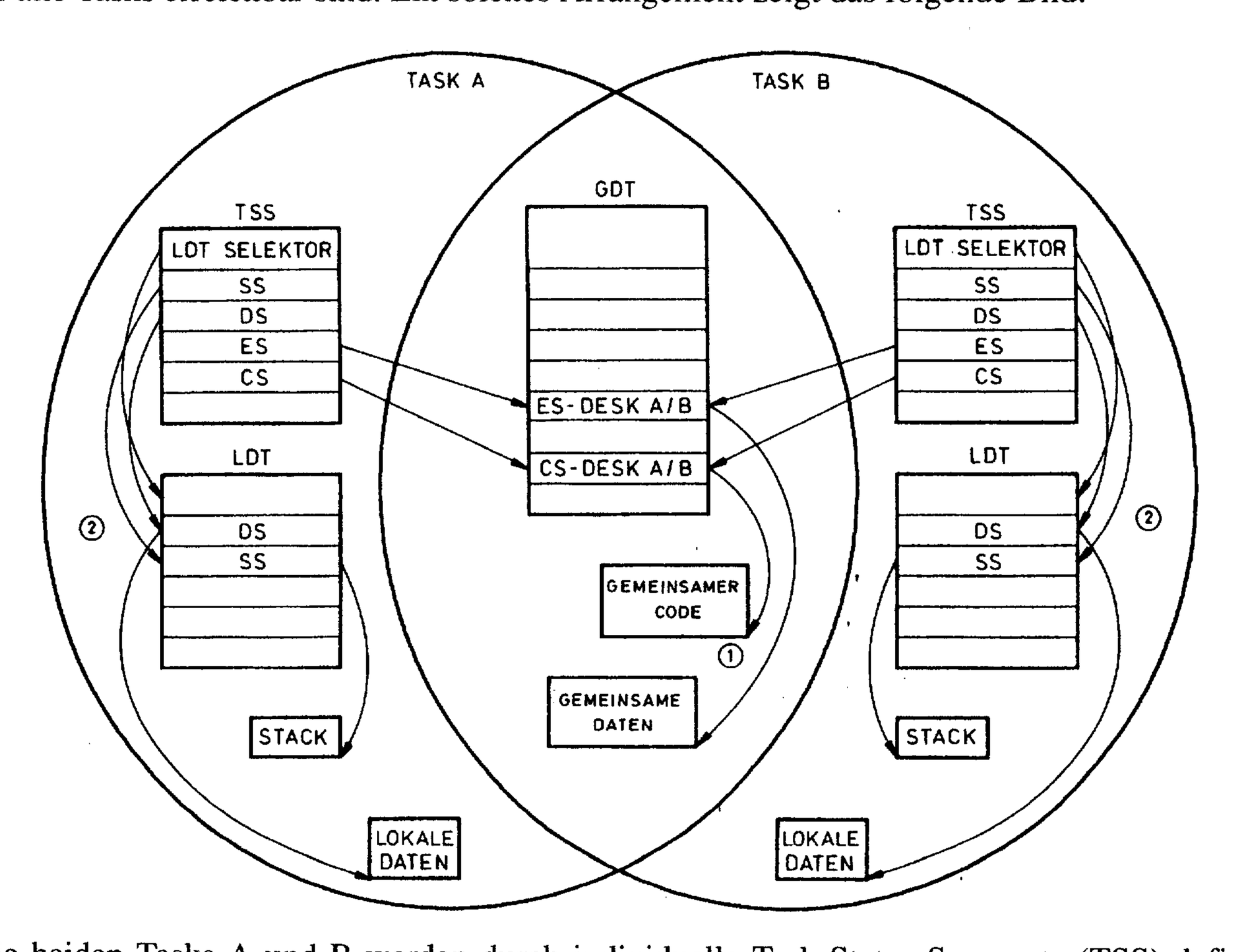
Im Beispiel enthält die gemeinsame LDT Segment-Deskriptoren, die von den SS- und DS-Selektoren der beiden Task-Status-Segmente ausgewählt werden und die ***lokalen*** Daten- und Stacksegmente beschreiben ①.

Zusätzlich sind in der LDT Deskriptoren enthalten, die von beiden Tasks ***gemeinsam*** benützt werden. So ist es möglich, daß gemeinsame Code- und Daten-Segmente über identische CS- und ES-Selektoren erreicht werden ②.

Die gemeinsame Benützung einer LDT setzt voraus, daß die Tasks kooperationsfähig sind, da das Risiko besteht, daß im Fehlerfall die eine Task die andere ungünstig beeinflußt.

2. Segment-Arrangement über die GDT

Das natürlichste Mittel zur Definition gemeinsamer Segmente für mehrere Tasks ist die GDT. Denn es ist ein Leistungsmerkmal der GDT, daß die in ihr enthaltenen Deskriptoren für alle Tasks erreichbar sind. Ein solches Arrangement zeigt das folgende Bild:



Die beiden Tasks A und B werden durch individuelle Task-Status-Segmente (TSS) definiert. Dabei sind die Adreßräume der beiden Tasks in einen durch die GDT beschriebenen ***globalen*** Bereich und in zwei durch private LDT's beschriebene ***lokale*** Bereich unterteilt.

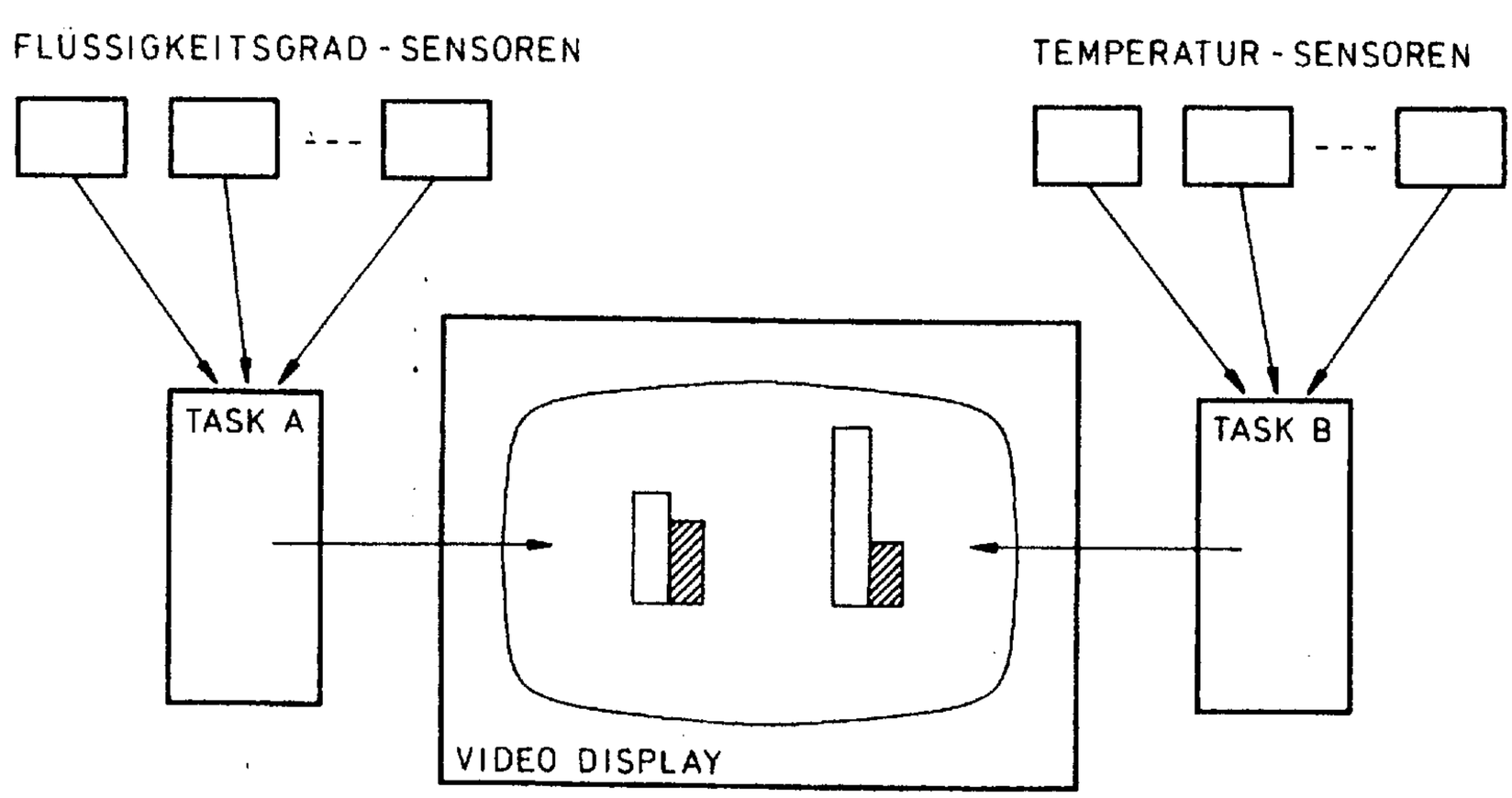
Die GDT enthält Segment-Deskriptoren, die z. B. von den ES- und CS-Selektoren der beiden Task-Status-Segmente ausgewählt werden können. Sie gibt damit den Tasks die Möglichkeit, ***gemeinsame*** Segmente zu benützen ①.

Beide Status-Segmente enthalten im LDT-Feld unterschiedliche Selektoren, die in der GDT die privaten LDT-Deskriptoren der beiden Tasks indizieren. Die von ihnen beschriebenen LDT's enthalten die Deskriptoren der ***lokalen*** Daten- und Stack-Segmente. Im Beispiel werden diese Deskriptoren von den DS- und SS-Selektoren in den beiden Task-Status-Segmenten ausgewählt ②.

3. Segment-Arrangement über Alias-Deskriptoren in unterschiedlichen LDT's

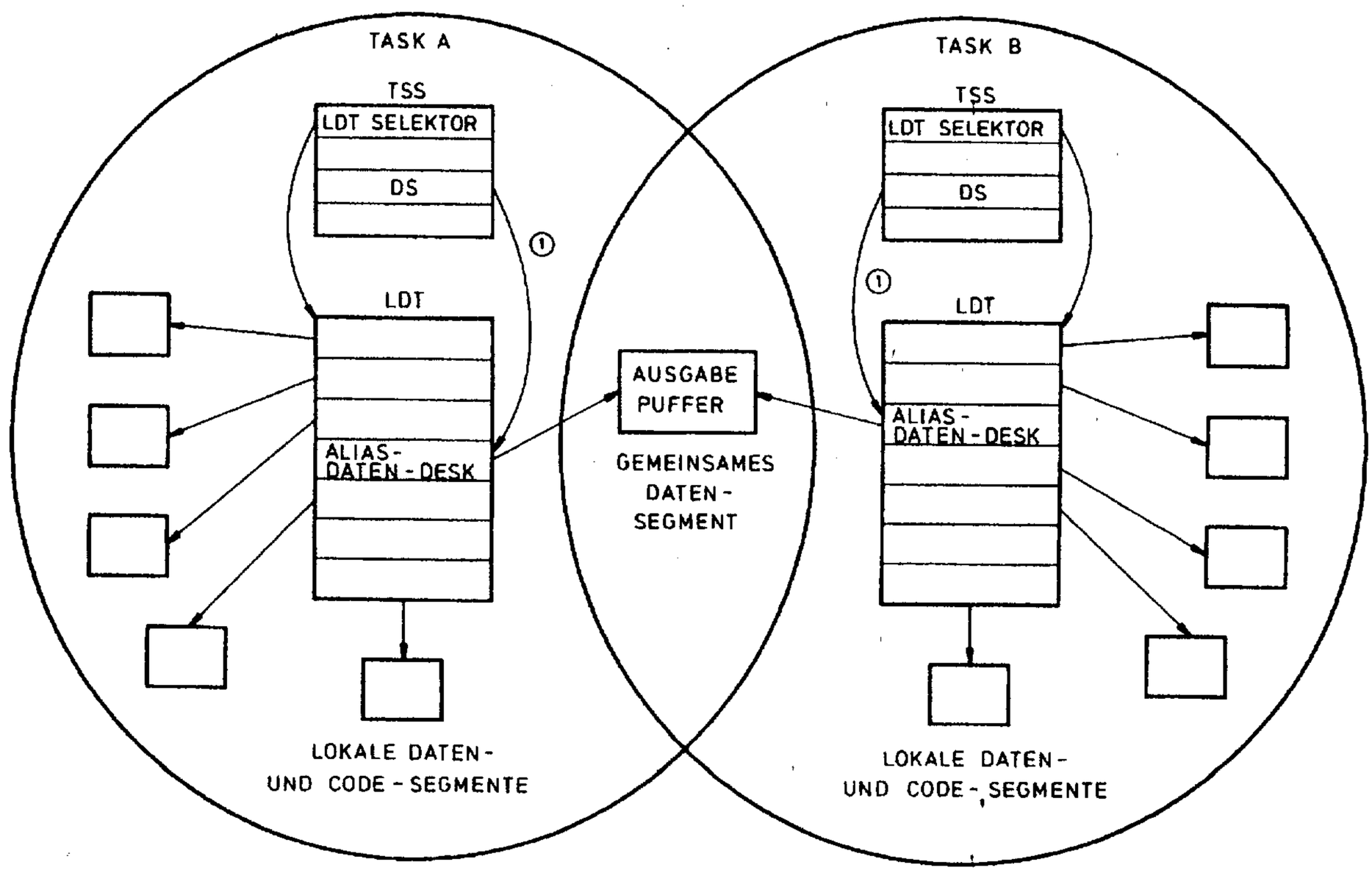
Welchen Nutzen Alias-Deskriptoren bei der Definition gemeinsamer Segmente für mehrere Task haben, soll die folgende Anwendung verdeutlichen:

> Innerhalb eines Produktions-Prozesses existiert ein Video-Display, an das ständig Status-Informationen über die Temperatur und den Flüssigkeitsgrad eines bestimmten Produktionsmittels ausgegeben werden. In dieser Anwendung gibt es zwei Tasks, die an unterschiedlichen Aspekten des Prozesses interessiert sind.



Während Task A den Flüssigkeitsgrad verfolgt, mißt Task B die Tempertur. Beide Tasks benützen aber das Video-Display, um ihre Meßdaten anzuzeigen. Aus diesem Grund ist ein gemeinsamer Daten-Ausgabe-Puffer definiert, der die Meßdaten der beiden Tasks als Zwischenspeicher aufnimmt.

Wie das Segment-Arrangement für diese Anwendung aussehen könnte, zeigt das folgende Bild:

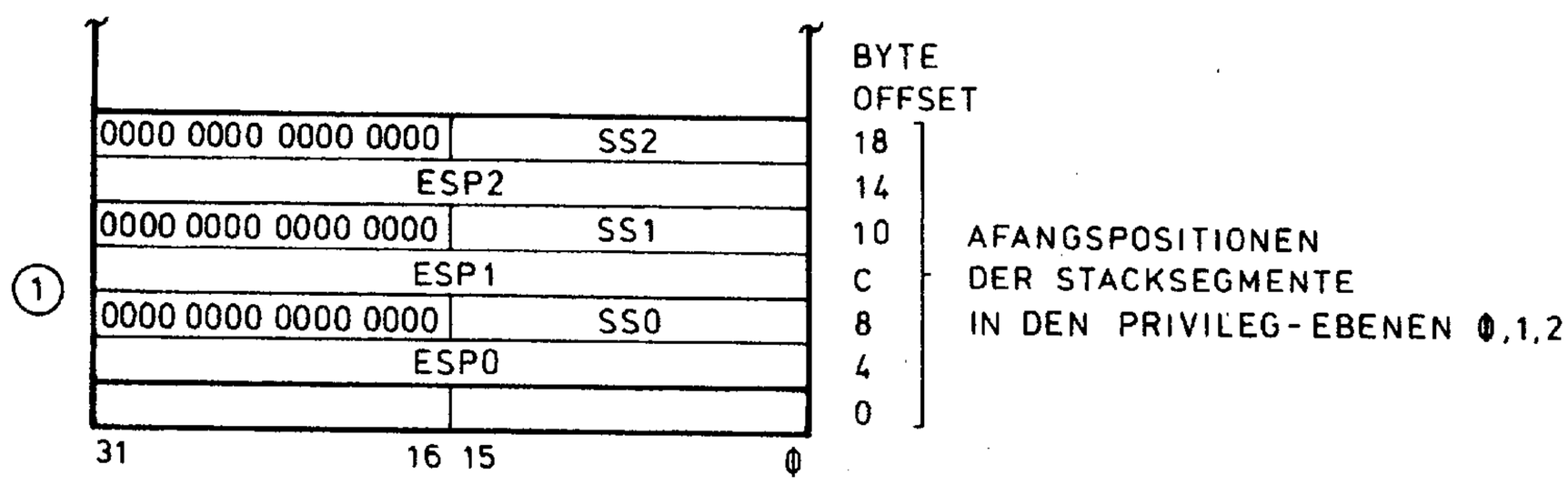


Die beiden Tasks A und B werden durch individuelle Task-Status-Segmente (TSS) definiert. Beide Status-Segmente enthalten im LDT-Feld unterschiedliche Selektoren, die in der GDT die privaten LDT-Deskriptoren der beiden Tasks auswählen. Die von ihnen beschriebenen LDT's enthalten die Deskriptoren der ***lokalen*** Daten- und Code-Segmente.

Zusätzlich ist in jeder LDT ein Deskriptor aufbewahrt, der als Alias-Datensegment-Deskriptor das gemeinsame Datensegment beider Tasks beschreibt. Dieses Datensegment enthält den Ausgabe-Puffer für die Meßdaten. Immer dann, wenn eine der Tasks ihre Meßdaten aktualisieren will, muß sie das gemeinsame Datensegment benützen. Sie verwendet dabei z. B. den DS-Selektor im Task-Status-Segment und indiziert damit den Alias-Datensegment-Deskriptor in ihrer LDT ①.

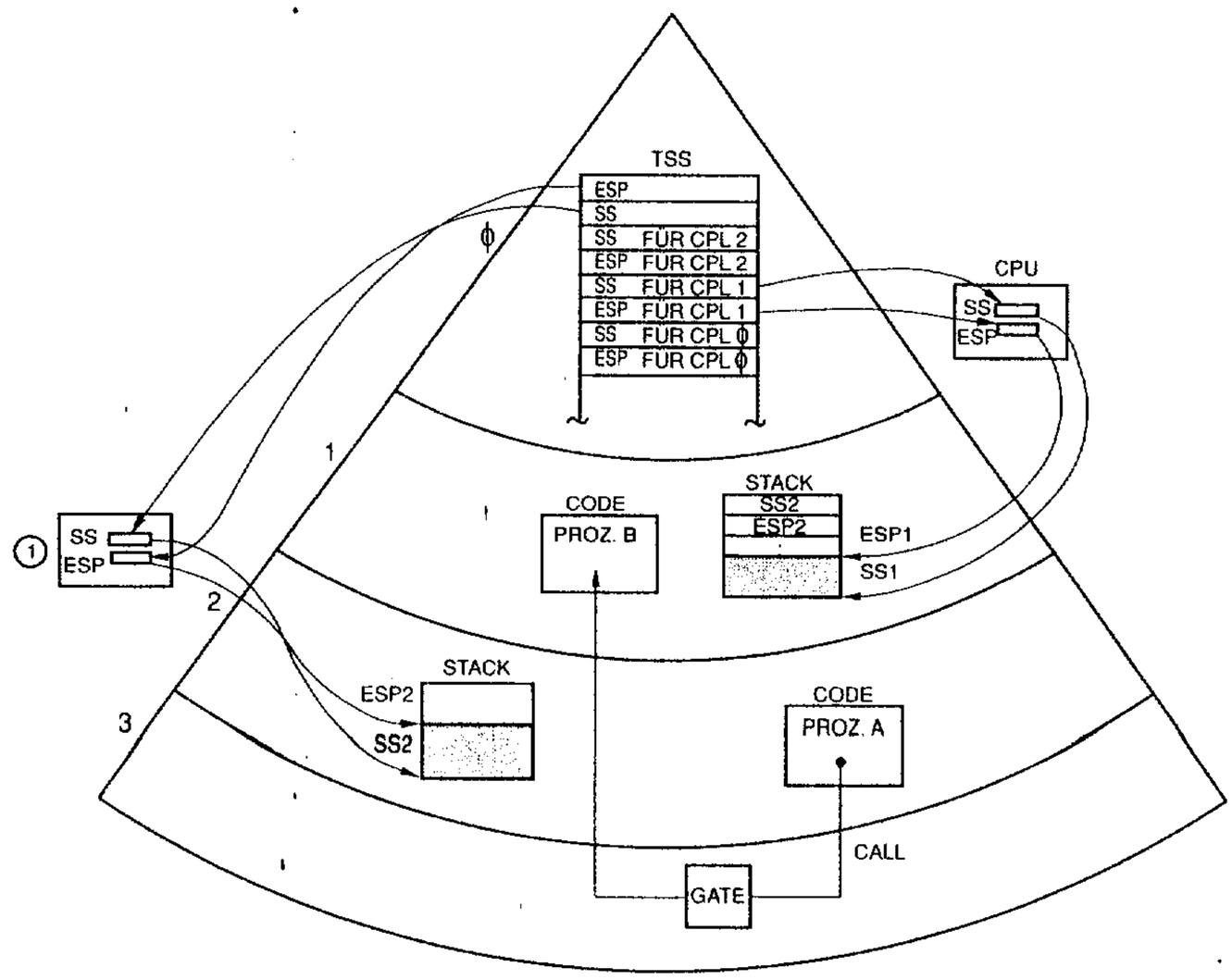
4.7 Privileg-Wechsel und Stack-Segment-Anfangspositionen

Für jede der Privileg-Ebenen 0, 1, 2 einer Task kann der „System Builder BLD386/486" die Anfangsposition eines Stack-Segments spezifizieren. Diese Anfangspositionen sind als statische Eintragungen im Task-Status-Segment jeder Task gespeichert ①.



Die Anfangspositionen der Stack-Segmente sind SS:ESP-Zeiger, die immer dann benützt werden, wenn durch einen Call-Befehl ein Übergang in eine höhere Privileg-Ebene (2, 1 oder 0) erfolgt.

Ist ein solcher Privileg-Wechsel realisiert, wird der zur neuen Privileg-Ebene gehörende SS:ESP-Zeiger dem Task-Status-Segment entnommen, um die Anfangsposition des neuen Stack-Segments zu kennzeichnen. Dies soll das folgende Bild verdeutlichen:

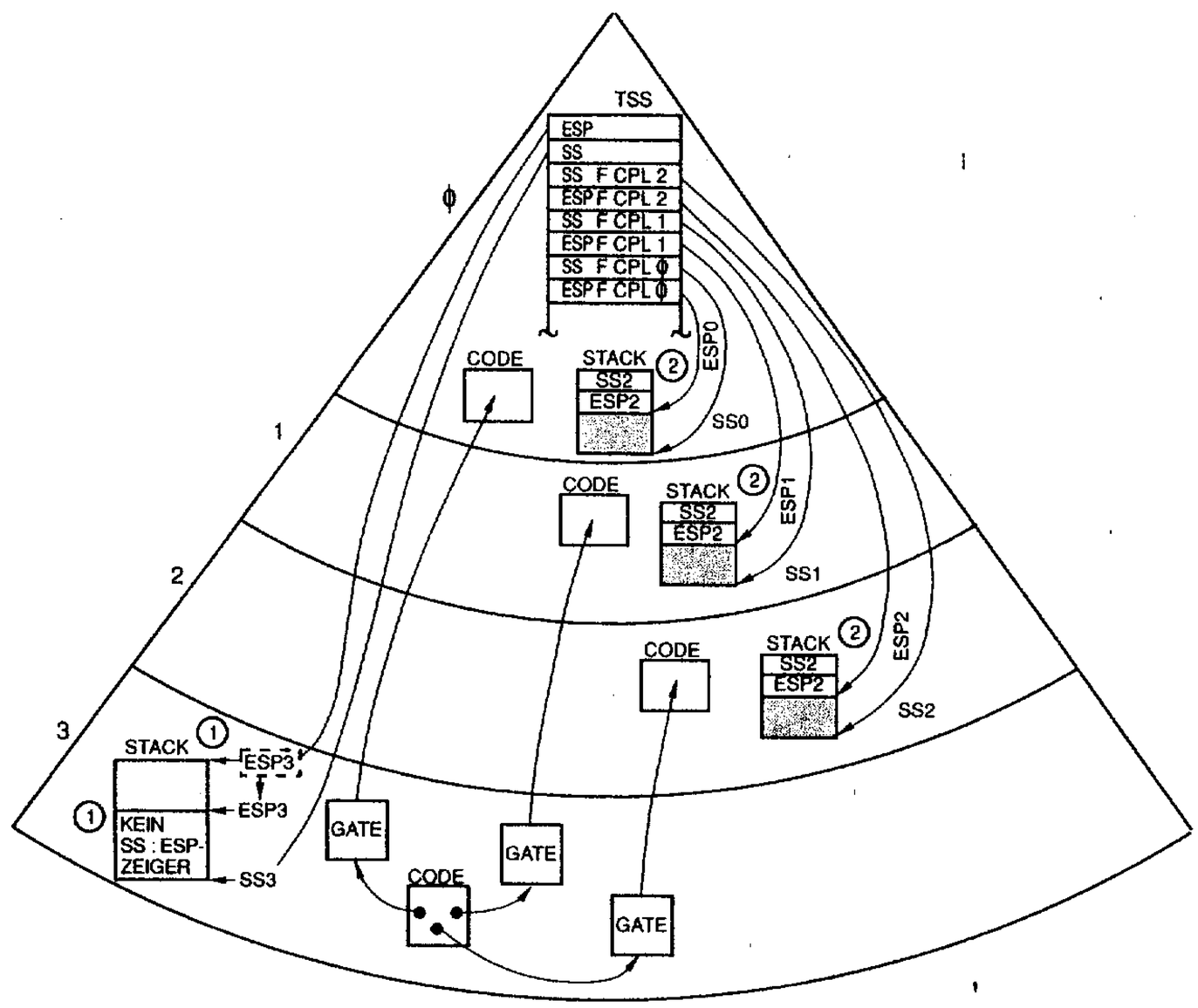


Es ist zu erkennen, daß die Prozedur A in der Privileg-Ebene 2 über „Call Gate" die Prozedur B in der Privileg-Ebene 1 aufruft. Dabei ist die Anfangsposition des Stack-Segments in Ebene 2 durch die SS-ESP-Felder des Task-Status-Segments festgelegt ①. Nach dem Privileg-Wechsel überschreiben die Task-Felder SS und ESP für CPL 1 die CPU-Register, so daß die Angangsposition des Stack-Segments in der Ebene 1 spezifiziert ist.

Die zur aufrufenden Prozedur A gehörende Stack-Position SS2:ESP2 wird im Stack-Segment der neuen Privileg-Ebene oberhalb der Rücksprung-Adresse gespeichert, wodurch eine Verbindung zum weniger privilegierten Stack in der Ebene 2 hergestellt ist.

Wie bereits im Kapitel „Schutz" erläutert worden ist, kann ein Privileg-Wechsel nur in Richtung privilegierterer Ebenen erfolgen. Durch diese Einschränkung ist erklärbar, warum ein Task-Status-Segment nur SS:ESP-Zeiger für die Privileg-Ebene 0, 1 und 2 enthält. Führt nämlich die Task ein Code-Segment in der Privileg-Ebene 3 aus, können Privileg-Wechsel in die Ebenen 0, 1 oder 2 nur deswegen realisiert werden, weil die Anfangspositionen der neuen Stack-Segmente in den SS-ESP-Feldern für CPL 0, CPL 1 und CPL 2 des Task-Status-Segments spezifiziert sind. Das folgende Bild illustriert diesen Fall:

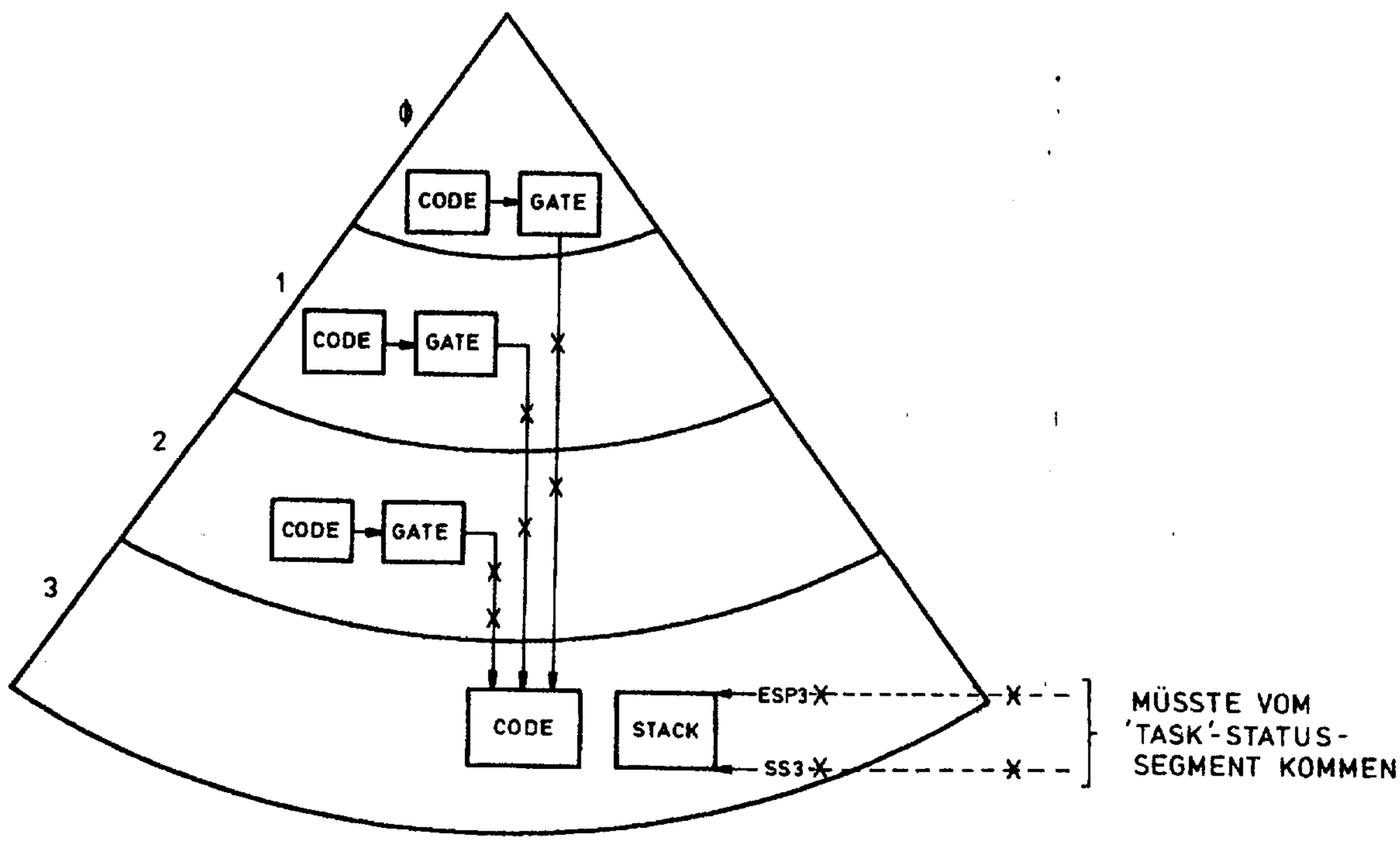
Es ist ersichtlich, daß sich die Anfangsposition des Stacks in der Privilegebene 3 im dynamischen SS-ESP-Feld des Task-Status-Segments befindet ①, während die augenblickliche Position des Ebene-3-Stacks in den privilegierteren Stacks in Ebene 2, 1 und 0 gespeichert ist ②.



Dies läßt auch erkennen, daß der Ebene-3-Stack ***niemals*** Verbindungen in Form von SS:ESP-Zeigern zu anderen Stacks, die ja privilegierter sind, haben kann ③.

Das Beispiel zeigt, daß ein statisches SS-ESP-Feld für CPL 3 im Task-Status-Segment ***nicht*** erforderlich ist. Ein solches Feld ist nur für CPL 0, 1 oder 2 notwendig, um bei Privileg-Wechsel die Anfangspositionen höher privilegierter Stacks festzulegen.

Ein SS-ESP-Feld für CPL 3 hätte erst dann eine Berechtigung, wenn Privileg-Wechsel in Richtung weniger privilegierter Ebenen erlaubt wären. Wie das folgende Bild zeigt, müßte beim Übergang in die Ebene 3 die Anfangsposition des Stacks in dieser Ebene durch einen SS3:ESP3-Zeiger im Task-Status-Segment spezifiziert werden.



Da sich in weniger privilegierten Ebenen Code-Segmente mit geringerem Vertrauensgrad aufhalten, sind solche Übergänge ***nicht*** erlaubt. Daher hat ein statisches SS-ESP-Feld für CPL 3 im Task-Status-Segment keine Berechtigung. Sollte aber eine Task in der Privileg-Ebene 3 starten, muß selbstverständlich ein Zeiger existieren, der die Anfangsposition des Ebene-3-Stack festlegt. Als Lieferant eines solchen Zeigers eignet sich in diesem Fall das dynamische Feld SS:ESP im TSS.

4.8 Task-Wechsel

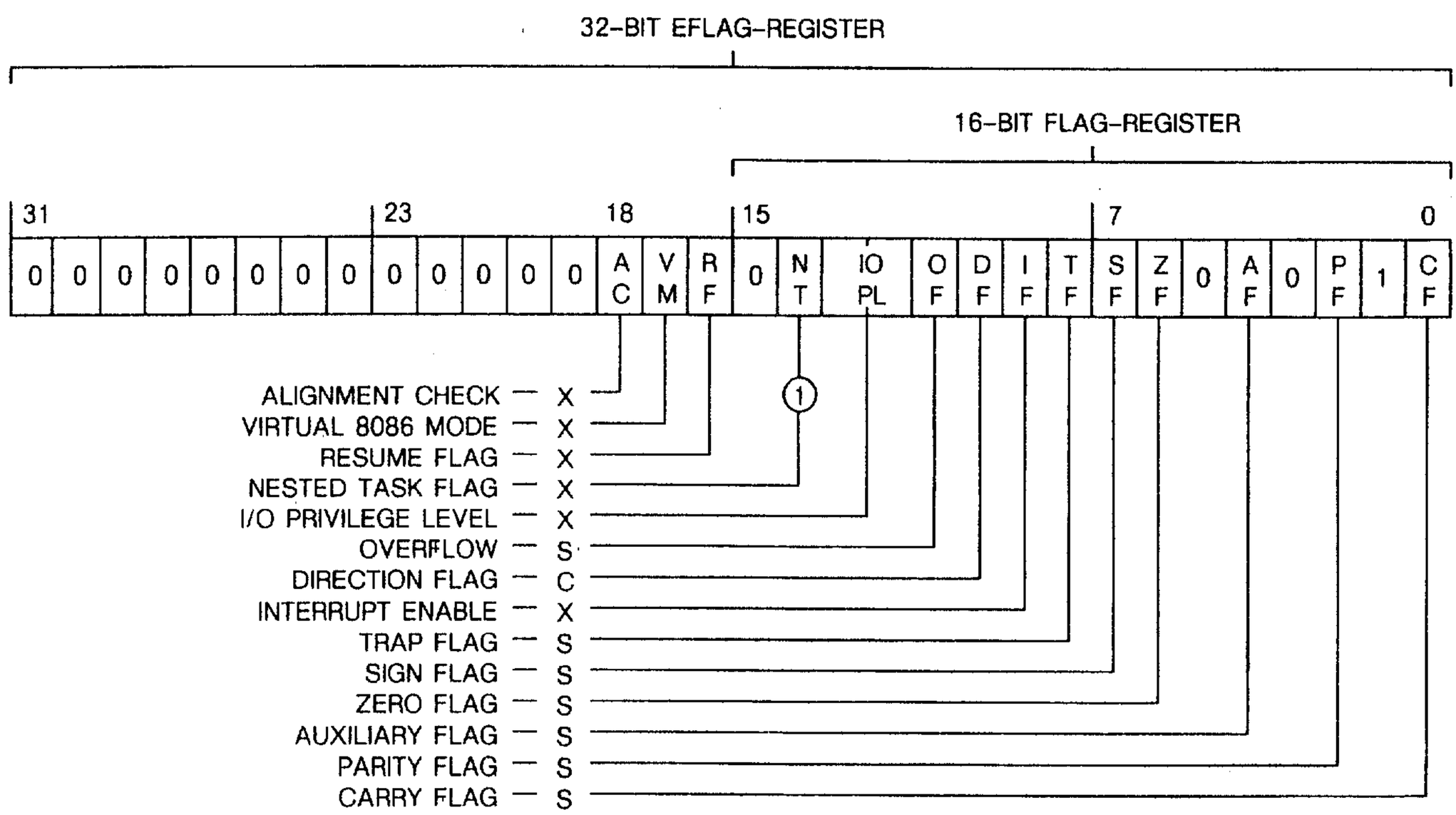
Da ein Multitasking-System typischerweise mehrere Tasks hat, müssen einige Vorbereitungen getroffen werden, damit der Prozessor die Ausführung der einen Task einstellen und die Ausführung einer anderen Task beginnen kann.

Der 80486 hat mehrere solcher Mechanismen, wobei, abhängig von den unterschiedlichen Situationen, der jeweils passende auszuwählen ist. Alle diese Mechanismen benützen die Standard-Befehle ***JMP, CALL, INT, IRET*** des 80486.

Unterschieden werden diese ***Standardbefehle*** von ***Task-Wechsel-Operationen*** entweder

- durch den Typ des Deskriptors, auf den sich der Befehl bezieht, oder
- durch das NT-Bit (Nested Task Flag) im EFLAG-Register

Die Position des NT-Bits im EFLAG-Register ist aus der folgenden Darstellung ersichtlich ①; seine Bedeutung wird im Laufe dieses Kapitels noch erläutert.



S=STATUS FLAG, C=CONTROL FLAG, X=SYSTEM FLAG

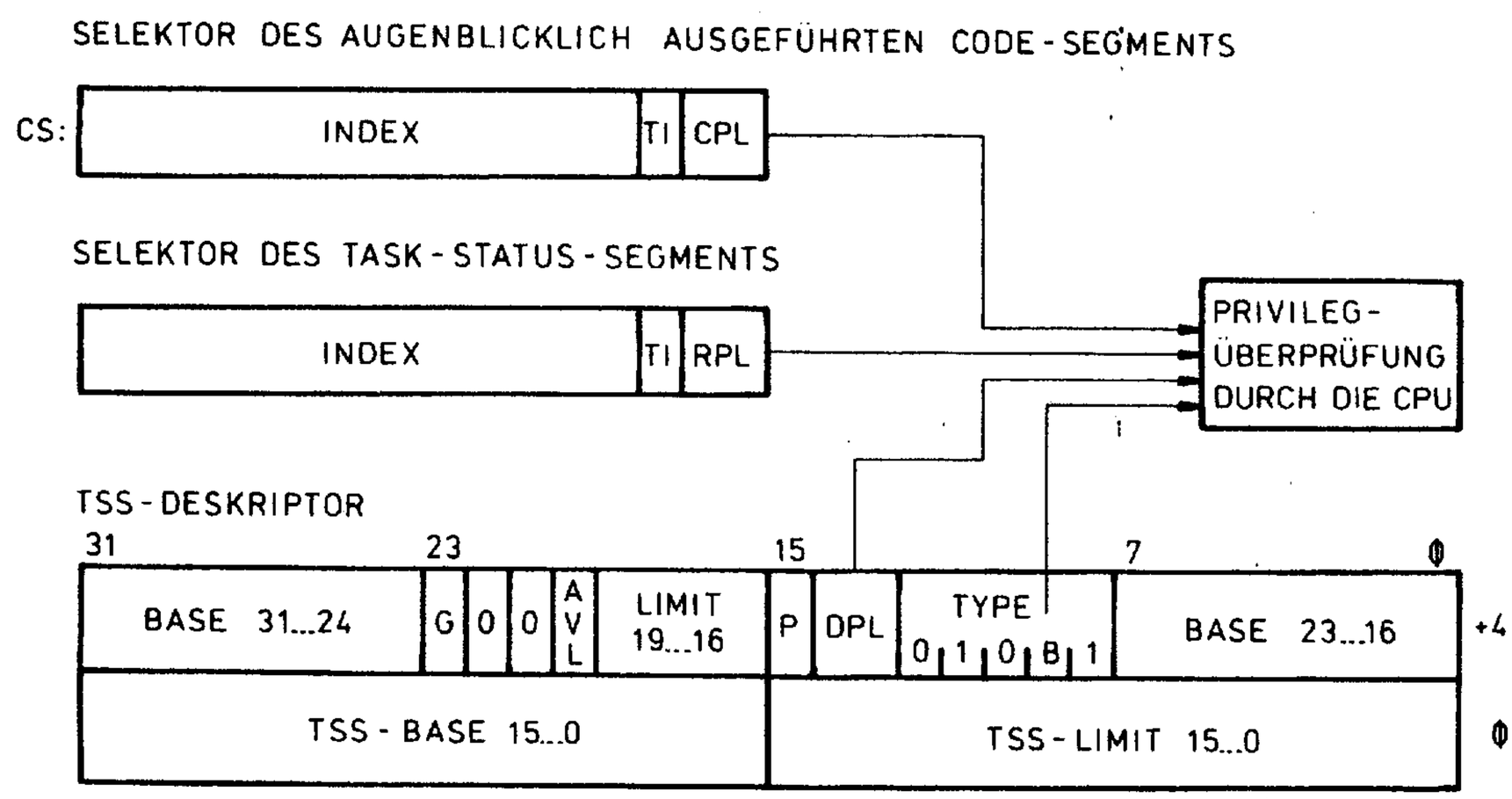
An dieser Stelle sollen zunächst die Befehle

- ***JMP SELEKTOR:OFFSET***
- ***CALL SELEKTOR:OFFSET***

benützt werden, um den Mechanismus eines Task-Wechsels zu zeigen.

Die Selektor-Komponente dieser Befehle können z. B. den Deskriptor eines ***Task-Status-Segments*** auswählen. Dabei müssen die Offset-Komponenten zwar präsent sein, sie werden aber ignoriert.

Bei einem Task-Wechsel über einen TSS-Deskriptor benutzt der Prozessor ***vier*** verschiedene Felder, um die Gültigkeit des beabsichtigten Intertask-Programmablaufs zu überprüfen.



Wie das angegebene Bild zeigt, sind dies:

- die augenblickliche Privilegebene (CPL) des augenblicklich ausgeführten Code-Segments.
- die Privilegebene (RPL) des vom CALL- oder JMP-Befehl gelieferten Selektors.
- die Privilegebene (DPL) im TSS-Deskriptor und
- das Busy-Bit (B) im Type-Feld des TSS-Deskriptors.

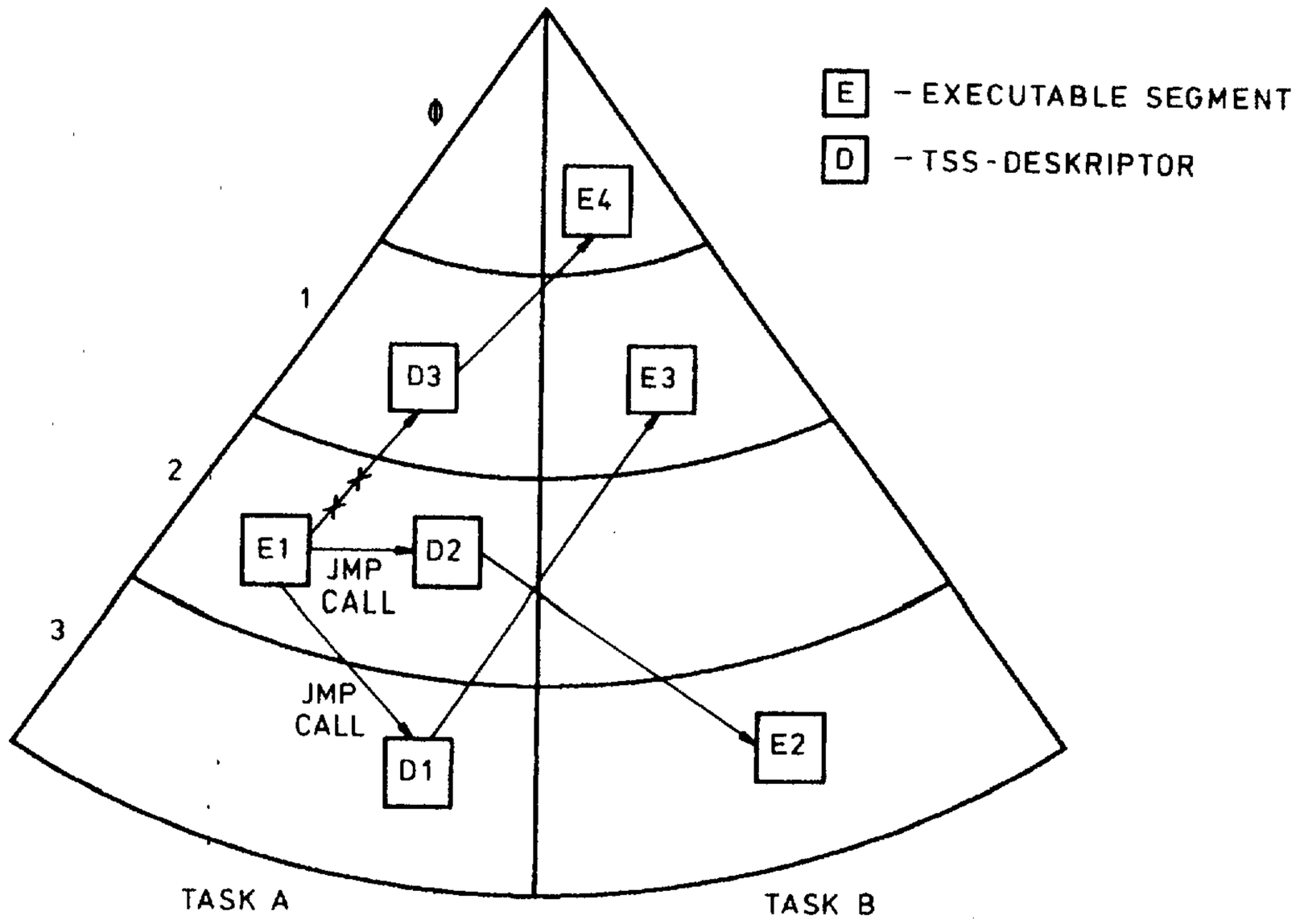
Dabei muß von JMP oder CALL die folgende Beziehung zwischen den Kennziffern der Privileg-Ebenen eingehalten werden:

$$\boxed{\text{MAX („Task“ CPL, Selektor RPL)} \leqslant \text{TSS-Deskriptor DPL}}$$

Ein Task-Wechsel ist also nur dann realisierbar, wenn die effektive Privileg-Ebene des JMP- oder CALL-Selektors numerisch ***kleiner*** oder ***gleich*** der Privileg-Ebene DPL des Task-Status-Segments ist. Zusätzlich muß das Busy-Bit im TSS-Deskriptor ***gelöscht*** sein (B = 0), da eine aktive Task (B = 1) ***nicht*** aufgerufen werden kann. Um ***nicht-autorisierten Prozeduren*** außerhalb des Betriebssystems die Möglichkeit zu nehmen, einen Task-Wechsel zu starten, sollte der DPL-Wert des TSS-Deskriptors 0 sein.

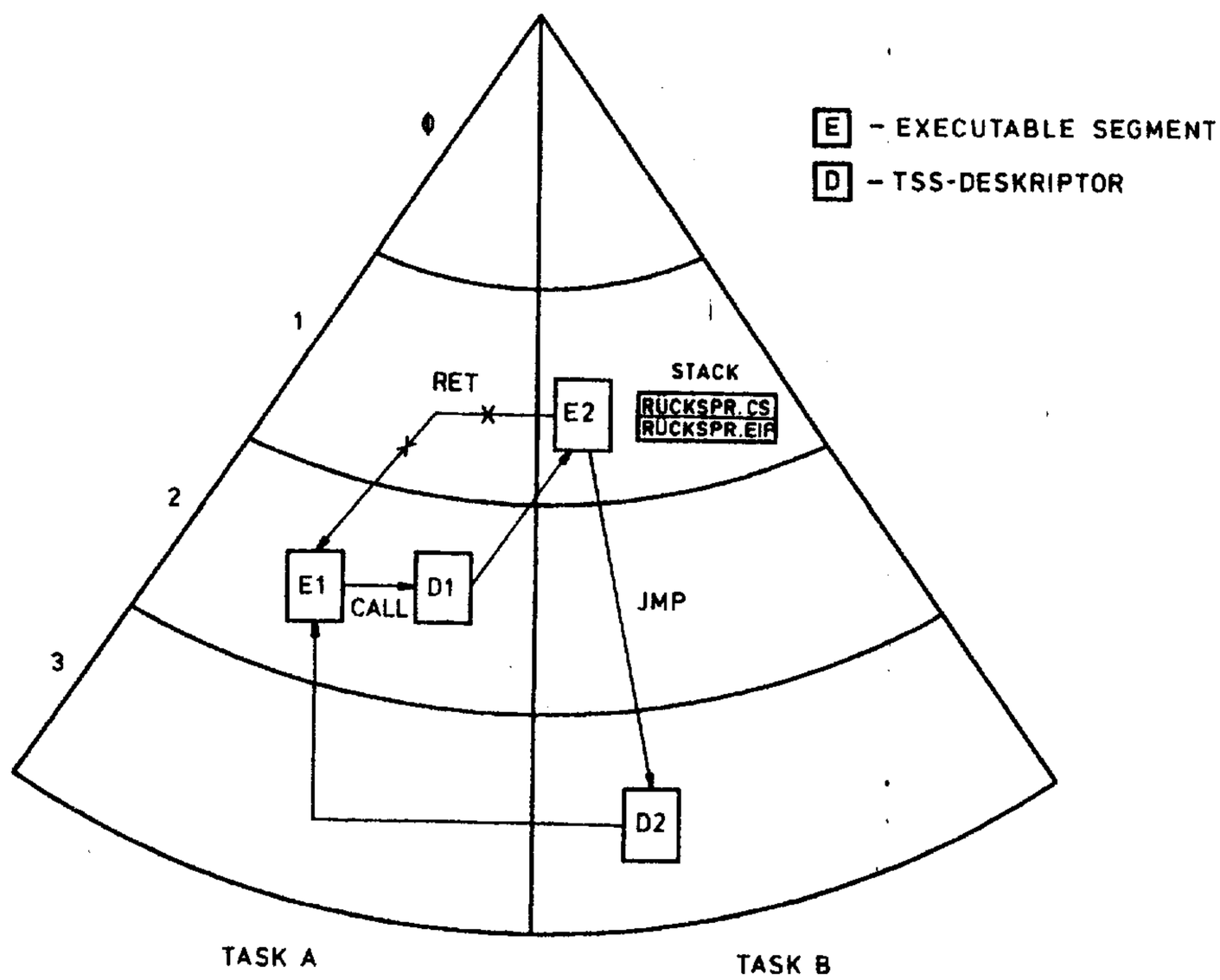
Es sei darauf hingewiesen, daß die Privileg-Ebene der aufzurufenden Task durch die augenblickliche Privileg-Ebene CPL der ausscheidenden Task ***nicht*** eingeschränkt wird. D. h. die aufzurufende Task kann sich auf jeder beliebigen Privileg-Ebene befinden. Die Begründung hierfür liegt in der Tatsache, daß Tasks gegeneinander durch separate Adreßräume und unterschiedliche Maschinenzustände isoliert sind.

Das folgende Bild illustriert gültige und ungültige Versuche, einen Wechsel von Task A nach B zu realisieren:



Es ist erkennbar, daß die Pfade ***E1-D2-E2*** und ***E1-D1-E3*** gültig sind, da hier jeweils die Bedingung MAX („Task“ CPL, Selektor RPL) ≤ TSS-Deskriptor DPL erfüllt ist. Im Gegensatz dazu ist der Pfad ***E1-D3-E4*** ungültig.

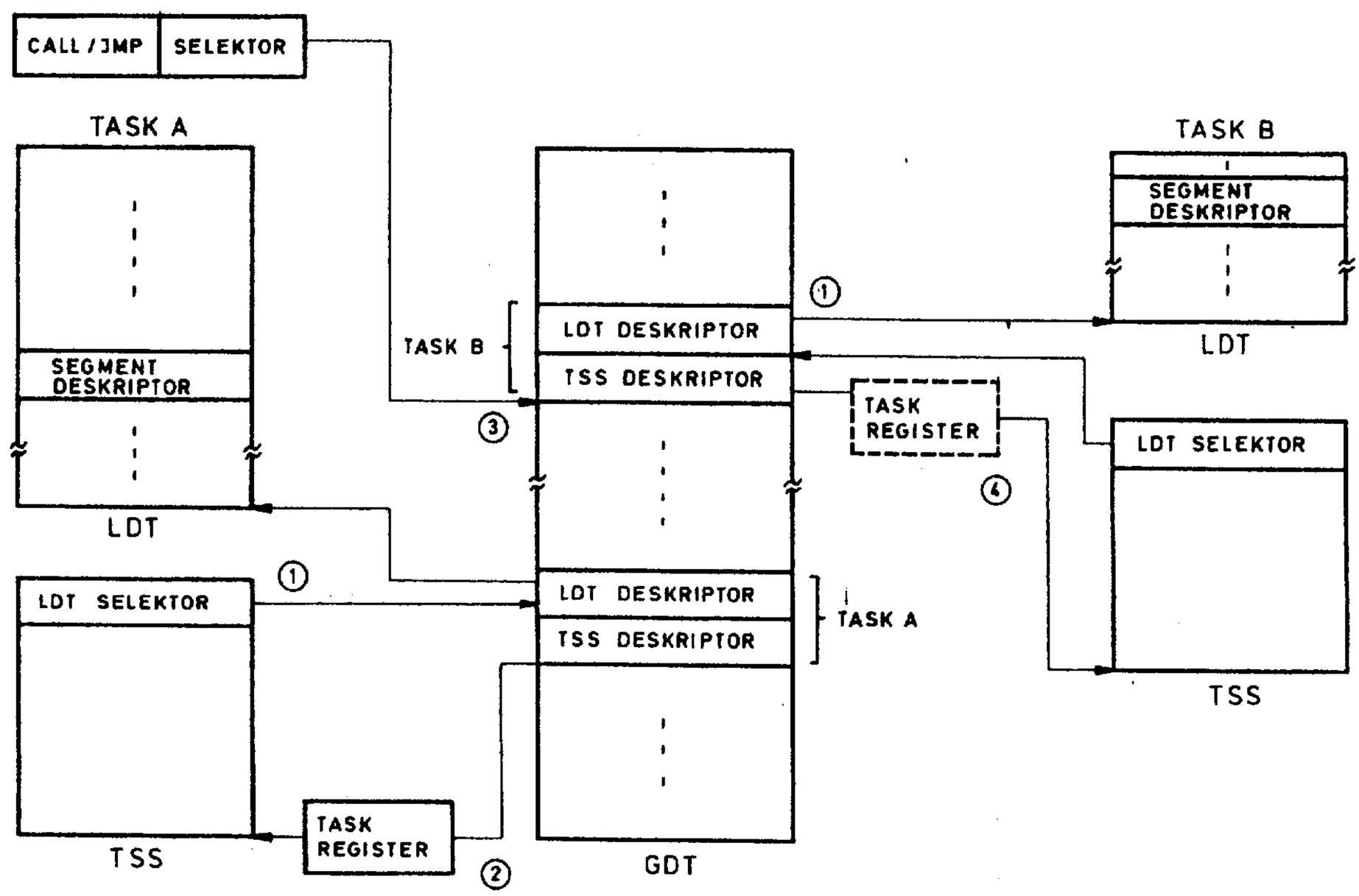
Der Versuch, von Task B in die aufrufende Task A mit einem normalen RET-Befehl zurückzukehren, stellt eine Ausnahmesituation dar und führt zur Unterbrechung des Programms. Die soll das folgende Bild verdeutlichen:



Im Code-Segment der Task A benützt ein CALL-Befehl den TSS-Deskriptor D1, um von Task A nach Task B zu wechseln. In diesem Fall müßte im Stack des E2-Code-Segments die Rücksprung-Adresse CS:EIP zum aufrufenden E1-Code-Segment aufbewahrt werden. Wie später gezeigt wird, ist für die angegebene Situation diese Technik unbrauchbar.

Da ein RET-Befehl lediglich eine Rücksprung-Adresse benützt, ***nicht*** aber einen TSS-Deskriptor, ist eine Rückkehr nach Task A nicht realisierbar. Für eine Rückkehr könnte aber im E2-Code-Segment z. B. ein JMP-Befehl den TSS-Deskriptor D2 benützen.

Die Hardware-Struktur, die einen gültigen Wechsel von Task A nach Task B unterstützt, wird im folgenden Bild gezeigt:



Es zeigt, daß die Tasks A und B durch ihre Task-Status-Segmente (TSS) und durch ihre lokalen Deskriptor-Tabellen (LDT) beschrieben werden. Sowohl die LDT- als auch die TSS-Deskriptoren befinden sich in der globalen Deskriptor-Tabelle GDT.
Dabei werden die LDT-Deskriptoren der individuellen Tasks durch entsprechende Werte in den LDT-Selektor-Feldern der zugehörigen Task-Status-Segmente ausgewählt ①.

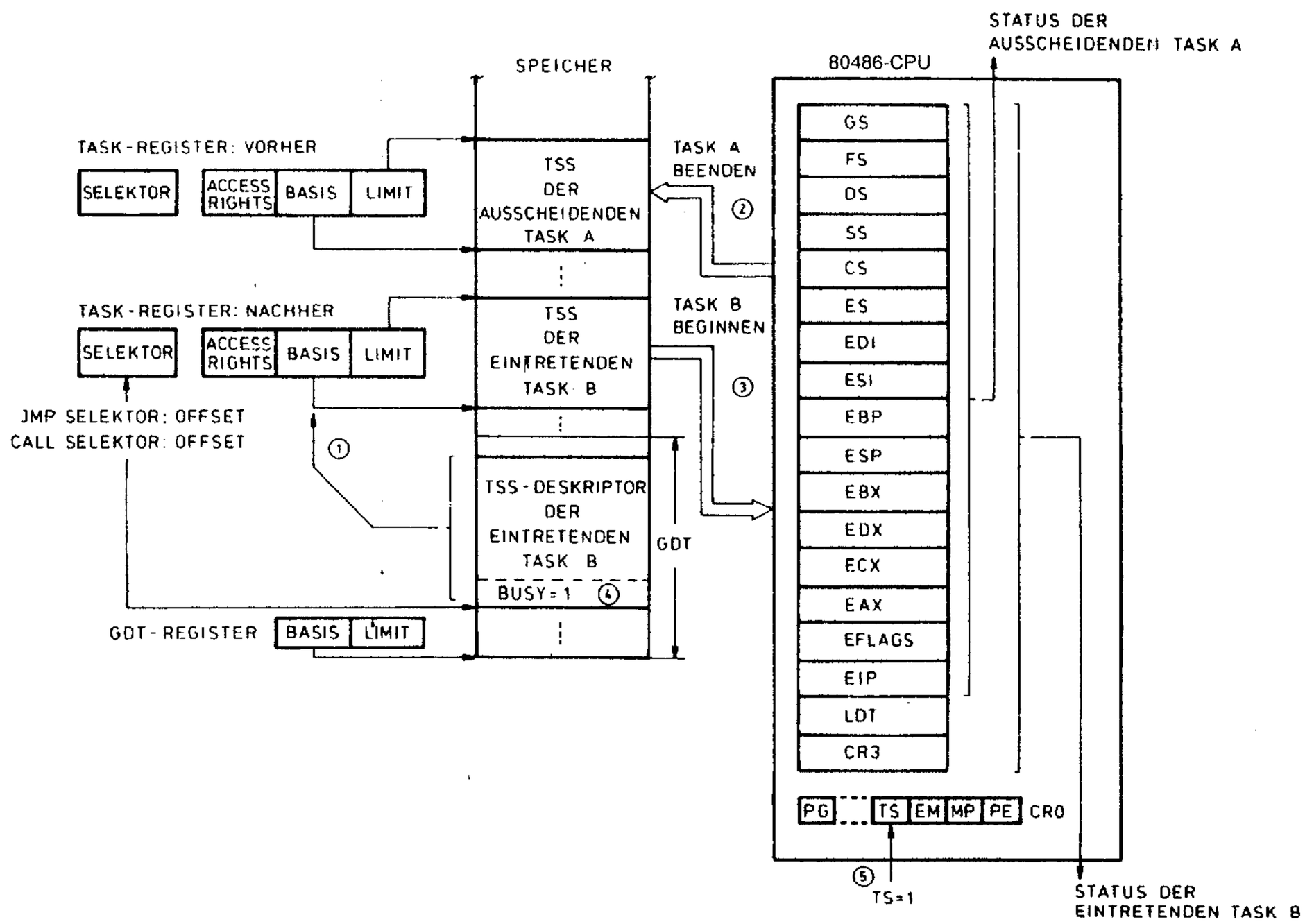
Da im Beispiel zunächst Task A aktiv ist, befindet sich ihr TSS-Deskriptor im Task-Status-Register der CPU ②. Benützt nun der Selektor eines CALL- oder JMP-Befehls in Task A den TSS-Deskriptor von Task B ③, wir das Task-Status-Register der CPU umgeladen. Es bekommt den TSS-Deskriptor der jetzt aktiven Task B ④.

Eine ***aktive*** Task (TSS-Deskriptor im Task-Register) wird zur ***ausscheidenden*** Task, wenn sie z. B. durch einen JMP- oder CALL-Befehl einen Task-Wechsel auslöst.

Wenn die Ausführung der ausgeschiedenen Task nach Beendigung einer anderen Task wieder aufgenommen wird, muß sie mit demjenigen Status weitermachen, den sie vor ihrer Ablösung hatte. Dies schließt ein, daß die Task das Progamm unmittelbar hinter dem Befehl fortsetzt, der den Task-Wechsel gestartet hat. Daher muß der richtige Rückkehr-Positionswert im EIP-Register enthalten sein.

Dies alles wird von der 80486-CPU realisiert, indem sie bei jedem Task-Wechsel den Status der ausscheidenden „Task" zwischenspeichert und die CPU-Register mit dem Status der ***eintretenden*** „Task" initialisiert.

Diese Operations-Schritte soll das folgende Bild verdeutlichen:



In der augenblicklich aktiven Task A löst ein JMP- oder CALL-Befehl einen Wechsel nach Task B aus. ***Bevor*** der Selektor den TSS-Deskriptor der eintretenden Task B in das CPU-Task-Cache-Register lädt ①, wird zunächst der augenblickliche CPU-Status der ausscheidenden Task A im zugehörigen Task-Status-Segment zwischengespeichert ②. Dieser Status entspricht den CPU-Registern EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI, ES, CS, SS, DS, FS, GS, EIP und den EFLAGS.

Dabei zeigt das EIP-Feld im TSS der ausscheidenden Task A auf den Befehl, der dem JMP- oder CALL-Befehl unmittelbar folgt.

Durch diese Operation ist es möglich, die Registerwerte für einen Neu-Start von Task A wieder in die CPU-Register zurückzuschreiben.

Als nächstes werden folgende Task-Status Segment-Felder der eintretenden Task B in die CPU-Register übertragen: GS, FS, DS, SS, CS, ES, EDI, ESI, EBP, ESP, EBX, EDX, ECX, EAX, EFLAGS, EIP, LDT-Selektor und CR3 ③. Ist die CPU initialisiert, beginnt die Ausführung von Task B, und das Type-Feld im Access-Byte des TSS-Deskriptors bekommt den Wert B (Busy-Bit = 1) ④.
Dadurch, daß die eintretende Task B als „BUSY“ gekennzeichnet ist, wird verhindert, daß sie sich selbst aufruft oder von anderen Tasks aufgerufen wird.

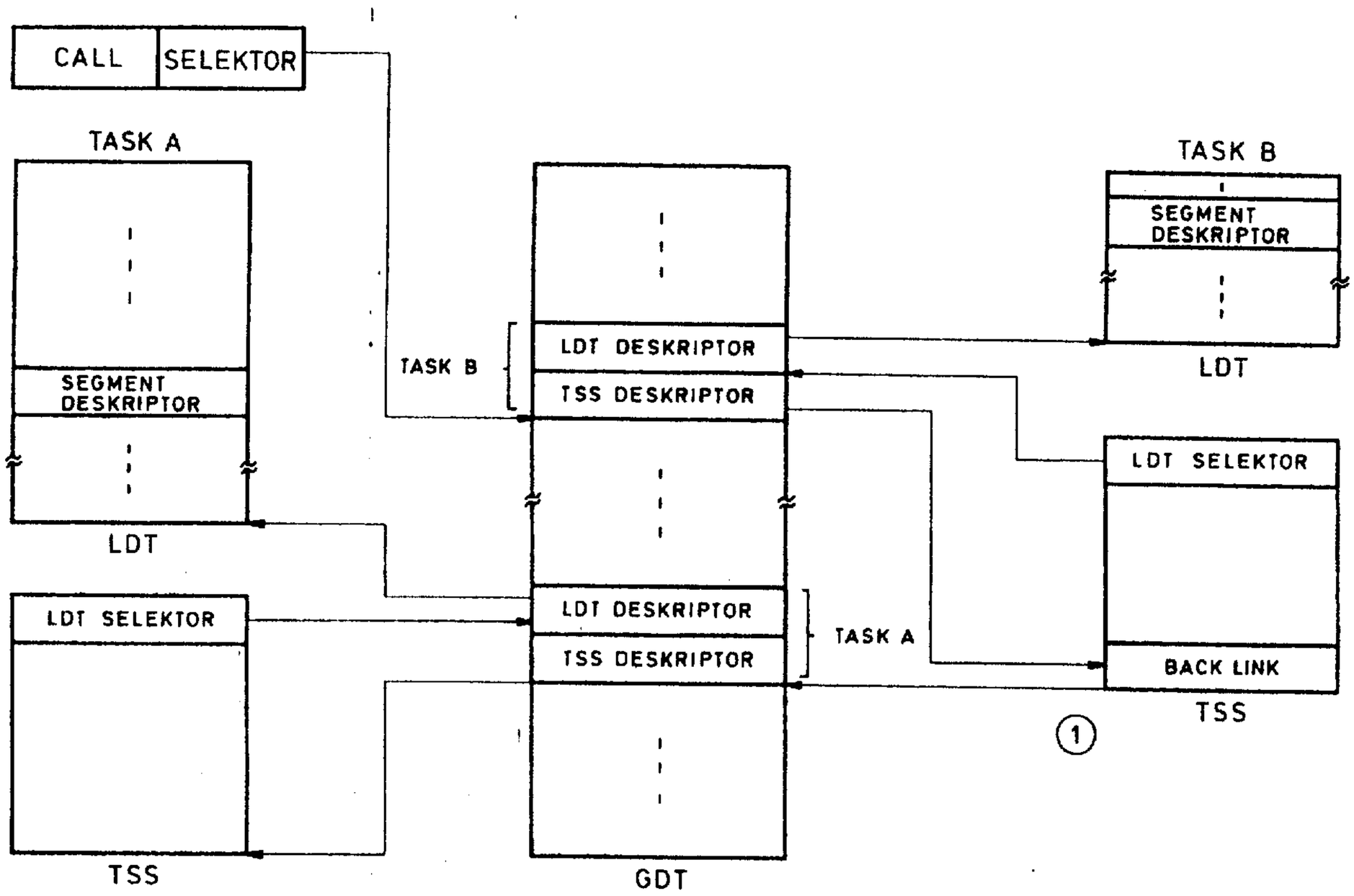
Schließlich wird noch das TS-Flag (***T***ask ***S***witched) im CR0-Register gesetzt ⑤.

Dieses Flag wir im Zusammenhang mit der numerischen Prozessor-Erweiterung benutzt. Dabei zeigt das TS-Bit an, daß der Status (Context) der Prozessor-Erweiterung ***nicht*** zur augenblicklichen 80486-Task gehört.

4.9 Verkettung von Tasks

Jedes Task-Status-Segment hat ein Feld, das als ***„BACK LINK“*** bezeichnet wird. Es enthält einen Selektor, der zum TSS-Deskriptor der ausscheidenden Task zeigt. Immer dann, wenn ein CALL-Befehl einen Task-Wechsel initiiert, wird automatisch das „BACK LINK“-Feld des neuen Task-Status-Segments mit dem Selektor des TSS-Deskriptors der ausscheidenden Task beschrieben.

Das folgende Bild illustriert dieses Mechanismus am Beispiel eines Wechsels von Task A nach Task B:

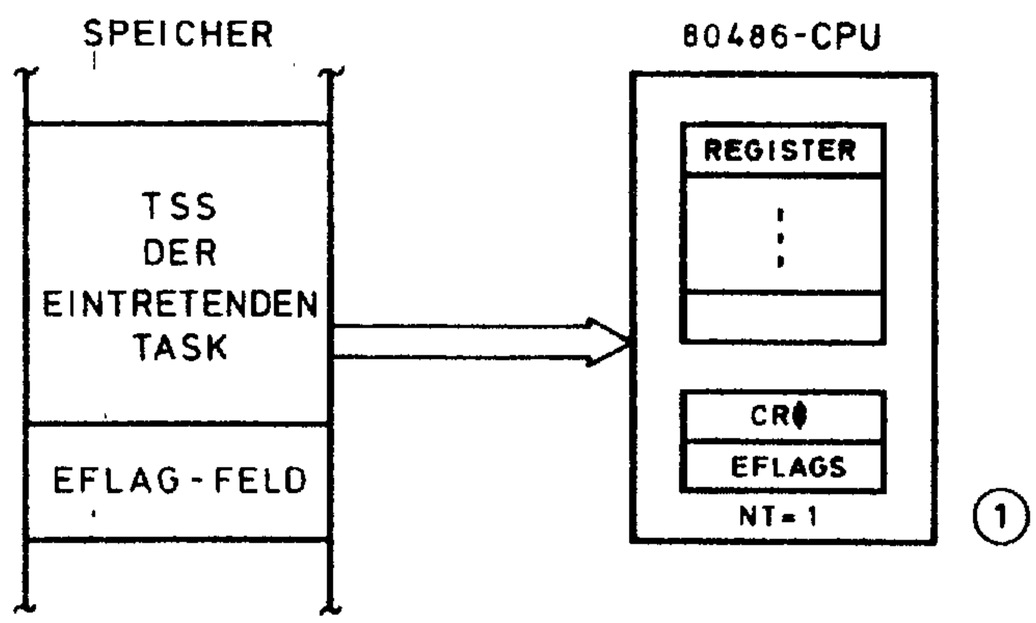


Es ist zu erkennen, daß nach dem erfolgten Task-Wechsel das „BACK LINK"-Feld des Task-B-Status-Segments den Selektor zum TSS-Deskriptor der ausscheidenden Task A enthält ①. Dadurch ist eine Verbindung zur vorherigen Task A hergestellt.

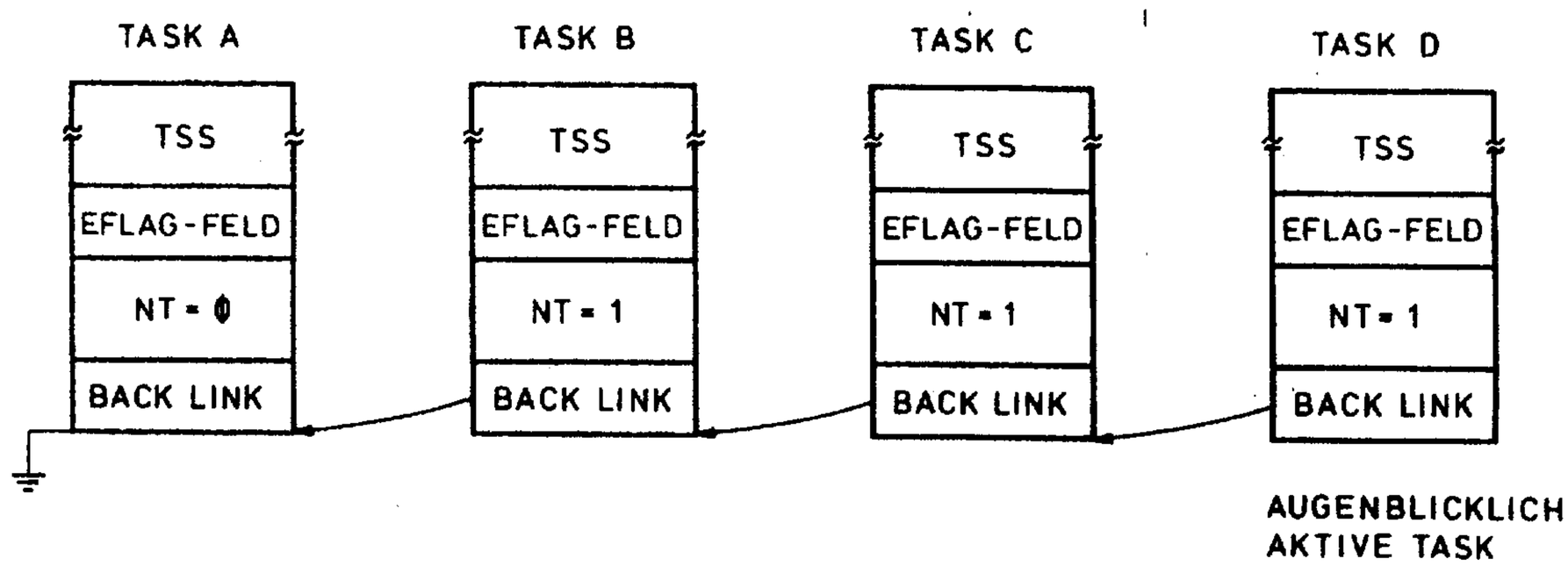
Diese Verbindung wird ausschließlich vom ***IRET***-Befehl benützt. Wie bereits früher erwähnt worden ist, können Task-Wechsel durch Interrupts, JMP-, CALL- und IRET-Befehle realisiert werden.

Während in JMP- und CALL-Befehlen Selektoren enthalten sind, ist dies beim IRET-Befehl nicht der Fall. Da aber für einen Task-Wechsel ein TSS-Deskriptor erforderlich ist, benützt der IRET-Befehl den zugehörigen Selektor im BACK LINK-Feld, um in die ausgeschiedene Task A zurückzukehren.

Jeder Task-Wechsel, initiiert durch eine CALL-Befehl, setzt automatisch das NT-Bit (***N***ested ***T***ask) im ***CPU***-EFLAG-Register der eintretenden Task ①.



Damit wird einem Programm angezeigt, daß eine verschachtelte Task-Struktur vorliegt bzw. daß sich im BACK LINK-Feld ein gültiger Selektor befindet. Das folgende Bild zeigt als Beispiel eine solche Struktur:



Es ist ersichtlich, daß jede Task in ihrem Status-Segment eine gültige Verbindung (NT = 1) zum Status-Segment ihres Vorgängers hat. Dadurch ist eine eindeutige Verkettungsordnung spezifiziert. Da die Task A keine Verbindung zu einem Vorgänger hat (NT = 0), enthält das BACK LINK-Feld ihres Status-Segments einen ***Null***-Selektor.

4.9.1 Task-Wechsel-Effekt durch JMP, CALL

Welchen Einfluß Task-Wechsel, die durch JMP- oder CALL-Befehle ausgelöst worden sind, auf

- das ***BUSY***-Bit im TSS-Deskriptor,
- das ***NT***-Bit im CPU-EFLAG-Register
- das ***BACK LINK WORD*** im Task-Status-Segment

der eintretenden und der ausscheidenden Tasks haben, zeigt die folgende Tabelle:

Beeinflußtes Feld	Art der Beeinflussung durch JMP	Art der Beeinflussung durch CALL
BUSY-Bit im TSS-Deskriptor der eintretenden Task	(Muß vorher 0 sein.) Wird gesetzt	(Muß vorher 0 sein.) Wird gesetzt
BUSY-Bit im TSS-Deskriptor der ausscheidenden Task	Wird gelöscht	Bleibt unverändert
NT-Bit im CPU-EFLAG-Register der eintretenden Task	Wird gelöscht	Wird gesetzt
NT-Bit im CPU-EFLAG-Register der ausscheidenden Task	Bleibt unverändert	Bleibt unverändert
„BACK LINK“-Feld im TSS der eintretenden Task	Bleibt unverändert	Es wird ein Selektor zum Task-Status-Segment der ausscheidenden „Task“ gesetzt
„BACK LINK“-Feld im TSS der ausscheidendenTask	Bleibt unverändert	Bleibt unverändert

4.10 Task-Gates

Um eine Task abzurufen, stehen beim 80486 mehrere unterschiedliche Mechanismen zur Verfügung. Einer dieser Mechanismen wird von Task-Gates unterstützt.

Task-Gates werden so wie Call-Gates von den Befehlen

- ***JMP SELEKTOR:OFFSET***
- ***CALL SELEKTOR:OFFSET***

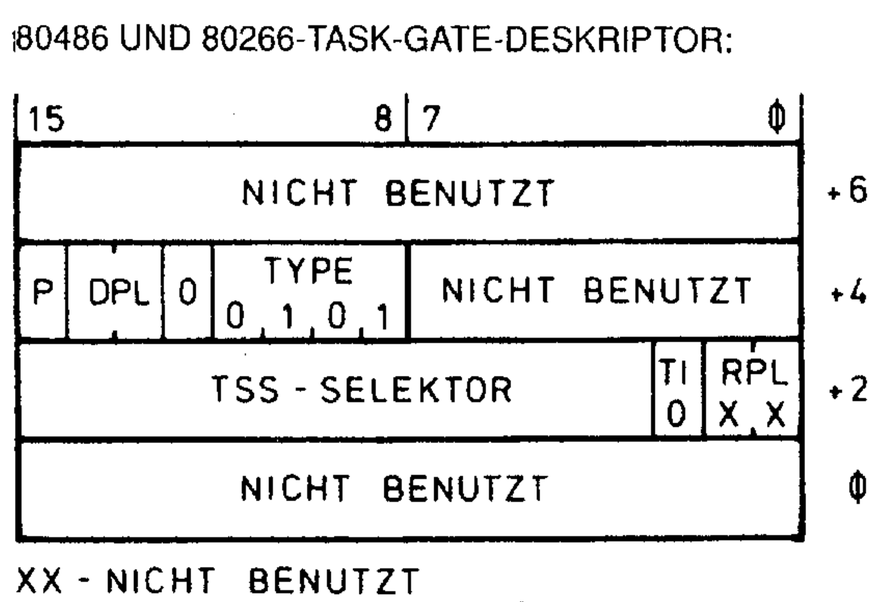
benützt.

Dabei wählt die Selektor-Komponente einer dieser Befehle den Deskriptor eines Task-Gates aus, der sich entweder

- in der GDT oder
- in der LDT

aufhalten kann. Die Offset-Komponenten dieser Befehle müssen zwar präsent sein, sie werden aber ignoriert.

Das komplette Layout eines Task-Gate-Deskriptors zeigt das folgende Bild:



4.10.1 Die Felder TSS-SELECTOR, TYPE, das Bit P (Present)

Das Feld TSS-SELECTOR
Dieses Feld enthält einen Selektor, der immer in der GDT (TI = 0) den Deskriptor eines Task-Status-Segments auswählt. Das RPL-Feld in diesem Selektor wird nicht benützt (X X).

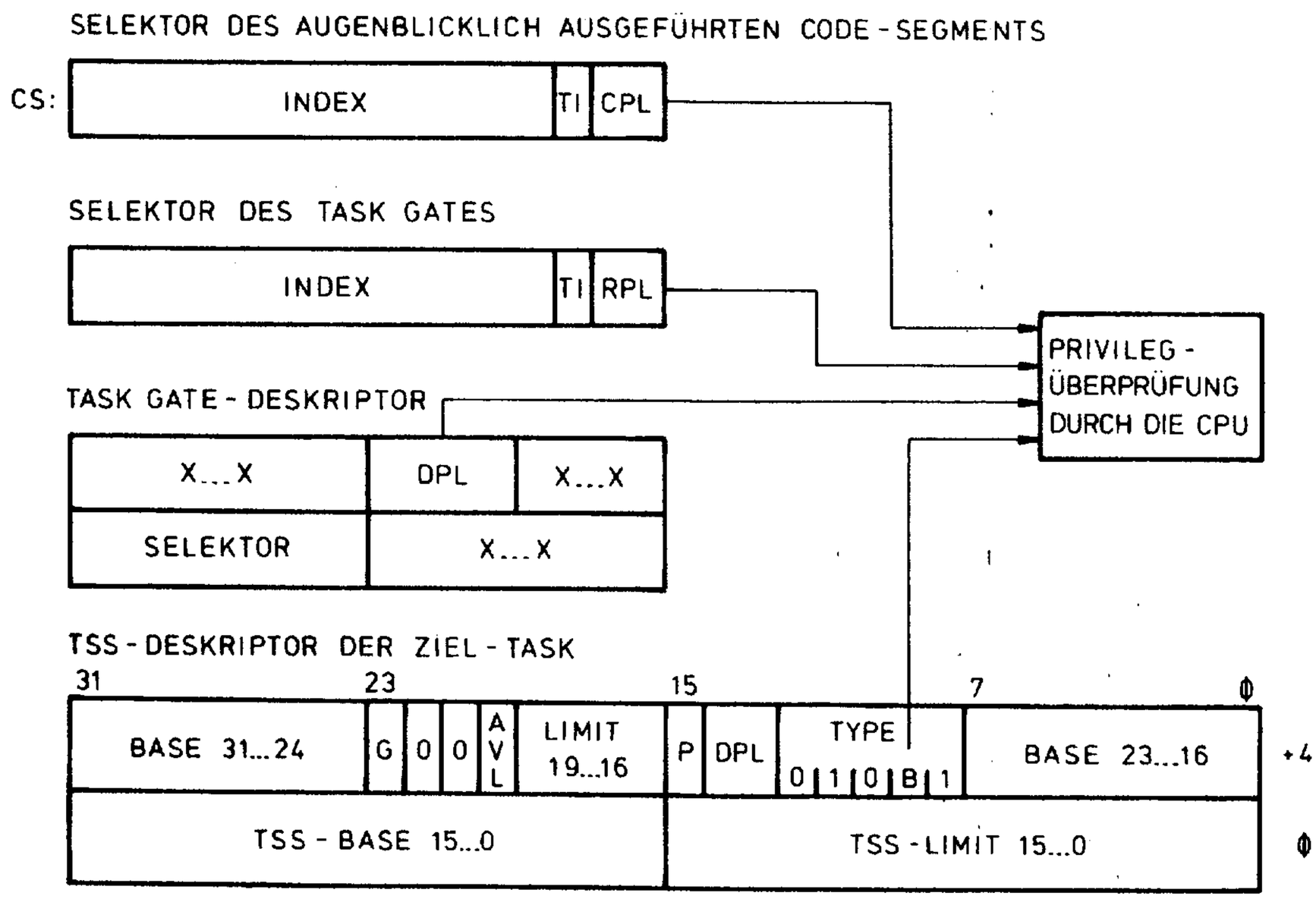
Das Feld TYPE
Da der angegebene Deskriptor die Klasse der Gate-Deskriptoren spezifiziert, wird im TYPE-Feld durch den Wert ***05*** angegeben, daß es sich um einen ***Task-Gate***-Deskriptor handelt.

Das Bit P (Present)
Genauso wie ein Call-Gate-Deskriptor, bezieht sich ein Task-Gate-Deskriptor ***nicht*** auf ein „normales" Segment. Daher hat das Present-Bit im Task-Gate-Deskriptor die gleiche Bedeutung wie das Present-Bit im Call-Gate-Deskriptor (Siehe Kap. 3.10.3).

4.10.2 Task-Gate-Deskriptor Privilege-Level (DPL)

Der Wert im DPL-Feld spezifiziert die Privileg-Ebene, auf der sich der Task-Gate-Deskriptor befindet. Genauso wie bei einem Call-Gate-Deskriptor überwacht auch hier DPL die Benützung des Task-Gate-Deskriptors durch einen JMP- oder CALL-Befehl. So kann DPL ein Programm daran ***hindern***, über einen Task-Gate-Deskriptor einen Task-Wechsel auszulösen.

Um die Gültigkeit des beabsichtigten „Intertask"-Programmablaufs über einen „Task Gate"-Deskriptor zu überprüfen, benützt der Prozessor ***vier*** verschiedene Felder:



Wie das angegebene Bild zeigt, sind dies:

- die augenblickliche Privilegebene (CPL) des augenblicklich ausgeführten Code-Segments,
- die Privilegebene (RPL) des vom CALL- oder JMP-Befehl gelieferten Selektors,
- die Privilegebene (DPL) im Task Gate-Deskriptor und
- das Busy-Bit (B) im Type-Feld des TSS-Deskriptors.

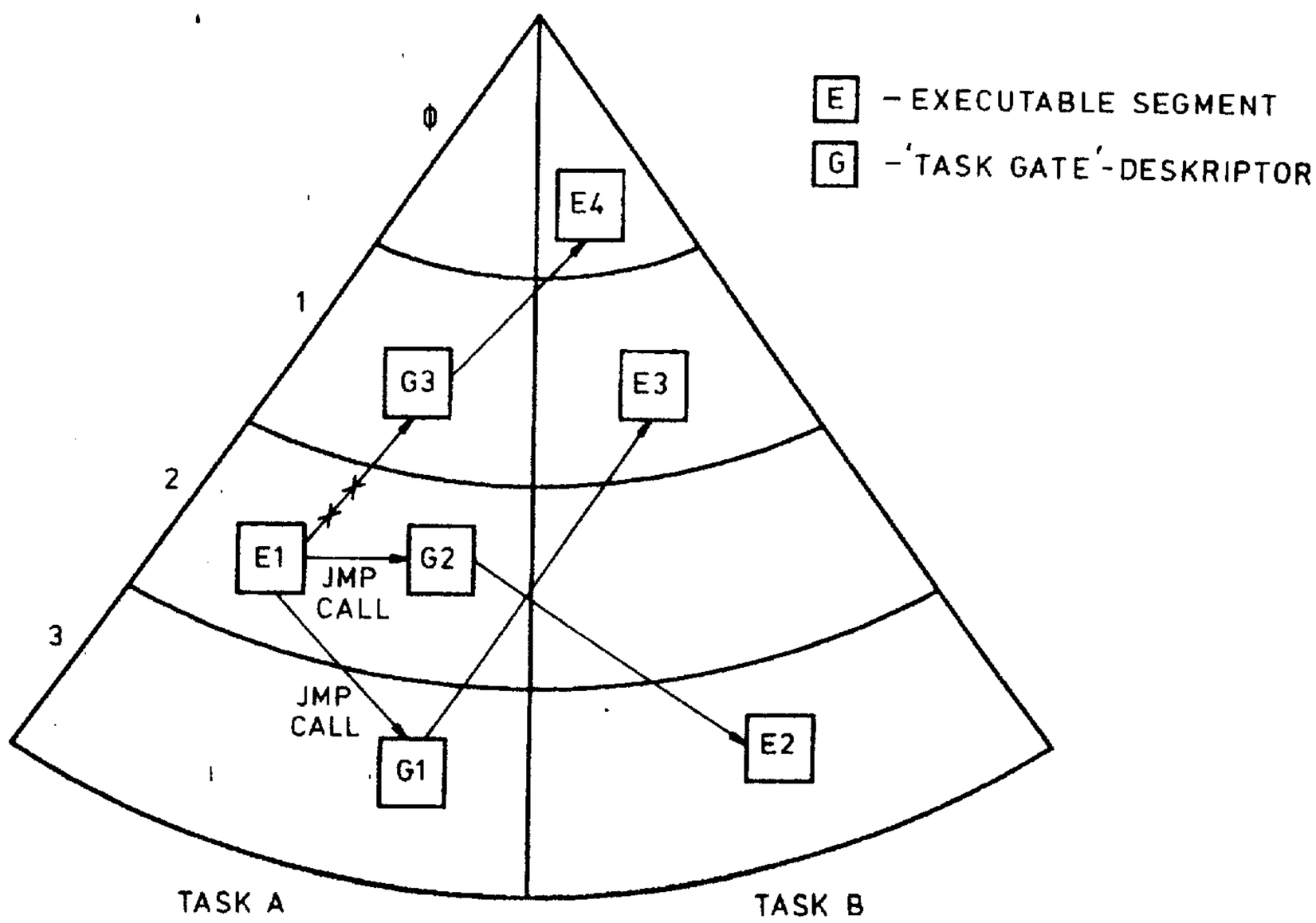
Ein Task-Wechsel über einen Task-Gate-Deskriptor ist nur realisierbar, wenn folgende Regel befolgt wird:

MAX(„Task" CPL, Selektor RPL) ≤ „Task Gate"-DPL

Es muß also die effektive Privileg-Ebene des JMP- oder CALL-Selektors numerisch ***kleiner*** oder ***gleich*** der Privileg-Ebene DPL des „Task Gate"-Deskriptors sein.

Zusätzlich muß das Busy-Bit im TSS-Deskriptor der eintretenden Ziel-Task ***gelöscht*** sein (B = 0), da eine aktive Task (B = 1) ***nicht*** aufgerufen werden kann.

Das folgende Bild illustriert gültige und ungültige Versuche, um z. B. einen Wechsel von Task A nach Task B zu realisieren:

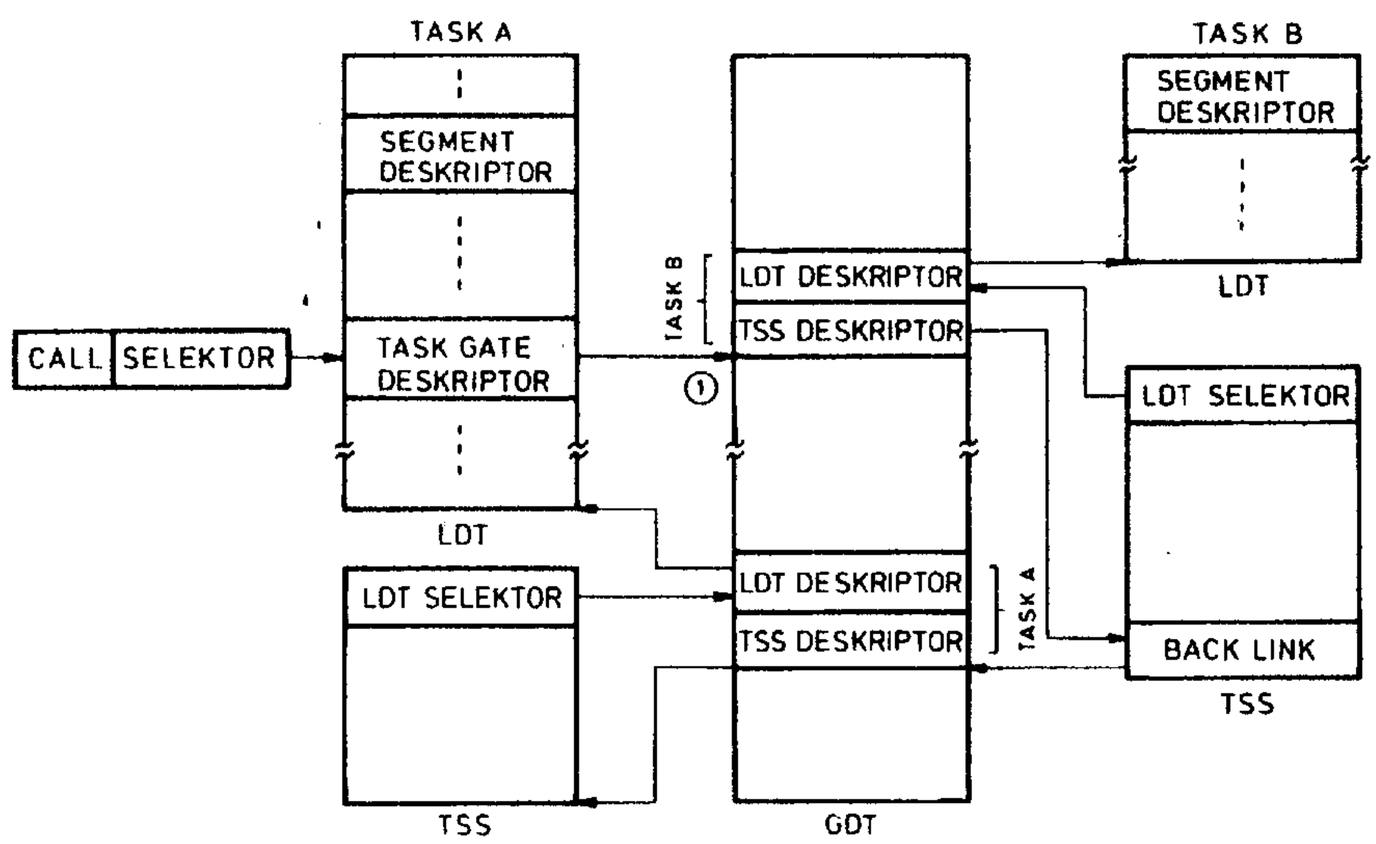


Es ist zu sehen, daß die Pfade ***E1-G2-E2*** und ***E1-G1-E3*** gültig sind, da hier jeweils die Bedingung MAX(„Task" CPL, Selektor RPL) ≤ „Task Gate"-DPL erfüllt ist. Im Gegensatz dazu ist der Pfad ***E1-G3-E4*** ungültig, da hier diese Bedingung nicht erfüllt ist.

Die TSS-Deskriptoren für alle Tasks müssen in der GDT untergebracht sein. Um zu verhindern, daß irgendeine Task eine andere Task unkorrekt aufruft, halten sich die TSS-Deskriptoren normalerweise in der Privileg-Ebene 0 auf.

Im Gegensatz dazu können Task-Gate-Deskriptoren auch in der LDT plaziert sein. Damit wird der private Zugriff auf die selektierten Tasks bei vollem Privileg-Schutz erlaubt.

Das folgende Bild illustriert als Beispiel einen Wechsel von Task A nach Task B über einen ***privaten*** Task-Gate-Deskriptor:



Hat die Selektor-Komponente des CALL-Befehls das Task-Gate als gültig erkannt, wählt der Selektor im Task-Gate mit TI = 0 (GDT) den TSS-Deskriptor der eintretenden Task B aus ①.

Ab diesem Zeitpunkt werden die gleichen Aktionen durchgeführt, wie sie bereits in den letzten Abschnitten dieses Kapitels beschrieben worden sind. Die daraus resultierende Hardware-Struktur ist daher die gleiche wie sie im Abschnitt „Verkettung von Tasks" angegeben ist.

4.11 Starten der ersten Task („Urtask")

Immer dann, wenn die Stromversorgung des 80486 eingeschaltet oder die CPU zurückgesetzt wird (RESET), werden bestimmte Register mit vordefinierten Werten initialisiert. Alle zusätzlich gewünschten Initialisierungen müssen dann von der Anwender-Software vorgenommen werden.

Der Prozessor-Status nach einem System-RESET ist folgender:

- ***EFLAGS = 00000002H***
  Dies bedeutet, daß der maskierbare Interrupt-Eingang INTR der 80486-CPU gesperrt ist (Interrupt-Flag IF = 0).

	31							24	23					16	15							8	7							0
EFLAGS:	-	-	-	-	-	-	-	-	-	-	-	-	-	AC	VM	RF	-	NT	IOPL		OF	DF	IF	TF	SF	ZF	-	AF	-	PF	-	CF
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

AC:	Alignment Check	OF: Overflow-Flag	ZF: Zero-Flag
VM:	Virtual 8086 Mode	DF: Direction-Flag	AF: Auxiliary Carry-Flag
RF:	Resume-Flag	IF: Interrupt Enable-Flag	PF: Parity-Flag
NT:	Nested Task-Flag	TF: Trap-Flag	CF: Carry-Flag
IOPL:	I/O-Privilege Level	SF: Sign-Flag	

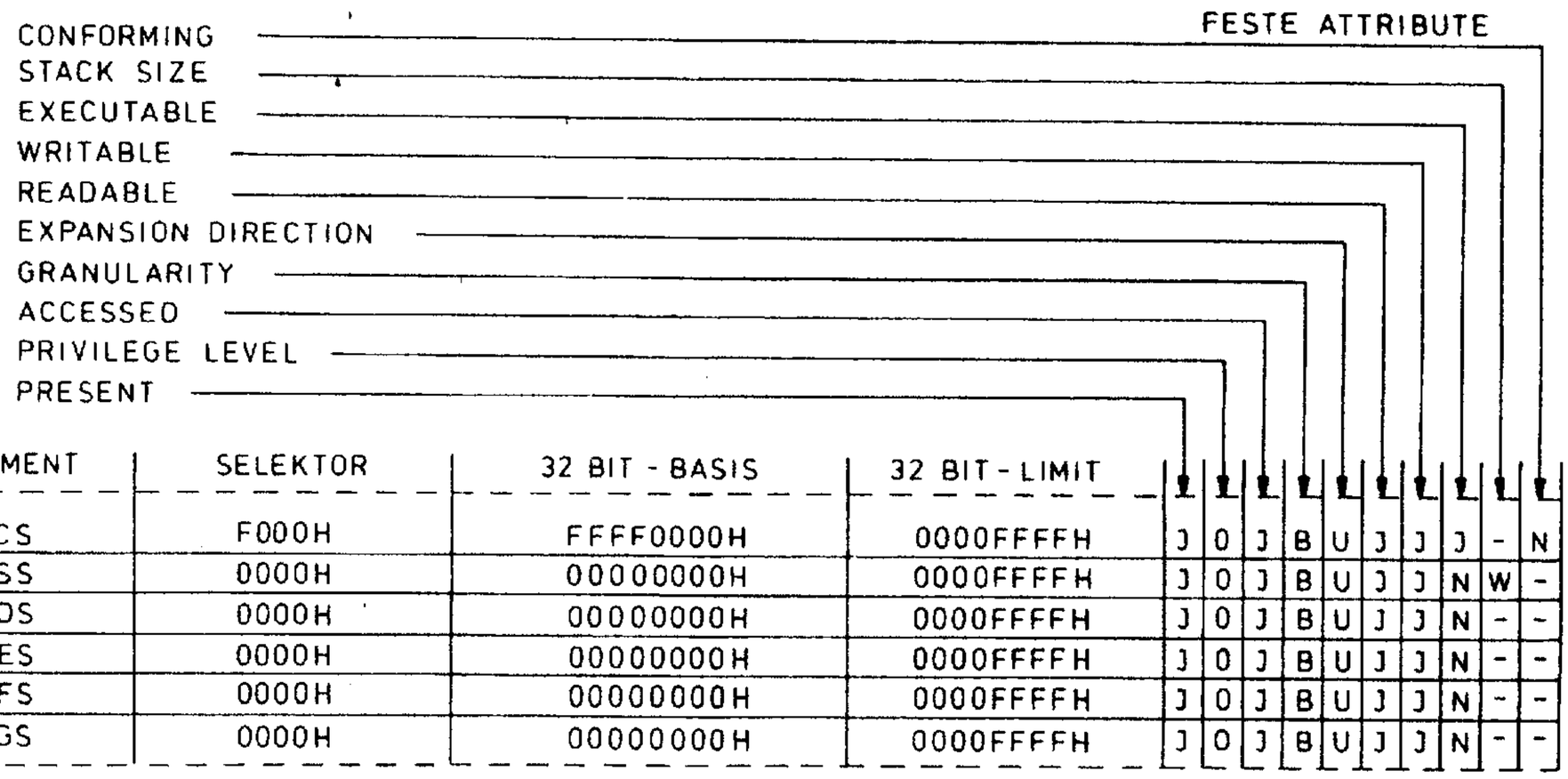
- ***Machine Control Register CR0 = 00000000H***
  Dies bedeutet, daß die 80486-CPU im „Real Address Mode" startet (Protection Enable PE = 0, Paging Enable PG = 0).

	31	30	29					24	23					18		16	15							8	7			5					0
CR0:	PG	CE	WT											AM		WP												NE	ET	TS	EM	MP	PE
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

HINWEIS: Der 80286 initialisiert das MSW-Register mit FFF0H, der 80486 mit 0000H. Dieser Unterschied hat allerdings keinen Effekt, weil die differierenden Bits beim 80286 undefiniert sind.

- Die ***Adreßleitungen*** $A_{31} - A_{20}$ werden automatisch auf High-Pegel gebracht.
- ***IP = FFF0H***
- ***IDT-Basis = 000000H, IDT-Limit = 3FFH***
- ***Segment-Deskriptor-Cache-Register***

Inhalte im Real Address Mode:



SEGMENT	SELEKTOR	32 BIT - BASIS	32 BIT - LIMIT									
CS	F000H	FFFF0000H	0000FFFFH	J	0	J	B	U	J	J	J	-	N
SS	0000H	00000000H	0000FFFFH	J	0	J	B	U	J	J	N	W	-
DS	0000H	00000000H	0000FFFFH	J	0	J	B	U	J	J	N	-	-
ES	0000H	00000000H	0000FFFFH	J	0	J	B	U	J	J	N	-	-
FS	0000H	00000000H	0000FFFFH	J	0	J	B	U	J	J	N	-	-
GS	0000H	00000000H	0000FFFFH	J	0	J	B	U	J	J	N	-	-

Schlüssel: J = Ja
B = Byte-granuliert
U = Expand UP
N = Nein
W = PUSH/POP 16-Bit Words
– = Wird für dieses Segment-Cache-Register nicht benützt

Nachdem der Prozessor in den „Protected Mode" umgeschaltet worden ist, kann er die erste Task („Urtask") starten, sobald ***zwei*** Bedingungen erfüllt sind:

Bedingung 1

- Es muß für diese Task ein gültiges Task-Status-Segment vorhanden sein. Dabei müssen die Register-Felder der „Urtask" passende Werte enthalten. So ist es notwendig, daß die Register-Felder auf ***gültige*** Segmente zeigen oder den Wert 0 haben.
- Die Stack-Anfangszeiger SS:ESP müssen ebenfalls auf ***gültige*** Stack-Segmente zeigen. Dabei müssen ihre Privileg-Ebenen numerisch ***kleiner*** oder ***gleich*** der augenblicklichen „Urtask"-CPL sein.
- Der LDT-Selektor muß auf einen gültigen LDT-Deskriptor in der GDT zeigen oder den Wert 0 haben, wenn die Task keine LDT benützt.
- Schließlich sollte das BACK LINK-Feld im Urtask-Status-Segment den Wert 0 enthalten, da die erste Task keinen Vorgänger hat.

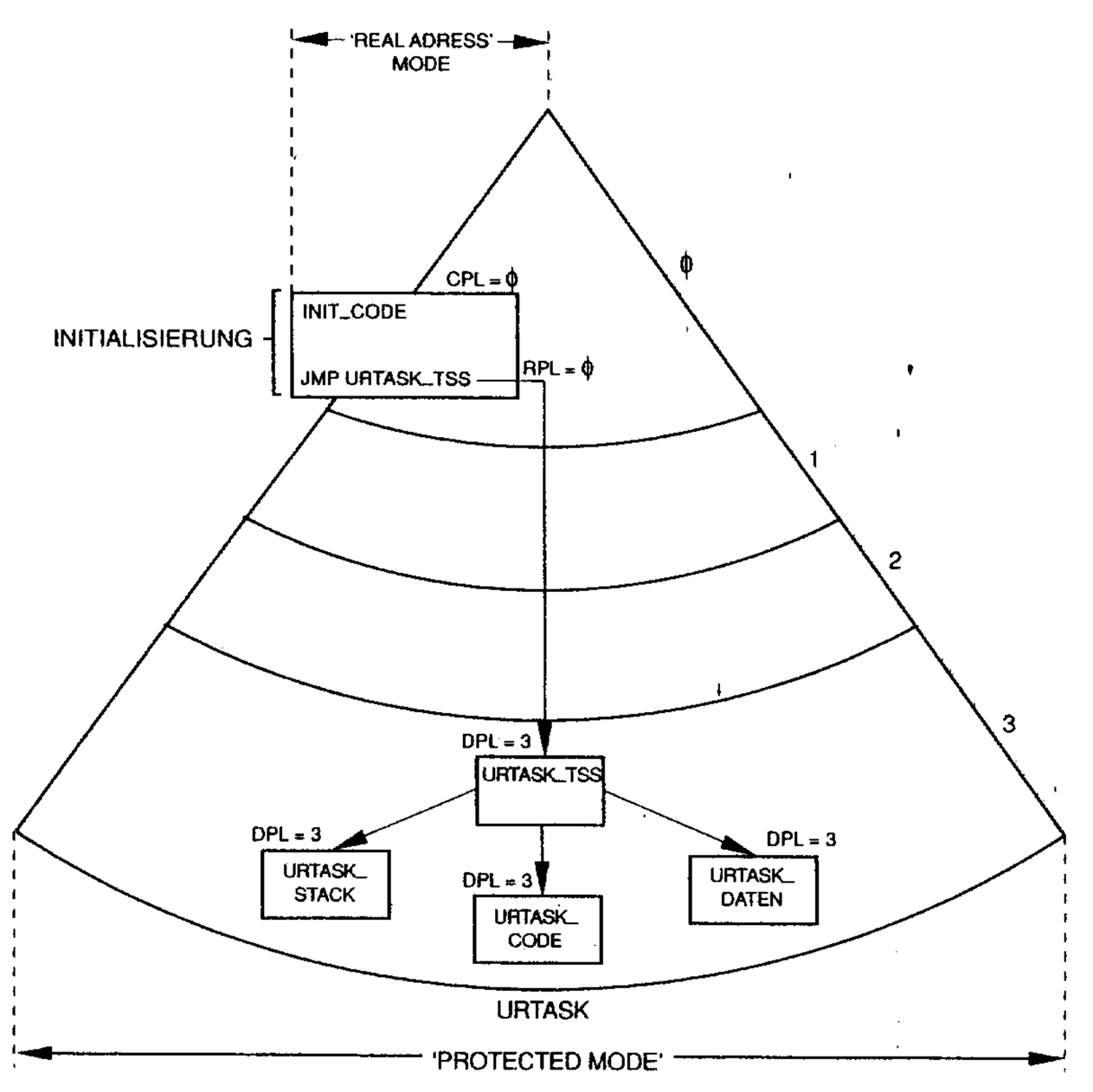
Bedingung 2

- Damit der Übergang in die „Urtask" realisiert werden kann, ist es notwendig, den augenblicklichen Prozessor-Status zwischenzuspeichern. Zu diesem Zweck muß das CPU-Task-Register TR auf einen dafür erforderlichen Speicherbereich zeigen.
- Ist der Übergang in die „Urtask" erfolgt, werden die darin aufbewahrten Informationen ***nicht*** mehr benötigt. Dieser Speicherbereich kann dann für andere Zwecke benützt werden.

Sind die oben genannten Bedingungen erfüllt, kann die erste Task in einfacher Weise dadurch gestartet werden, daß ein FAR JMP-Befehl über den Deskriptor eines Tasks-Status-Segments oder über ein Task Gate auf diesen Deskriptor ausgeführt wird.

4.11.1 Programmierbeispiel: „Urtask"

Das folgende Programm-Modul enthält einen ***ablauffähigen*** ASM386/486-Code zur Initialisierung eines geschützten virtuellen Adreß-Modus-Systems.
Während der Initialisierungsphase bringt das Modul die 80486-CPU zunächst in den „Protected Mode" und ruft anschließend die erste Task („Urtask") über einen TSS-Deskriptor auf.



Die angegebene Software-Struktur illustriert diesen Vorgang. Dabei ist zu erkennen, daß die Initialisierung teilweise im „Real Address Mode" und teilweise im „Protected Mode" stattfindet.

Hat das Modul den Prozessor mit MOV CR0, EAX (PE-Bit setzen) in den „Protected Mode" umgeschaltet, befindet sich das Initialisierungs-Segment INIT_CODE in der Privileg-Ebene 0. Ab diesem Zeitpunkt ist es möglich, geschützte Befehle wie z. B. LTR (Load Task Register) auszuführen.

Im angegebenen Beispiel erfolgt in INIT_CODE (CPL = 0) mit JMP URTASK_TSS (RPL = 0) über den TSS-Deskriptor URTASK_TSS (DPL = 3) der Sprung in die erste „Task". Dies ist erlaubt, da die Bedingung

- MAX (CPL(0),Selektor RPL(0)) ≤ TSS-Deskriptor DPL(3)

erfüllt ist.

Das Modul arbeitet in Verbindung mit Spezifikationen des System Builders BLD386/486, die für den Anwender „unsichtbar“ sind.
So initialisiert „BLD386/486“ die Felder des Urtask-Status-Segments URTASK_TSS und definiert die Deskriptoren für das Urtask-Code-Segment URTASK_CODE, das Urtask-Daten-Segment URTASK_DATEN, das Urtask-Stack-Segment URTASK_STACK, aber auch für das Initialisierungs-Segment INIT_CODE.

Die Felder des Urtask-Status-Segments besetzt „BLD386/486“ mit folgenden Anfangswerten, vorausgesetzt, daß die „Urtask“ eine eigene LDT hat.

	URTASK_TSS 31 16	15 0	BYTE OFFSET
URTASK_LDT_DESK in der RAM-GDT. Offset = 18H, TI = 0, RPL = 0	0000H	TASK-LDT-SELEKTOR : 0018H	60
Nicht benutzt	0000H	GS-SELEKTOR : 0000H	5C
Nicht benutzt	0000H	FS-SELEKTOR : 0000H	58
URTASK_DATEN_DESK in der RAM-LDT. Offset = 18H, TI = 1, RPL = 3	0000H	DS-SELEKTOR : 001FH	54
URTASK_STACK_DESK in der RAM-LDT. Offset = 20H, TI = 1, RPL = 3	0000H	SS-SELEKTOR : 0027H	50
URTASK_CODE_DESK in der RAM-LDT. Offset = 10H, TI = 1, RPL = 3	0000H	CS-SELEKTOR : 0017H	4C
Nicht benutzt	0000H	ES-SELEKTOR : 0000H	48
Nicht initialisiert	EDI	: 0000 0000H	44
Nicht initialisiert	ESI	: 0000 0000H	40
Nicht initialisiert	EBP	: 0000 0000H	3C
Laenge des URTASK_STACK-Segments ist 10 Bytes	ESP	: 0000 000AH	38
Nicht initialisiert	EBX	: 0000 0000H	34
Nicht initialisiert	EDX	: 0000 0000H	30
Nicht initialisiert	ECX	: 0000 0000H	2C
Nicht initialisiert	EAX	: 0000 0000H	28
Interrupt gesperrt. IF – Bit = 0	EFLAGS	: 0000 0000H	24
Einsprungstelle in das URTASK_CODE-Segment	EIP	: 0000 0000H	20
Nicht benutzt	CR3 (Page Directory-Basis)	: 0000 0000H	1C
Nicht benutzt	0000H	SS für CPL2 : 0000H	18
Nicht benutzt	SP für CPL2	0000 0000H	14
Nicht benutzt	0000H	SS für CPL1 : 0000H	10
Nicht benutzt	SP für CPL1	0000 0000H	C
Nicht benutzt	0000H	SS für CPL0 : 0000H	8
Nicht benutzt	SP für CPL0	0000 0000H	4
Nicht benutzt	0000H	BACK LINK : 0000H	0

Damit „BLD386/486" das EIP-Feld mit dem Wert der Einsprungstelle in das URTASK_CODE-Segment initialisieren kann, muß die Einsprungstelle dem „BUILDER" durch PUBLIC zugänglich gemacht werden. Dem Programm-Listing ist zu entnehmen, daß diese Einsprungstelle durch das Symbol START gekennzeichnet ist.

Damit JMP URTASK_TSS den Sprung in die „Urtask" realisieren kann, muß zur Ablage des augenblicklichen Prozessor-Status das CPU-Task-Register TR einen hierfür notwendigen Speicherbereich kennzeichnen.

Zu diesem Zweck enthält die EPROM-GDT bzw., nach dem Kopieren in den RAM-Bereich, die RAM-GDT einen weiteren von „BLD386/486" spezifizierten Deskriptor, der nach der Ausführung von

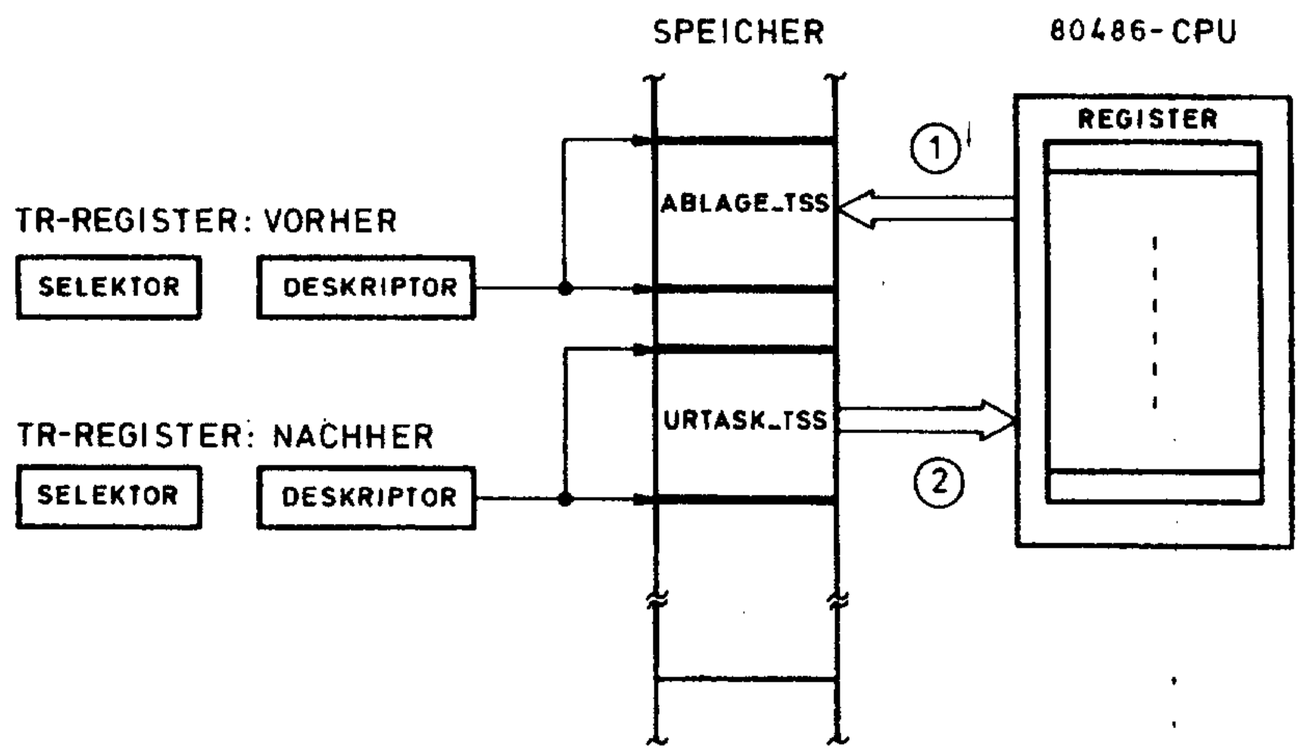
- ***LEA AX,CS:ABLAGE_TSS_DESK***
- ***LTR AX***

dieses Task-Status-Ablagesegment definiert.

Der nachfolgende Befehl

- JMP URTASK_TSS

speichert nun den augenblicklichen Prozessor-Status in diesem Ablage-Segment ① und überträgt das von „BLD386/486" spezifizierte Urtask-Status-Segment URTASK_TSS in die CPU.



Wie bereits erwähnt, ist das Task-Status-Ablagesegment ***nicht*** mehr notwendig und kann für andere Zwecke verwendet werden.
Die TSS-Felder der eintretenden „Urtask" werden nun benützt, um in der FAR-Prozedur URTASK, die sich im URTASK_CODE-Segment aufhält, die ersten beiden Byte-Elemente des Feldes VEKTOR, das sich im URTASK_DATEN-Segment aufhält, zu duplizieren.

Weil das Urtask-Status-Segment URTASK_TSS von „BLD386/486" spezifiziert wird und deswegen in einem externen Modul zur Verfügung steht, ist dies dem Assembler ASM386/486 mit

- ***EXTRN URTASK_TSS:FAR***

mitzuteilen.

Wie das Programm-Listing zeigt, enthält die von „BLD386/486“ spezifizierte System-Komposition eine GDT, eine IDT, eine LDT, ein Ablage-Status-Segment und ein Urtask-Status-Segment. Alle diese Segmente sind im EPROM gespeichert und werden zu Manipulationszwecken ins RAM kopiert.

Die Positionen der GDT, IDT und LDT sind in ihren Pseudo-Deskriptoren spezifiziert. „BLD386/486“ benützt diese Deskriptoren, um darin die Positionen der zugehörigen System-Segmente zu plazieren. Daher müssen die Pseudo-Deskriptoren dem „BUILDER“ mit

- ***PUBLIC GDT_PSEUDO_DESK,IDT_PSEUDO_DESK,LDT_PSEUDO_DESK***

zugänglich gemacht werden.

Der im Programmierbeispiel angegebene ASM386/486-Code ist in einem ***Haupt***-Modul enthalten. Dies ist durch die Angabe der Element-Liste in der ***END***-Direktive zu erkennen.

- ***Das Element CS:INIT*** kennzeichnet die ***Start***-Adresse des Moduls. Es gibt an, daß nach dem Laden des Programms CS mit dem Segment-Selektor des Levels INIT (INIT_CODE) und IP mit dem Offset von INIT initialisiert ist. Dabei muß das Segment den Zugriffstyp EO oder ER und das ***USE16-Attribut*** haben.
- ***Das Element DS:URTASK_DATEN*** gibt an, daß DS mit dem Selektor des URTASK_DATEN-Segments initialisiert ist. Sein Zugriffstyp muß RO, ER oder RW sein.
- ***Das Element SS:URTASK_STACK*** gibt an, daß SS mit dem Selektor des Stack-Segments URTASK_STACK initialisiert ist. Dieses Segment ist durch die STACKSEG-Direktive definiert.

```
LOC       OBJ            LINE SOURCE

                          1          NAME    B17
                          2
                          3 ;Definiere das Layout fuer einen Deskriptor
                          4
--------                  5 DESK               STRUC
00000000                  6 LIMIT_0_15         DW   0          ;Segment-Obergrenze (Bits 0 .. 15)
00000002                  7 BASIS_0_15         DW   0          ;Basisadresse (Bits 0 .. 15)
00000004                  8 BASIS_16_23        DB   0          ;Basisadresse (Bits 16 .. 23)
00000005                  9 ZUGRIFF            DB   0          ;Access Rights Byte
00000006                 10 GRAN               DB   0          ;Flags und Segment-Obergrenze
                         11                                    ;(Bits 16 .. 19)
00000007                 12 BASIS_24_31        DB   0          ;Basisadresse (Bits 24 .. 31)
--------                 13 DESK               ENDS
                         14
    0001                 15 PE                 EQU  1          ;Protected Mode freigeben
                         16
                         17 EXTRN URTASK_TSS:FAR
                         18 PUBLIC GDT_PSEUDO_DESK, IDT_PSEUDO_DESK, LDT_PSEUDO_DESK
                         19
    ----                 20 INIT_CODE          SEGMENT ER USE16  ;Ausfuehrbares und lesbares Segment
                         21                                    ;mit Basis-Adresse=FFFF0000H und
                         22                                    ;DPL=0 (Festlegung durch BLD386/486)
    FFF0                 23                    ORG  0FF0H      ;RESET-Adresse=FFFFFFF0H
    FFF0 E94105          24          JMP       NEAR PTR INIT   ;Das CS-Register behaelt den
                         25                                    ;Wert F000H
```

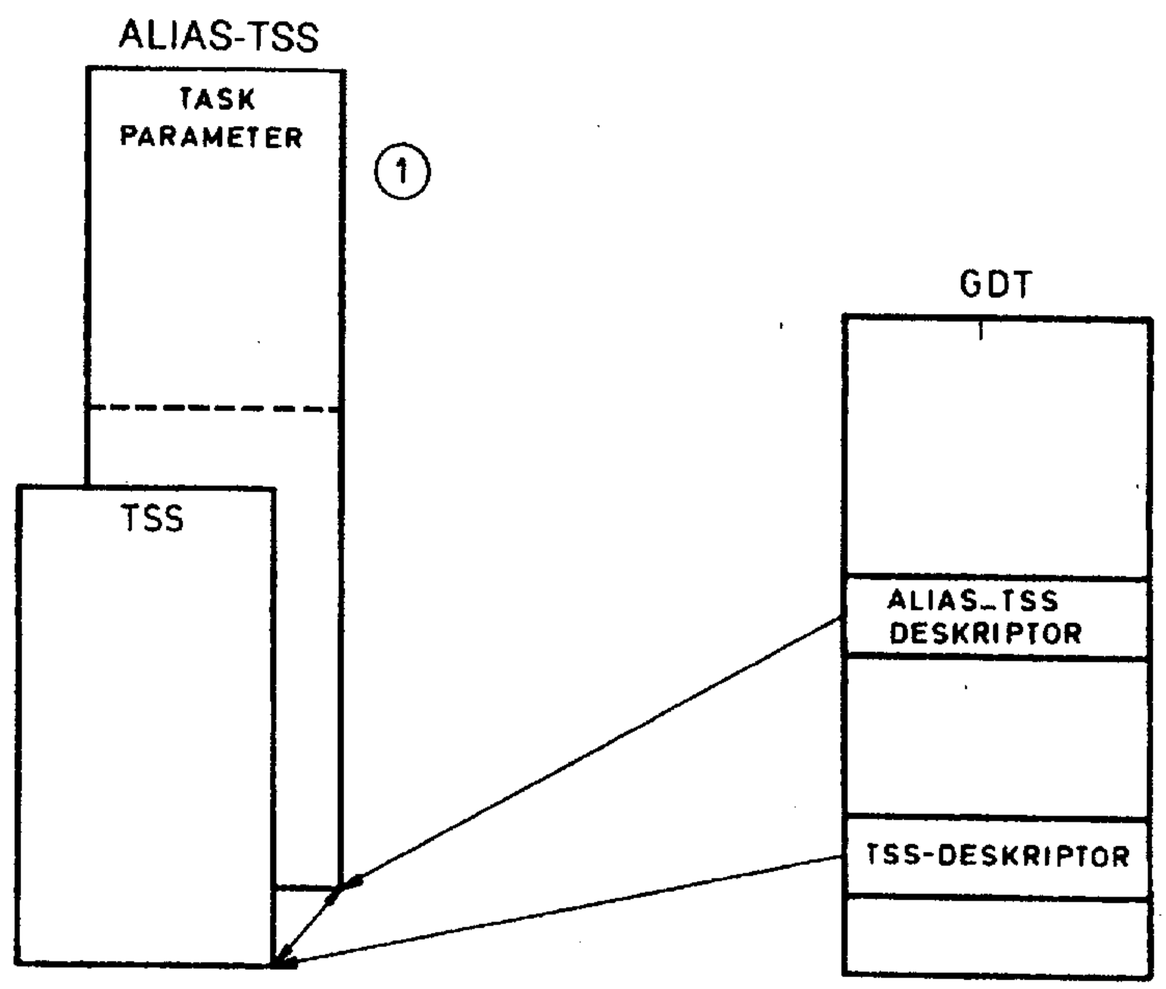
```
0000                    26                  ORG  0                  ;Die EPROM-GDT-Adresse=FFFF0000H
                        27
0000                    28  EPROM_START LABEL WORD
                        29
0000 (8                 30  GDT_NULL_DESK    DB SIZE DESK DUP(?)    ;Luecke und Null Deskriptor
     ??
     )
0008 (8                 31  GDT_ALIAS_DESK   DB SIZE DESK DUP(?)    ;Alias-Deskriptor der GDT
     ??
     )
0010 (8                 32  IDT_ALIAS_DESK   DB SIZE DESK DUP(?)    ;Alias-Deskriptoren der IDT
     ??
     )
0018 (8                 33  URTASK_LDT_DESK DB SIZE DESK DUP(?)     ;Deskriptor der Urtask-LDT
     ??
     )
0020 (8                 34  ABLAGE_TSS_DESK DB SIZE DESK DUP(?)     ;Ablage-TSS-Deskriptor
     ??
     )
0028 (8                 35  URTASK_TSS_DESK DB SIZE DESK DUP(?)     ;Urtask-TSS-Deskriptor
     ??
     )
0030 (8                 36  INIT_CODE_DESK    DB SIZE DESK DUP(?)    ;Deskriptor des Initialisierungs-
     ??
     )
                        37                                           ;Segments
                        38
0038 ????????????       39  GDT_PSEUDO_DESK DP    ?                  ;Reservierung von 6 Bytes fuer den
                        40                                           ;GDT-Pseudo-Deskriptor
003E ????????????       41  IDT_PSEUDO_DESK  DP    ?                 ;Reservierung von 6 Bytes fuer den
                        42                                           ;IDT-Pseudo-Deskriptor
0044 ????????????       43  LDT_PSEUDO_DESK DP    ?                  ;Reservierung von 6 Bytes fuer den
                        44                                           ;LDT-Pseudo-Deskriptor
                        45
0100                    46                  ORG  100H                ;Die EPROM-LDT Adresse=FFFF0100H
                        47
0100 (8                 48  LDT_NULL_DESK    DB SIZE DESK DUP(?)    ;Luecke und Null Deskriptor
     ??
     )
0108 (8                 49  LDT_ALIAS_DESK   DB SIZE DESK DUP(?)    ;Alias-Deskriptor der LDT
     ??
     )
010  (8                 50  URTASK_CODE_DESK  DB SIZE DESK DUP(?)   ;Deskriptor des Urtask-Codesegments
     ??
     )
0118 (8                 51  URTASK_DATEN_DESK  SIZE DESK DUP(?)     ;Deskriptor des Urtask-Datensegments
     ??
     )
0120 (8                 52  URTASK_STACK_DESK  DB SIZE DESK DUP(?)  ;Deskriptor des Urtask-Stacksegments
    ??
    )
                        53
0200                    54                  ORG  200H                ;Die EPROM-IDT Adresse=FFFF0200H
                        55
0200 (256               56  INTERRUPT_DESK  DB 32*SIZE DESK DUP(?)  ;Deskriptoren fuer 32 Interrupt-
     ??
     )
                        57                                           ;Prozeduren
0400                    58                  ORG  400H                ;Die Adresse des EPROM-Ablage-
                        59                                           ;Status-Segments=FFF0400H
0400 (26                60                  DD   26 DUP(?)           ;TSS enthaelt 26 Doppelworte
     ????????
     )
```

```
                                61
0500                            62                ORG   500H          ;Die Adresse des EPROM-Urtask-
                                63                                    ;Status-Segments=FFFF0500H
0500 (26                        64                DW    26 DUP(?)     ;TSS enthaelt 26 Doppelworte
     ????
     )
                                65
0534                            66 EPROM_ENDE LABEL     WORD
                                67
0534 FC                         68 INIT:     CLD                       ;Benuetze Autoinkrement-Modus
0535 31FF                       69            XOR     DI,DI             ;RAM-Zielindex loeschen
0537 89FE                       70            MOV     SI,DI             ;EPROM-Quellindex loeschen
0539 B99A02                     71            MOV     CX,(EPROM_ENDE-EPROM_START)/2
                                72                                      ;Setze Laenge des EPROM-Bereichs
053C F32EA5                     73 REP        MOVS WORD PTR ES:[DI],WORD PTR CS: [SI]
                                74                                      ;EPROM-GDT, EPROM-IDT und
                                75                                      ;EPROM-Pseudo-Deskriptoren ab Adresse
                                76                                      ;FFFF0000H in RAM ab Adresse
                                77                                      ;00000000H kopieren
053F 8D1E3800          R        78            LEA     BX,CS:GDT_PSEUDO_DESK ;BX=effektive Adresse des
                                79                                      ;   GDT-Pseudo-Deskriptors
0543 260F0117                   80            LGDT    ES:[BX]           ;Basisadresse und Tabellen-Obergrenze
                                81                                      ;der RAM-GDT festlegen
0547 8D1E3E00          R        82            LEA     BX,CS:IDT_PSEUDO_DESK ;BX=effektive Adresse des
                                83                                      ;   IDT-Pseudo-Deskriptors
054B 260F011F                   84            LIDT    ES:[BX]           ;Basisadresse und Tabellen-Obergrenze
                                85                                      ;der RAM-IDT festlegen
054F 660F20C0                   86            MOV     EAX,CR0           ;EAX bekommt augenblicklichen
                                87                                      ;CR0-Status
0553 660D01000000               88            OR      EAX,PE            ;Setzte PE-Bit
0559 660F22C0                   89            MOV     CR0,EAX           ;Schalte in den Protected Mode
055D EB00                       90            JMP     SHORT ENTFERNE    ;Real Address Mode-Befehle aus der
055F                            91 ENTFERNE:                            ;Instruction Queue entfernen
055F 8D062000          R        92            LEA     AX,CS:ABLAGE_TSS_DESK ;AX enthaelt den Selektor des
                                93                                      ;Ablage-TSS-Deskriptors
0563 0F00D8                     94            LTR     AX                ;Deskriptor in das Task-Register laden
0566 66EA00000000- - - -E       95            JMP     URTASK_TSS        ;Sprung in die Urtask ueber einen
                                96                                      ;TSS-Deskriptor
                                97
- - - -                         98 INIT_CODE          ENDS
***WARNING #377 IN 98, (PASS 2) SEGMENT CONTAINS PRIVILEGED INSTRUCTION(S)
                                99
- - - - - - - -                 100 URTASK_CODE       SEGMENT EO USE32  ;Nur ausfuehrbares Segment mit
                                101                                     ;Basis-Adresse=1000H und DPL=3
                                102                                     ;(Festlegung durch BLD386/486)
                                103 ASSUME DS:URTASK_DATEN
                                104
00000000                        105 URTASK  PROC      FAR
                                106
                                107 PUBLIC START
00000000 66A100000000          R 108 START:  MOV     AX,WORD PTR VEKTOR ;AX bekommt die ersten zwei
                                109                                     ;Vektor-Elemente
00000006 66A302000000          R 110         MOV     WORD PTR VEKTOR+2,AX;Vektor-Elemente duplizieren
0000000C EBFE                   111 SCHL:     JMP     SCHL
0000000E                        112 URTASK  ENDP
                                113
- - - - - - - -                 114 URTASK_CODE       ENDS
                                115
- - - - - - - -                 116 URTASK_DATEN      SEGMENT RW USE32  ;Schreib- und lesbares Segment mit
                                117                                     ;Basis-Adresse=2000H und DPL=3
                                118                                     ;(Festlegung durch BLD386/486)
00000000 11                     119 VEKTOR DB         11H,22H,?,?       ;Vektor mit vier Byte-Elementen
00000001 22
```

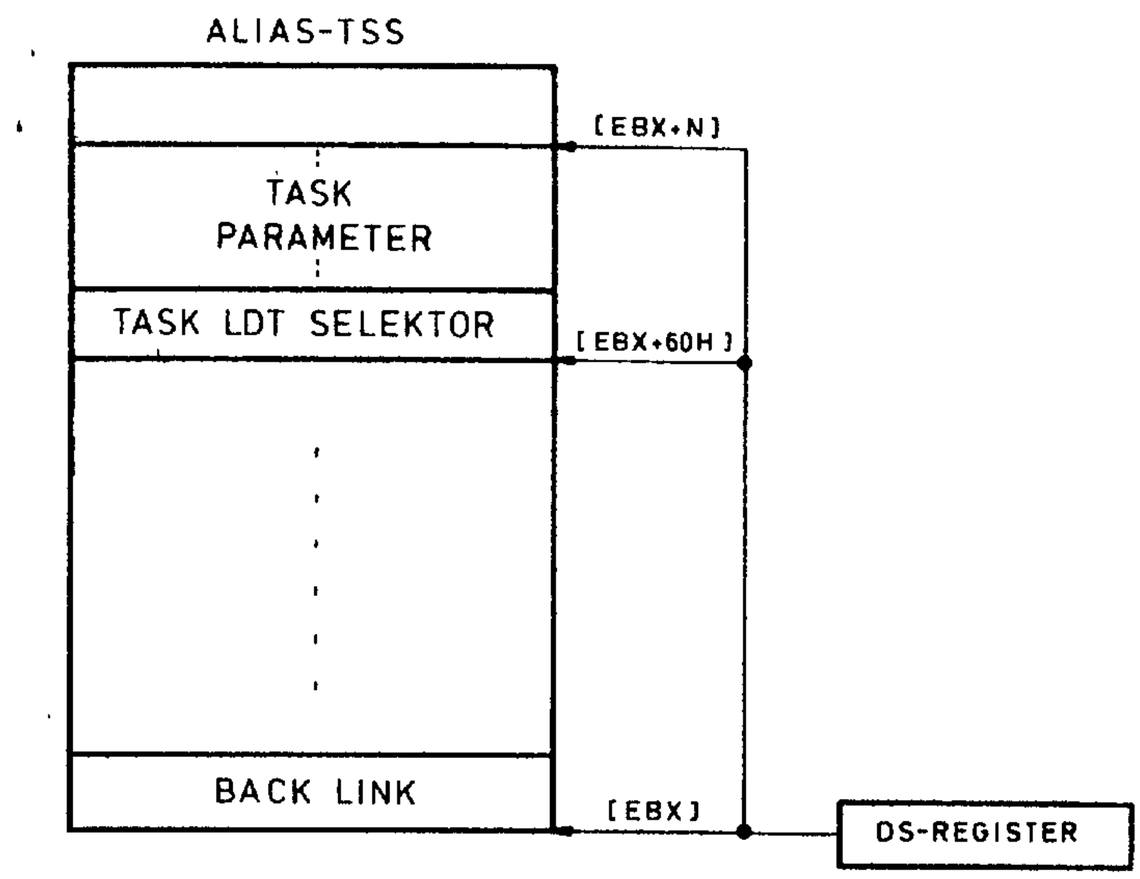
```
00000002 ??
00000003 ??
                        120
--------                121 URTASK_DATEN    ENDS
                        122
--------                123 URTASK_STACK    STACKSEG 10          ;Stack-Segment mit einer Laenge von
                        124                                      ;10 Bytes. Basis-Adresse=3000H und
                        125                                      ;DPL=3 (Festlegung durch BLD386/486)
                        126
                        127         END CS:INIT,DS:URTASK_DATEN,SS:URTASK_STACK
ASSEMBLY COMPLETE, 1 WARNING, NO ERRORS.
```

4.12 Task-Status-Alias-Segmente und -Deskriptoren

Damit das Betriebssystem auf die Task-Status-Felder zugreifen kann, muß in der GDT mindestens ein weiterer Deskriptor existieren, der das Task-Status-Segment als schreib- und lesbares Alias-Segment definiert.
Diese Situation verdeutlicht das folgende Bild:



Es ist ersichtlich, daß das vom Alias-TSS-Deskriptor beschriebene Task-Status-Alias-Segment (ALIAS-TSS) ***nicht*** die gleiche Länge haben muß wie das originale Task-Status-Segment (TSS). Dadurch ist es möglich, die Task mit zusätzlichen Parametern wie z. B. „Priorität der Task“ zu versorgen ①.

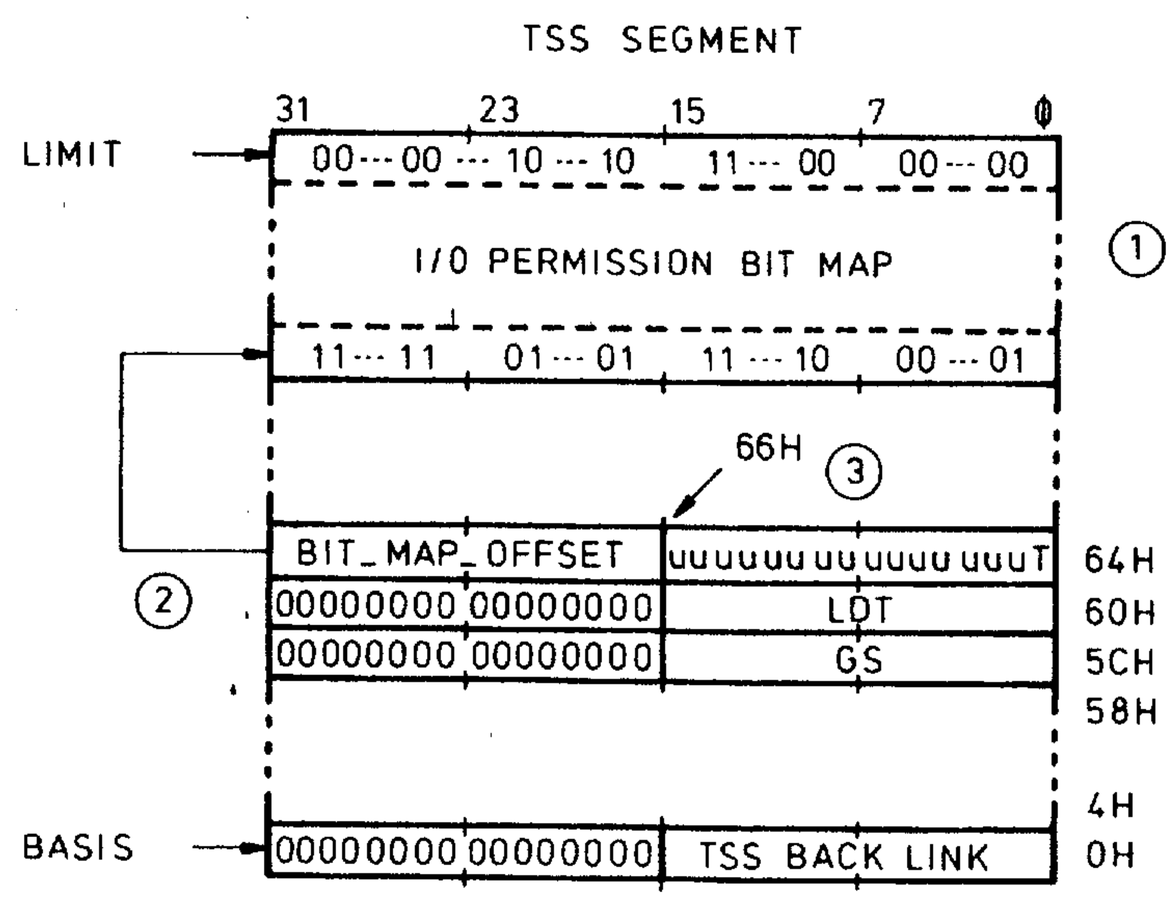


Um nun die Felder im Task-Status-Segment zu erreichen, können z. B. die Zeiger DS: [EBX] bis DS: [EBX + N] benützt werden.

4.13 Die I/O Permission Bit Map

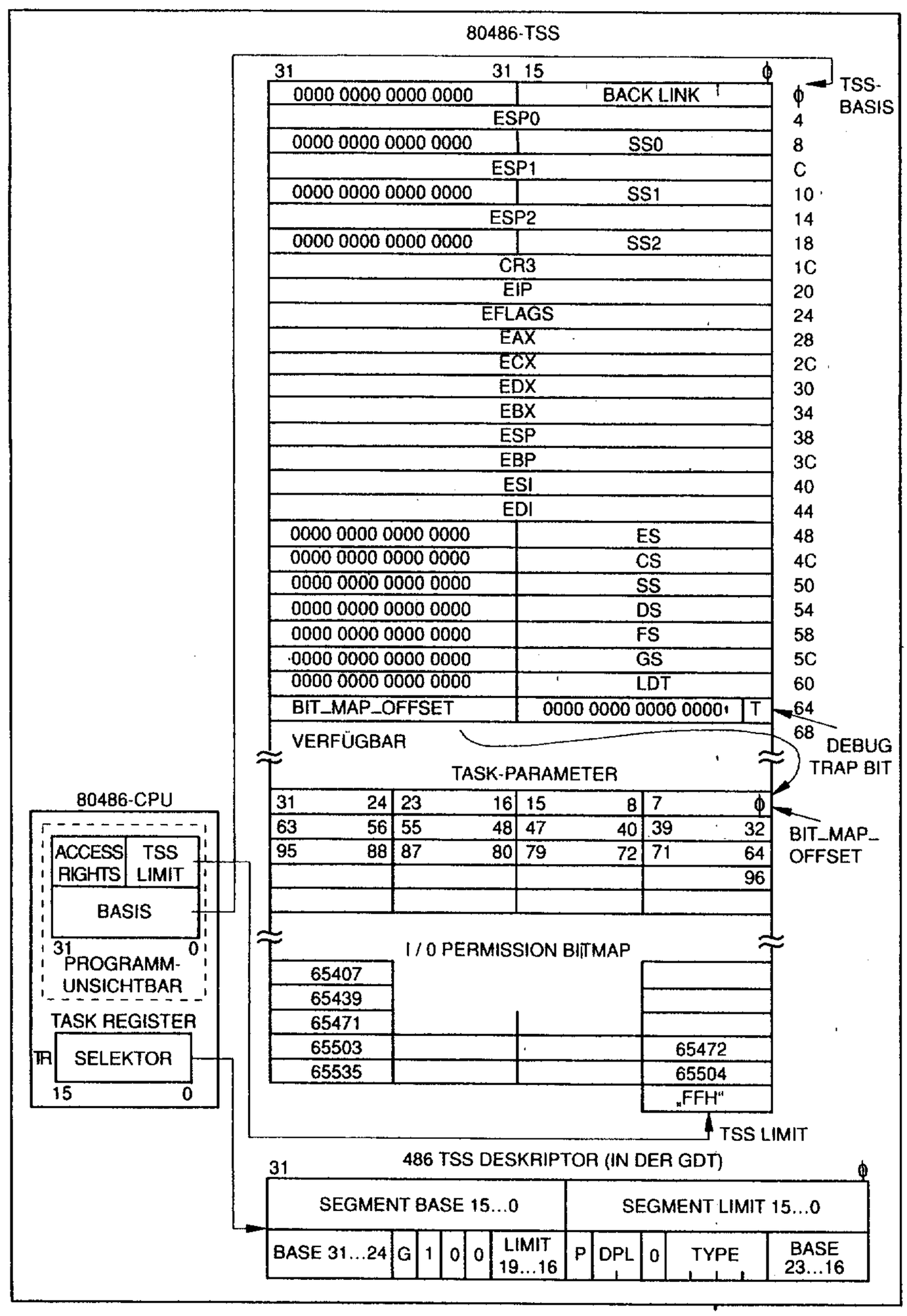
Immer dann, wenn sich der 80486 mit I/O-Befehlen wie IN, INS, OUT und OUTS auf seinen I/O-Raum bezieht, hat er die Möglichkeit, Portzugriffe auf spezifische I/O-Adressen selektiv zu sperren oder freizugeben.

Die Struktur, die diese I/O-Referenzen steuert, ist die sogenannte ***„I/O Permission Bit Map“*** im Task-Status-Segment ①.



Sie kann als ***Bit-String*** aufgefaßt werden, dessen Länge im Bereich 0 ... 64 KBits variabel ist. Die Start-Position der „I/O Permission Bit Map“ bestimmt das „BIT_MAP_OFFSET“-Feld im Task-Status-Segment ②. Dieses Feld enthält einen 16 Bit-Pointer, der bei Offset ***66H*** (102 dezimal) relativ zur TSS-Basis zu finden ist ③.

Das komplette Layout eines 80486-Task-Status-Segments einschließlich der mit dem Segment verbundenen Hardware-Struktur zeigt das folgende Bild:



Es ist zu erkennen, daß nach einem für das Betriebssystem verfügbaren Adreßraum die „I/O Permission Bit Map" mit maximal 65536 Bits (64 KBits) ab der Start-Adresse BIT_MAP_OFFSET folgt. Unmittelbar hinter dem letzten Byte dieses Bit-Strings liegt das sogenannte Terminator-Byte. Dieses Byte enthält ausschließlich 1-Bits (FFH) und muß abhängig von der Länge des Bit-Strings bei der TSS-Obergrenze liegen.

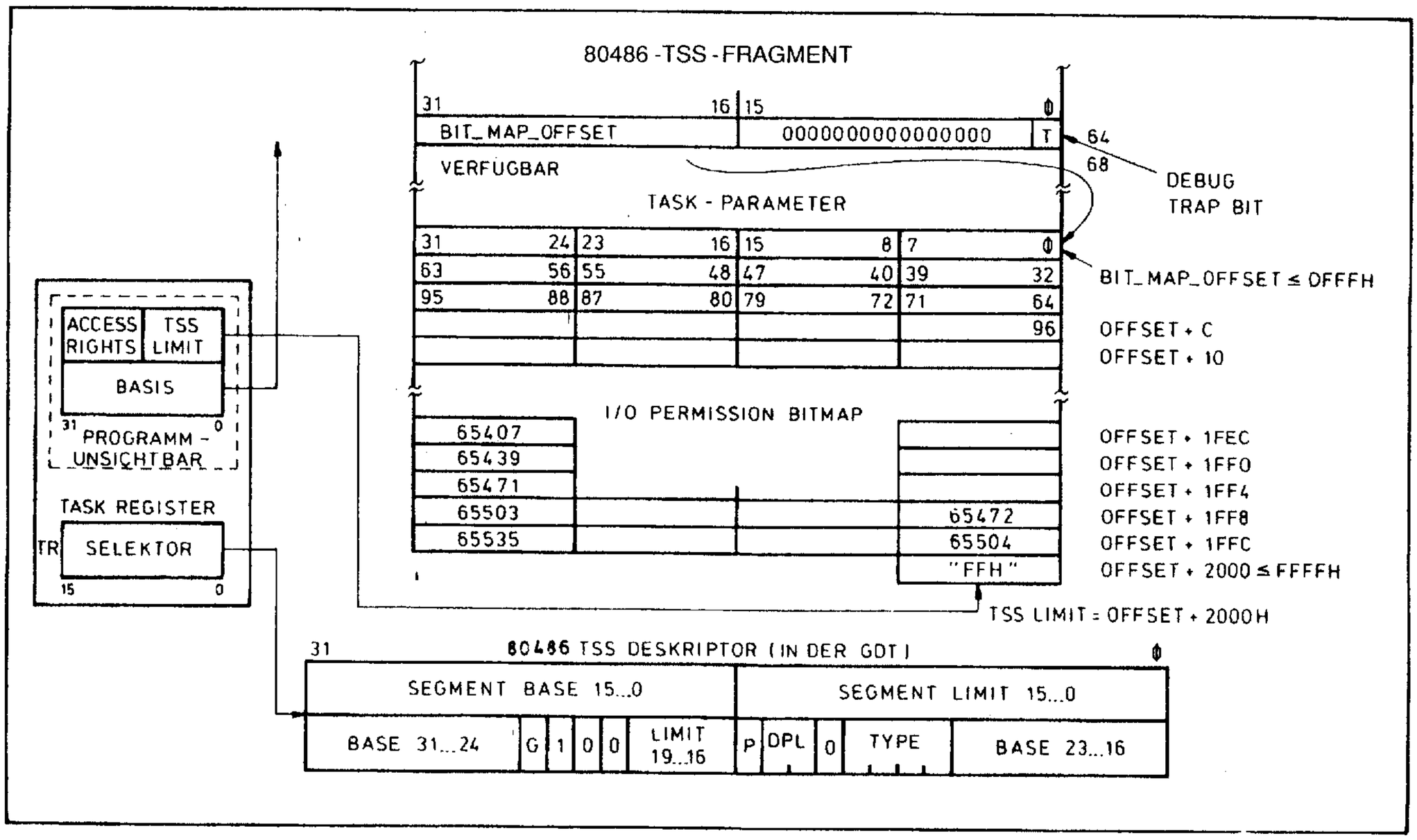
Die gesamte „I/O Permission Bit Map" und das Terminator-Byte FFH müssen bei Offsets liegen, die relativ zur TSS-Basis ***kleiner*** oder ***gleich*** FFFFH sind.

Da ein Bit-Vektor mit einer maximalen Länge von 64 KBits einen Umfang von 8 KBytes hat, liegen die Offsets dieser Bytes im Bereich 0 ... 1FFFH. Das Terminator-Byte am Ende der „Bit Map" liegt dann bei Offset 2000H.

Damit nun der maximale Bit-Vektor mit 64 KBits im Task-Status-Segment untergebracht werden kann, darf der 16 Bit-Pointer BIT_MAP_OFFSET den Wert DFFFH = FFFFH – 2000H ***nicht*** überschreiten:

(BIT_MAP_OFFSET ≤ DFFFH).

Dies verdeutlicht das folgende Bild:

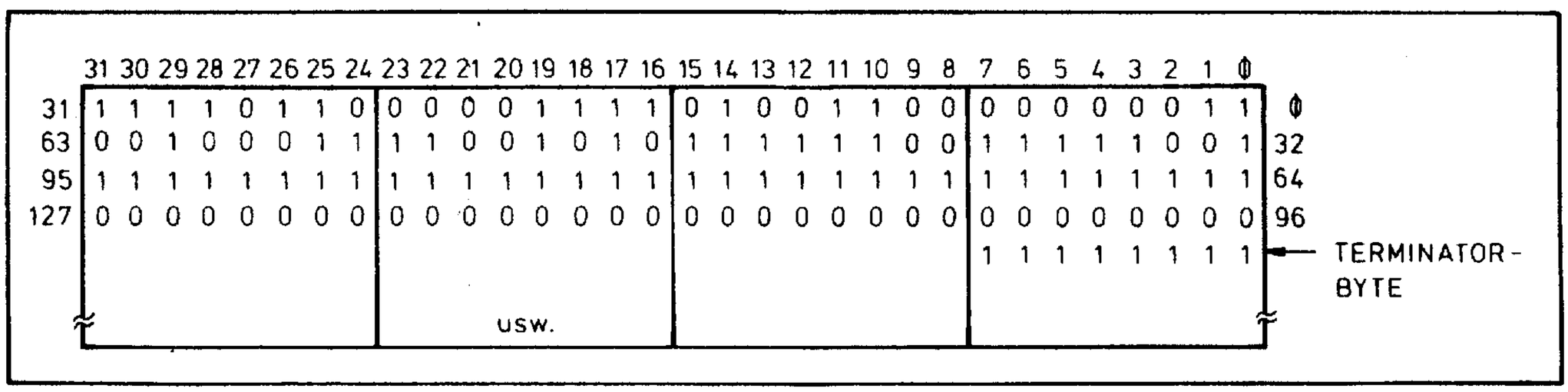


Jedes Bit in der „I/O Permission Bit Map" korrespondiert mit einer I/O-Port-***Byte***-Adresse. Hat z. B. der Port die Adresse 36, prüft der Prozessor beim Zugriff auf diesen Port das Bit 36 in der „I/O Permission Bit Map".

- Hat das der Port-Nummer zugeordnete Bit den Wert ***0***, ist der Port-Zugriff ***erlaubt*** und der I/O-Befehl kann ausgeführt werden.
- Hat das der Port-Nummer zugeordnete Bit den Wert ***1***, ist der Port-Zugriff ***nicht erlaubt***. Die Ausführung des I/O-Befehls bewirkt in diesem Fall eine Ausnahmesituation, die zur Programm-Unterbrechung führt.

Das folgende Bild zeigt das Fragment einer „I/O-Permission Bit Map" und illustriert erfolgreiche und erfolglose Port-Referenzen:

I/O PERMISSION BIT MAP:



Port-Nummern der adressierbaren I/O-Ports	Port-Nummern der nicht-adressierbaren I/O-Ports
2 bis 9	0, 1
12, 13, 15	10, 11, 14
20 bis 24	16 bis 19
27, 33, 34, 40, 41, 48, 50, 52, 53	25, 26
58 bis 60	28 bis 32
62, 63	35 bis 39
96 bis 127	42 bis 47
	49, 51
	54 bis 57
	61
	64 bis 95

In der 32-Bit-Umgebung des 80486 können sogenannte Byte-Interfaces (z. B. parallele Ein-/Ausgabe-Ports 8255) als Word-Interfaces (2 x 8255) oder als Doubleword-Interfaces (4 x 8255) konfiguriert werden.

In allen diesen Fällen korrespondieren in der „I/O-Permission Bit Map" 2 benachbarte Bits mit jeweils einem Word-Interface bzw. 4 benachbarte Bits mit jeweils einem Doubleword-Interface. Nur dann, wenn alle referenzierten Bits 0 sind, ist der I/O-Zugriff erlaubt. Sollte aber eines dieser Bits 1 sein, führt der versuchte I/O-Zugriff zur Ausnahmesituation.

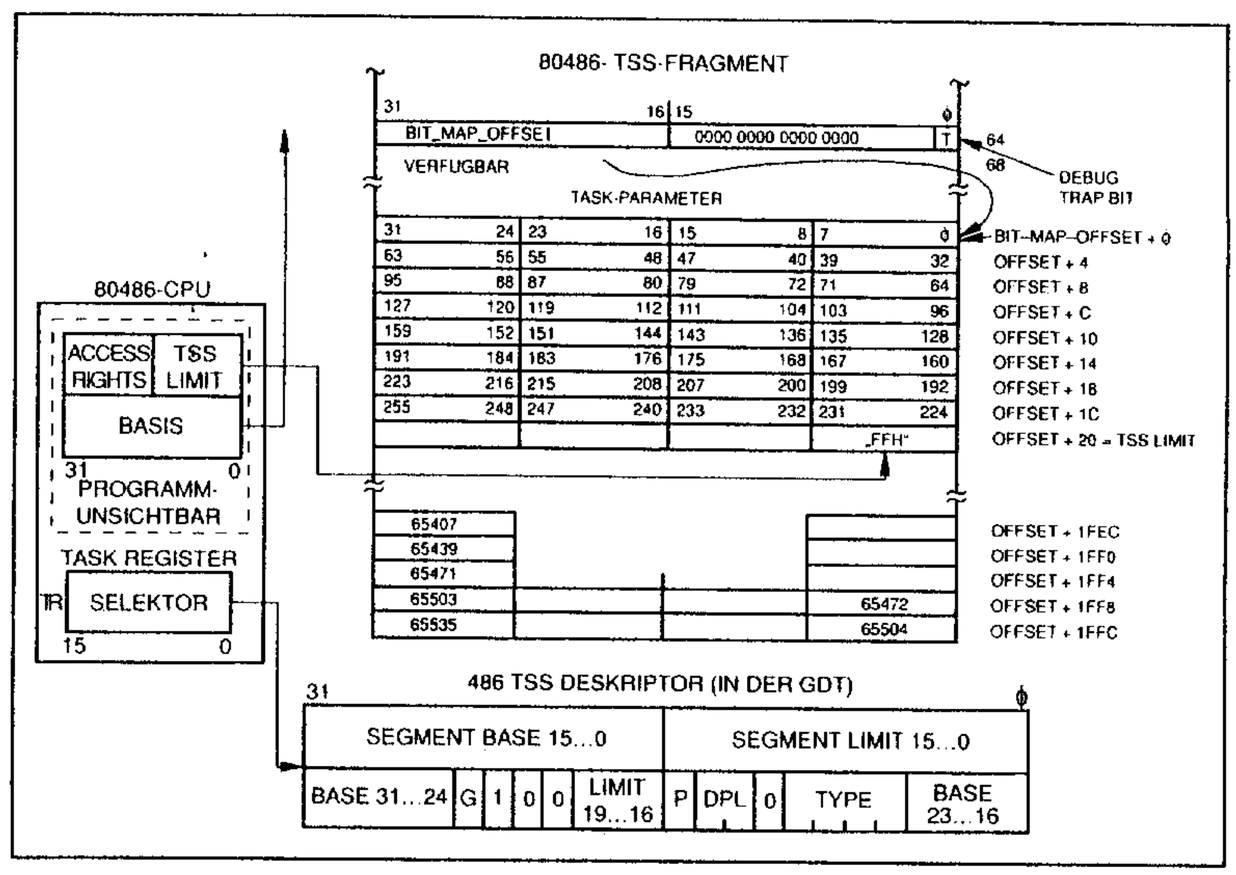
Für die „I/O-Permission Bit Map" ist es ***nicht*** erforderlich, daß sie alle I/O-Adressen repräsentiert. Wenn das System nur einen kleinen Teil des 64K-I/O-Raums beansprucht, muß in der „Permission Map" die Anzahl der korrespondierenden Bits entsprechend verringert werden,

Um dies zu realisieren, ist lediglich die Obergrenze des Task-Status-Segments durch einen geeigneten Wert im Limit-Feld des TSS-Deskriptors an die aktuellen Erfordernisse anzupassen.

BEISPIEL:
In einem 80486-I/O-System soll nur der Zugriff auf die ersten 256 I/O-Ports (0 ... 255) erlaubt werden.

In diesem Fall besteht der korrespondierende Bit-String in der „Permission Map" aus 32 Bytes, deren Offset-Bereich zwischen BIT_MAP_OFFSET + 0 und BIT_MAP_OFFSET + 1 FH (31) liegt.

Da jede „I/O-Permission Bit Map" mit dem Terminator-Byte FFH abgeschlossen ist, muß das Limit-Feld mit dem Wert BIT_MAP_OFFSET + 1 FH + 1 = BIT_MAP_OFFSET + 20H geladen werden. Dies verdeutlicht das folgende Bild:



Jeder I/O-Zugriff, der bei dieser Konfiguration auf die Ports 256 ... 65535 versucht wird, führt zwangsläufig zur Programm-Unterbrechung,. da die „Permission Bits" in diesem Bereich als 1-Bits behandelt werden.

> **HINWEIS:** Wenn der BIT_MAP_OFFSET ***größer*** oder ***gleich*** dem Limit-Wert im TSS-Deskriptor ist, hat das 80486-Task-Status-Segment keine „I/O- Permission Bit Map".

Um eine sogenannte „***Null*** Permission Map" zu definieren, ist das BIT_MAP_OFFSET-Feld mit FFFFH zu belegen. In diesem Fall ist dann ***kein*** Terminator-Byte erforderlich.

4.13.1 Adressierung individueller Bits

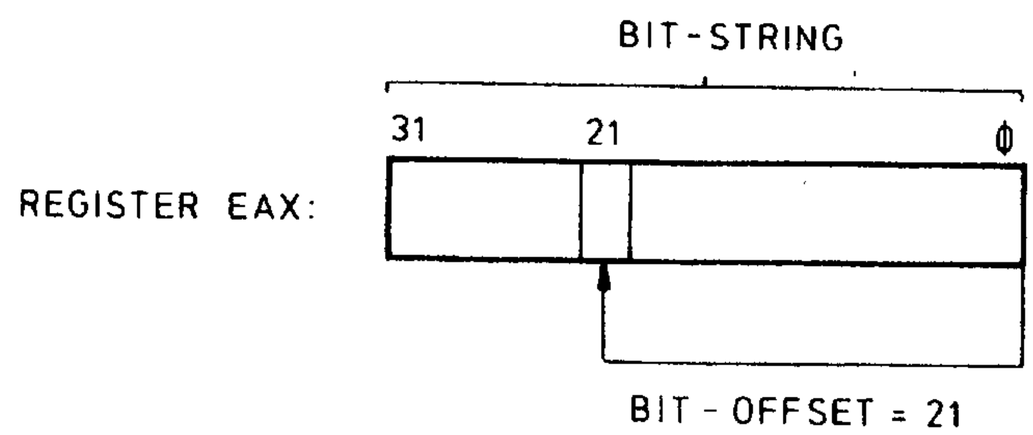
Ein spezieller Satz von 80486-Befehlen operiert mit Bit-Strings. Mit diesen Befehlen können individuelle Bits erreicht und z. B. zur Manipulation der „I/O-Permission Bit Map" benützt werden. Diese Befehle sind BT (Test Bit), BTS (Test Bit and Set), BTR (Test Bit and Reset) und BTC (Test Bit and Complement).

Ein Bit-String kann sich entweder in einem allgemeinen 16- bzw. 32-Bit-Register oder in einem 16- bzw. 32-Bit-Speicherplatz aufhalten. Dabei sind die einzelnen Bit von rechts nach links durchnumeriert, Für die Adressierung eines bestimmten Bits innerhalb eines Bit-Strings gilt folgende Syntax:

BIT - BASIS , BIT - OFFSET

Dabei kennzeichnet der Ausdruck Bis-Basis den Aufenthaltsort des Bit-Strings (Register oder Speicher).

- Wenn die Bit-Basis ein allgemeines Register ist, gibt der Bit-Offset die Position eines bestimmten Bits innerhalb des spezifizierten Registers an. Dabei ***muß*** der Bit-Offset im Bereich 0 ... 31 liegen. Dies verdeutlicht das folgende Bild:



Hier hält sich der Bit-String im Register EAX auf, so daß z. B. mit EAX,21 das Bit 21 erreicht werden kann.

CODIERBEISPIEL:

```
BT      EAX,21      ;Teste Bit
```

Dieser Befehl adressiert das Bit 21 in EAX und überträgt seinen Wert in das ***Carry***-Flag.

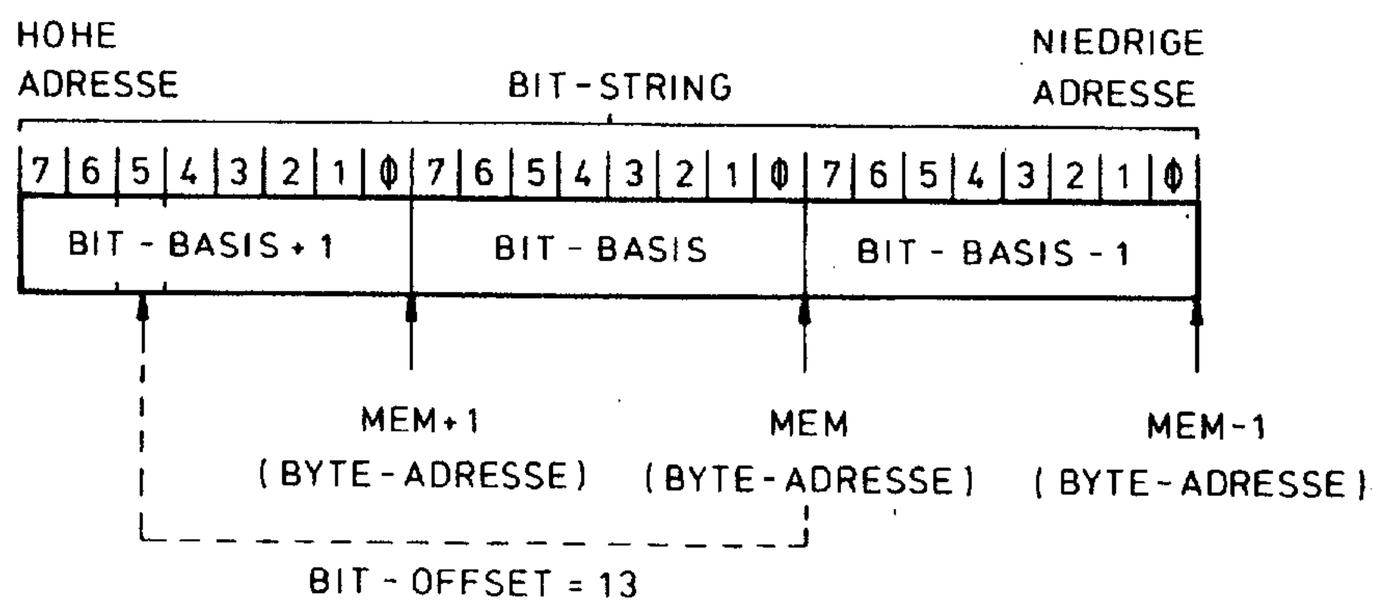
Jeder Bit-String im Speicher ist in ***Bytes*** unterteilt. Dies bedeutet, daß sich jedes Bit in irgendeinem Byte innerhalb des Bit-Strings aufhält.

Um nun ein bestimmtes Bit im Speicher-Bit-String zu erreichen, muß zunächst als ***Bit-Basis*** die Adresse des Bytes spezifiziert werden, in dem das Bit enthalten ist.

Beginnend bei dieser Byte-Position kann dann der Bit-Offset im Bereich–2 Gigabits ... + 2 Gigabits liegen.

Dies verdeutlichen die folgenden Bilder:

1. Positiver Offset



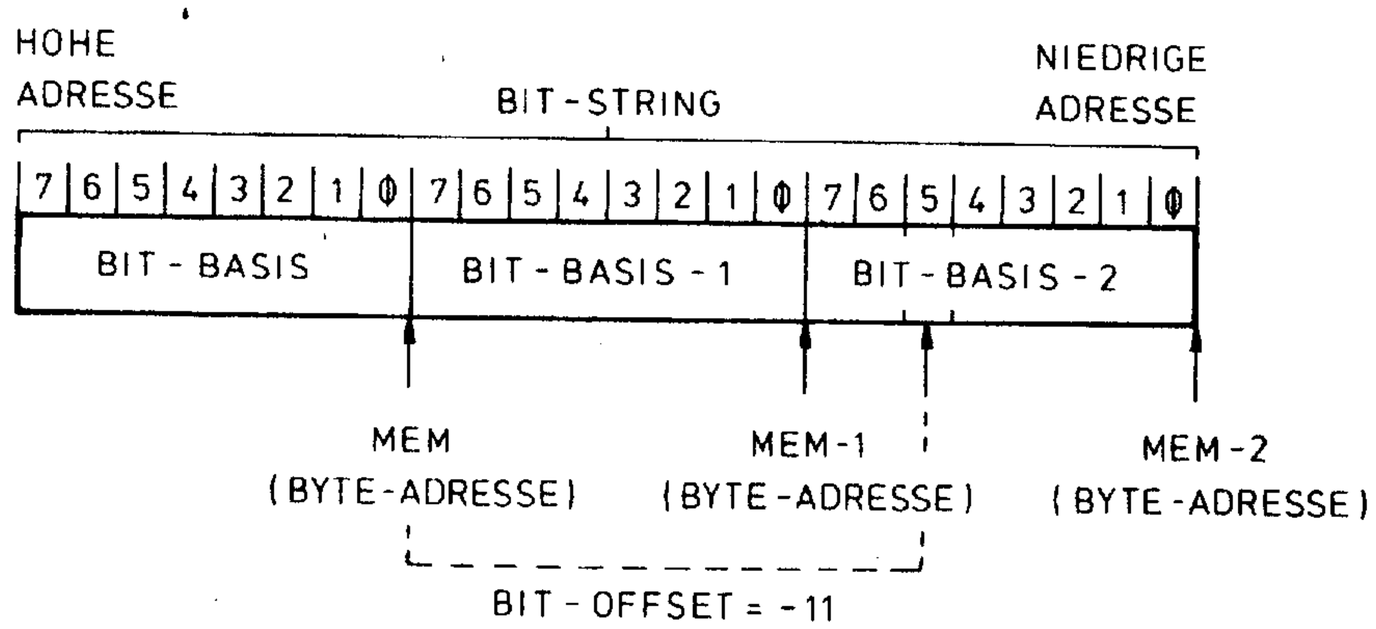
Hier hält sich der Bit-String in einem 3-Byte-Feld auf, so daß z.B. mit MEM,13 das Bit 13 links von MEM erreicht werden kann. Dieses Bit ist identisch mit dem Bit 5 im Byte MEM+1.

CODIERBEISPIEL:

```
BTC     MEM,13      ;Teste Bit und komplementiere
```

Dieser Befehl adressiert Bit 13 links von MEM, rettet es in das Carry-Flag und komplementiert anschließend seinen Wert.

2. Negativer Offset



Auch hier hält sich der Bit-String in einem 3-Byte-Feld auf, so daß z. B. mit MEM,–11 das Bit 11 rechts von MEM erreicht werden kann. Dieses Bit ist identisch mit dem Bit 5 im Byte MEM–2.

CODIERBEISPIELE:

```
BTS     MEM,–11     ;Teste und setze Bit
BTR     MEM,–11     ;Teste und lösche Bit
```

Beide Befehle adressieren Bit 11 rechts von MEM und retten es in das Carry-Flag. Während BTS dieses Bit anschließend setzt, wird es von BTR gelöscht.

4.13.2 Anwendung der ASM386/486-STRUC- und DBIT-Direktiven

Um in ASM386/486 einen Bit-Strang zu definieren, können die Direktiven ***DBIT*** und ***STRUC*** benützt werden. Dabei initialisiert die DBIT-Direktive eine Einzel-Bit-Variable vom Typ ***BIT***. Werden mehrere solche Bit-Variablen in einer Struktur untergebracht, entsteht ein zusammenhängendes Bit-Feld, das sich über viele Byte-Grenzen hinweg fortsetzen kann:

Das folgende ASM386/486-Programm-Feld zeigt, wie ein solcher Bit-String vereinbart werden kann:

```
DATEN                    SEGMENT RW USE32
                 ORG     1000H
BIT_STRING               STRUC
BIT_0                    DBIT          OB
BIT_1                    DBIT          OB
BIT_2                    DBIT          OB
BIT_3                    DBIT          OB
BIT_4                    DBIT          OB
BIT_5                    DBIT          OB
BIT_6                    DBIT          OB
BIT_7                    DBIT          OB
BIT_8                    DBIT          OB
BIT_9                    DBIT          OB
BIT_STRING               ENDS
BIT_STRING_VAR           BIT_STRING <>
DATEN                    ENDS
```

In diesem Beispiel wird die STRUC-Direktive benützt, um das „Rahmenwerk" für die Daten-Struktur BIT_STRING festzulegen. Diese Daten-Struktur enthält als Fundamental-Variablen 10 Einzel-Bit-Variablen (BIT_0 bis BIT_9), die mit DBIT definiert worden sind. Dabei hat jedes dieser Einzel-Bits den Wert 0.

Damit nun der Bit-Strang manipuliert werden kann, muß er sich in einer Variablen aufhalten. Aus diesem Grund wird mit

- BIT_STRING_VAR BIT_STRING < >

der Struktur-Variablen BIT_STRING_VAR die Daten-Struktur BIT_STRING zugewiesen. Dabei behalten die Einzel-Bit-Variablen wegen < > ihren Wert 0 bei.

Da der Bit-String länger als ein Byte ist, werden 2 Bytes angelegt, wobei sich dann BIT_8 und BIT_9 im zweiten Byte aufhalten. Die nicht-benützten Bit-Positionen sind mit 0 belegt.

4.13.3 Anwendung der ASM386/486-OFFSET- und BITOFFSET-Operatoren

Eine individuelle Einel-Bit-Variable ist erreichbar, wenn die ASM386/486-Operatoren (.), OFFSET und BITOFFSET benützt werden.

BEISPIEL:

```
MOV     EBX,OFFSET BIT_STRING_VAR.BIT_9
```

Dieser Befehl liefert den Offset des Bytes, in dem sich die Einzel-Bit-Variable BIT_9 aufhält. Wegen ORG 1000H liegt dieses Byte bei Offset 1001H relativ zum Beginn des DATEN-Segments (EBX = 1001H).

```
MOV     AX,BITOFFSET BIT_STRING_VAR.BIT_9
```

Dieser Befehl liefert den Bit-Offset von BIT_9 relativ zum Beginn des Bytes, in dem es enthalten ist. Da sich BIT_9 im zweiten Byte der Stuktur-Variablen aufhält, ist sein Bit-Offset 1 (AX = 1).

Soll nun z. B. BIT_9 mit BTS ***gesetzt*** werden, ist folgende Befehls-Sequenz verwendbar:

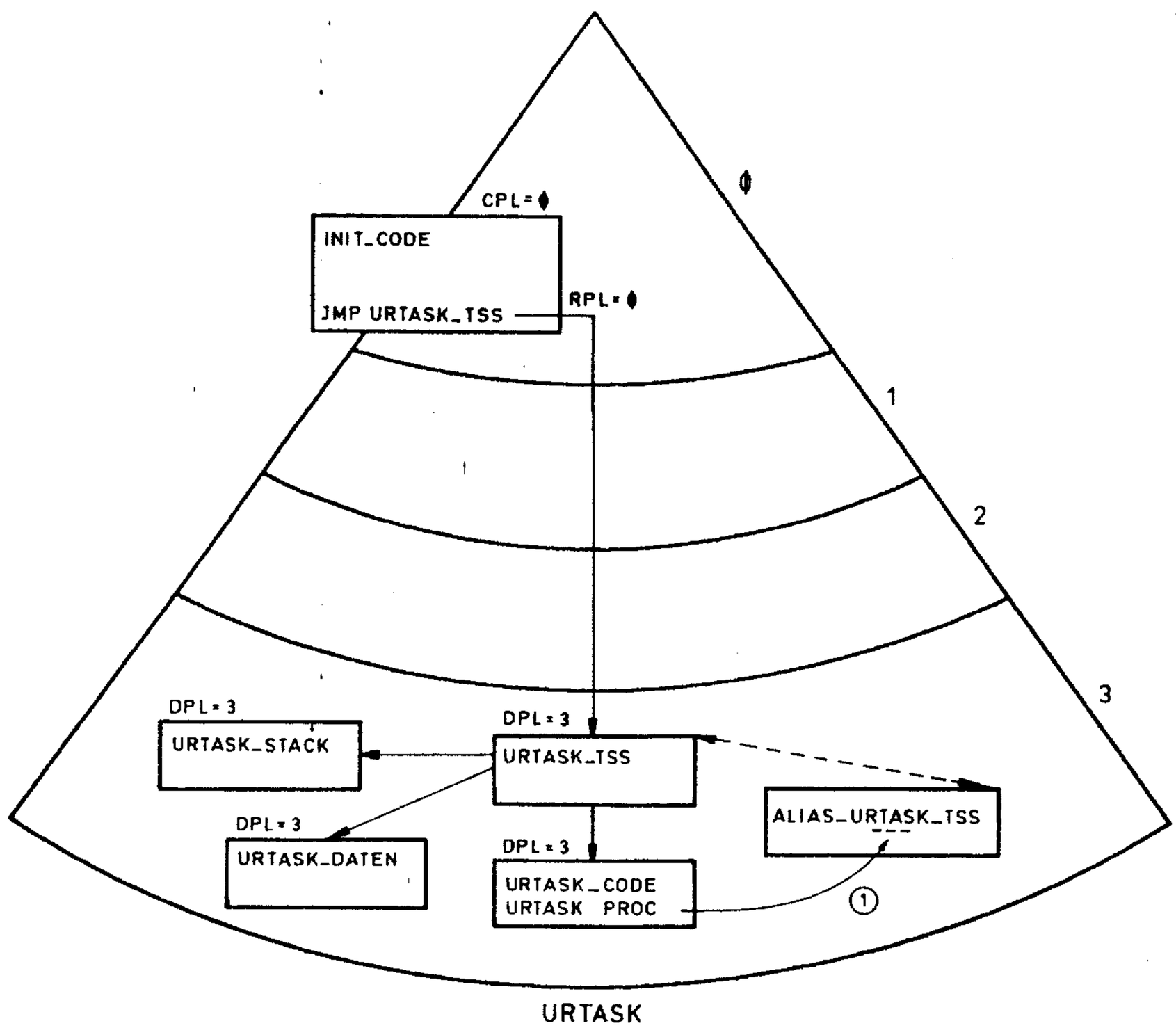
```
MOV     EBX,OFFSET BIT_STRING_VAR.BIT_9
BTS     BYTE PTR [EBX],BITOFFSET BIT_STRING_VAR.BIT_9
```

Dabei zeigt BYTE PTR [EBX] im BTS-Befehl auf das Byte, in dem BIT_9 enthalten ist.

4.13.4 **Programmierbeispiel:** Bit-Manipulation in der „I/O-Permission Bit Map“

Das folgende ASM386/486-Programm benützt die oben beschriebenen ASM386/486-Sprachkonstrukte, um ein individuelles Bit in der „I/O Permission Bit Map“ des Urtask-Status-Segments zu setzen.

Das Programm bezieht sich dabei auf eine Software-Struktur, wie sie durch das angegebene Bild illustriert ist:

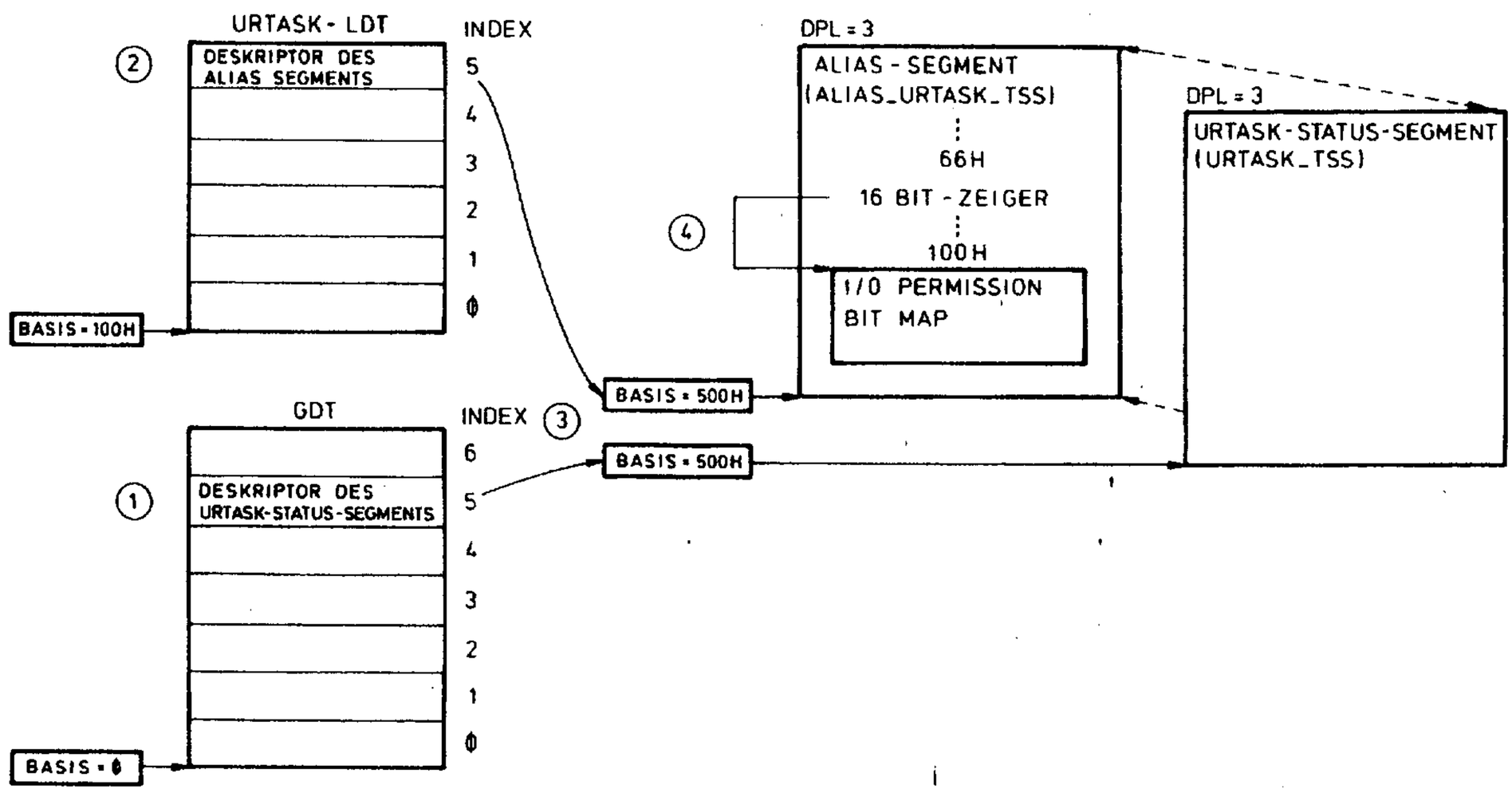


Diese Software-Struktur ist eine geringfügige Erweiterung des System-Programms „Urtask“. Wie das Bild zeigt, enthält die Struktur zusätzlich das ***Alias***-Segment ALIAS_URTASK_TSS für das Urtask-Status-Segment URTASK_TSS. Die Prozedur

URTASK im URTASK_CODE-Segment benützt dieses Alias-Segment, um Eintragungen im Urtask-Status-Segment zu erreichen ①.

Im folgenden Bild ist zu erkennen, daß sich der Deskriptor des Urtask-Status-Segments (URTASK_TSS) in der GDT bei Index 5 befindet ①, während der Deskriptor des Alias-Segments (ALIAS_URTASK_TSS) in der Urtask-LDT bei Index 5 aufbewahrt ist ②.

Sowohl das Urtask-Status-Segment URTASK_TSS, als auch das Alias-Segment ALIAS-URTASK_TSS haben die gemeinsame Basis-Adresse 500H und die gleiche Privileg-Ebene DPL = 3. Die Deskriptoren der beiden Segmente zeigen also auf die gleiche physikalische Position ③.



Im Alias-Segment liegt bei Offset 66H (ORG 66H) der 16 Bit-Zeiger auf die „I/O Permission Bit Map" der Urtask ④. Diese „Bit-Map" ist ab Offset 100H (ORG 1000H) als Bit-String in der Struktur-Variablen PERMISSION_BIT_MAP enthalten.

Dabei setzt sich der Bit-String aus den Einzel-Bits PORT_0 ... PORT_9 zusammen, die mit den I/O-Ports 0 ... 9 korrespondieren und I/O-Zugriffe auf dies Ports zunächst zulassen (Einzel-Bits PORT_0 ... PORT_9 = 0).

Die Prozedur URTASK initialisiert mit

- MOV AX,ALIAS_URTASK_TSS
- MOV ES,AX

Den sichtbaren Teil des ES-Registers mit dem Selektor des Alias-Segments ALIAS_URTASK_TSS.

Dieser Selektor zeigt zur Urtask-LDT, so daß unmittelbar danach der Prozessor den zugehörigen Alias-Deskriptor des Urtask-Status-Segments in das ES-Cache-Register laden kann. Anschließend führt die Prozedur die Befehle

- MOV EBX,OFFSET PERMISSION_BIT_MAP. PORT_9
- BTS ES:BYTE PTR [EBX],BITOFFSET PERMISSION_BIT_MAP. PORT_9

aus. Sie setzen in der „I/O Permission Bit Map“ das Korrespondenz-Bit PORT_9 und ***verhindern*** damit einen möglichen I/O-Zugriff zum zugehörigen I/O-Port 9.

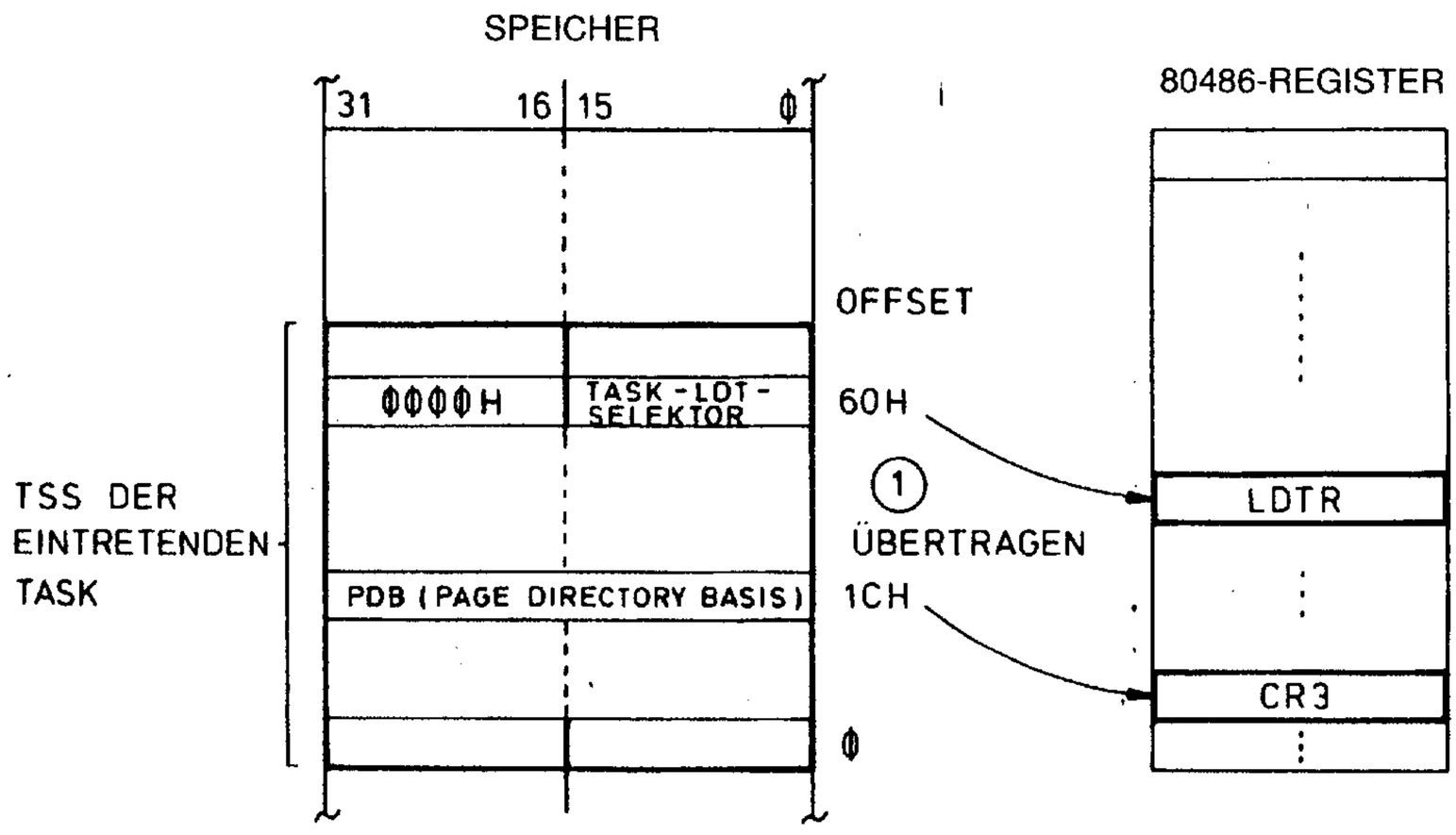
```
LOC  OBJ                LINE  SOURCE

                          1   NAME            B171
                          2
                          3   ;Definiere das Layout fuer einen Deskriptor
                          4
--------                  5   DESK            STRUC
00000000                  6   LIMIT_0_15      DW  0          ;Segment-Obergrenze (Bits 0..15)
00000002                  7   BASIS_0_15      DW  0          ;Basisadresse (Bits 0..15)
00000004                  8   BASIS_16_23     DB  0          ;Basisadresse (Bits 16..23)
00000005                  9   ZUGRIFF         DB  0          ;Access Right Byte
00000006                 10   GRAN            DB  0          ;Flags und Segment-Obergrenze
                         11                                  ;(Bits 16..19)
00000007                 12   BASIS_24_31     DB  0          ;Basisadresse (Bits 24..31)
--------                 13   DESK            ENDS
                         14
  0001                   15   PE              EQU 1          ;Protected Mode freigeben
                         16
                         17   EXTRN URTASK_TSS:FAR
                         18   PUBLIC GDT_PSEUDO_DESK, IDT_PSEUDO_DESK, LDT_PSEUDO_DESK
                         19
    ----                 20   INIT_CODE       SEGMENT ER USE16 ;Ausfuehrbares und lesbares Segment
                         21                                  ;mit Basis-Adresse=FFFF0000H und
                         22                                  ;DPL=0 (Festlegung durch BLD386)
    FFF0                 23                   ORG  0FF0H     ;RESET-Adresse=FFFFFF0H
    FFF0 E9705           24           JMP     NEAR PTR INIT  ;Das CS-Register behaelt den
                         25                                  ;Wert F000H
    0000                 26                   ORG  0         ;Die EPROM-GDT Adresse=FFFF0000H
                         27
    0000                 28   EPROM_START  LABEL  WORD
                         29
    0000 (8              30   GDT_NULL_DESK     DB SIZE DESK DUP(?) ;Luecke und Null Deskriptor
     ??
     )
    0008 (8              31   GDT_ALIAS_DESK    DB SIZE DESK DUP(?) ;Alias-Deskriptor der GDT
     ??
     )
    0010 (8              32   IDT_ALIAS_DESK    DB SIZE DESK DUP(?) ;Alias-Deskriptor der IDT
     ??
     )
    0018 (8              33   URTASK_LDT_DESK  DB SIZE DESK DUP(?) ;Deskriptor der Urtask-LDT
     ??
     )
    0020 (8              34   ABLAGE_TSS_DESK  DB SIZE DESK DUP(?) ;Ablage-TSS-Deskriptor
     ??
     )
    0028 (8              35   URTASK_TSS_DESK  DB SIZE DESK DUP(?) ;Urtask-TSS-Deskriptor
     ??
     )
    0030 (8              36   INIT_CODE_DESK   DB SIZE DESK DUP(?) ;Deskriptor des Initialisierungs-
     ??
     )
                         37                                  ;Segments
                         38
    0038 ???????????     39   GDT_PSEUDO_DESK  DP  ?          ;Reservierung von 6 Bytes fuer den
                         40                                  ;GDT-Pseudo-Deskriptor
    003E ???????????     41   IDT_PSEUDO_DESK  DP  ?          ;Reservierung von 6 Bytes fuer den
                         42                                  ;IDT-Pseudo-Deskriptor
    0044 ???????????     43   LDT_PSEUDO_DESK  DB  ?          ;Reservierung von 6 Bytes fuer den
                         44                                  ;LDT-Pseudo-Deskriptor
                         45
```

```
0100                    46          ORG     100H                ;Die EPROM-LDT Adresse=FFFF0100H
                        47
0100 (8                 48  LDT-NULL_DESK   DB SIZE DESK DUP(?) ;Luecke und Null Deskriptor
??
)
0108 (8                 49  LDT_ALIAS_DESK  DB SIZE DESK DUP(?) ;Alias-Deskriptor der LDT
??
)
0110 (8                 50  URTASK_CODE_DESK DB SIZE DESK DUP(?) ;Deskriptor des Urtask-Codesegments
)
0118 (8                 51  URTASK_DATEN_DESK DB SIZE DESK DUP(?) ;Deskriptor des Urtask-Datensegments
??
)
0120 (8                 52  URTASK_STACK_DESK DB SIZE DESK DUP(?) ;Deskriptor des Urtask-Stacksegments
??
)
0128 (8                 53  ALIAS_URTASK_DESK DB SIZE DESK DUP(?) ;Alias-Deskriptor des Urtask-Status-
??
)
                        54                                      ;Segments
                        55
0200                    56              ORG  200H               ;Die EPROM-IDT  Adresse=FFFF0200H
                        57
0200 (256               58  INTERRUPT_DESK  DB 32*SIZE DESK DUP (?) ;Deskriptoren fuer 32 Interrupt-
??
)
                        59                                      ;Prozeduren
0400                    60              ORG  400H               ;Die Adresse des EPROM-Ablage-
                        61                                      ;Status-Segments=FFFF0400H
0400 26                 62              DD   26 DUP(?)          ;TSS enthaelt 26 Doppelworte
????????
)
                        63
0500                    64          ORG     500H                ;Die Adresse des EPROM-Urtask-
                        65                                      ;Status-Segments=FFFF0500H
0500 (26                66              DD   26 DUP(?)          ;TSS enthaelt 26 Doppelworte
????????
)
                        67
0568                    68  EPROM_ENDE  LABEL  WORD
                        69
0568                    70  INIT:       ;EPROM-GDT, EPROM-IDT, EPROM-Pseudo-Deskriptoren,
                        71              ;EPROM-Ablage-TSS und EPROM-Urtask-TSS in den RAM kopieren
                        72              ;Basisadressen der RAM-GDT und RAM-IDT festlegen
                        73              ;In den Protected Mode umschalten
                        74              ;Ablage-TSS-Deskriptor laden
                        75
0568 66EA00000000- - - -E76             JMP     URTASK_TSS          ;Sprung in die Urtask ueber einen
                        77                                          ;TSS_Deskriptor
                        78
- - - -                 79  INIT_CODE           ENDS
                        80
- - - - - - - -         81  URTASK_CODE     SEGMENT EO  USE32   ;Nur auffuehrbares Segment mit
                        82                                      ;Basis-Adresse=1000H und DPL=3
                        83                                      ;(Festlegung durch BLD386)
00000000                84  URTASK PROC     FAR
                        85
                        86  PUBLIC START
00000000 66B8000090- - - R87 START:  MOV     AX,ALIAS_URTASK_TSS ;AX=Selektor des Urtask-TSS-
                        88                                      ;   Alias-Segments
00000007 8EC0           89          MOV     ES,AX               ;ES_Deskriptor des TSS-Alias-
                        90                                      ;   Segments
00000009 BB1010000      R91         MOV     EBX,OFFSET PERMISSION_BIT_MAP.PORT_9
```

```
                                                                    ;EBX=Byte-Adresse von Permission-
                                                                    ;Bit PORT_9
0000000E 260FBA2B019090909090           BTS     ES:BYTE PTR [EBX],BITOFFSET PERMISSION_BIT_MAP.PORT_9
                                                                    ;Zugriff auf Port 9 sperren

00000018 EBFE           SCHL:   JMP     SCHL
0000001A                URTASK ENDP

--------                URTASK_CODE     ENDS

--------                URTASK_DATEN    SEGMENT RW  USE32 ;Schreib- und lesbares Segment mit
                                                          ;Basis-Adresse=2000H und DPL=3
                                                          ;(Festlegung durch BLD386)
                                ;Segment enthaelt irgendwelche
                                ;Urtask-Daten

--------                URTASK_DATEN    ENDS

--------                URTASK_STACK    STACKSEG  10      ;Stack-Segment mit einer Laenge von
                                                          ;10 Bytes. Basis-Adresse=3000H und
                                                          ;DPL=3 (Festlegung durch BLD386)

--------                ALIAS_URTASK_TSS  SEGMENT RW  USE32 ;Schreib- und lesbares Alias-Segment
                                                          ;mit Basis-Adresse=500H und DPL=3
                                                          ;(Festlegung durch BLD386)
                                ;Dieser Raum enthaelt
                                ;Urtask-Status-Informationen

00000066                        ORG     66H               ;Lage des 16 Bit-Zeigers zur
                                                          ;I/O Permission Bit Map
00000066 00010000       R       BIT_MAP_OFFSET DD WORD PTR PERMISSION_BIT_MAP
                                                          ;Das Low-Word von BIT_MAP_OFFSET
                                                          ;enthaelt den 16 Bit-Zeiger
                                                          ;zur I/O Permission Bit Map

                                ;Dieser Raum enthaelt
                                ;Urtask-Parameter

00000100                        ORG     100H              ;Beginn der I/O Permission Bit Map

--------                BIT_MAP STRUC                     ;Vereinbarung der Datenstruktur
                                                          ;BIT_MAP
00000000                PORT_0 DBIT   0B
00000000                PORT_1 DBIT   0B
00000000                PORT_2 DBIT   0B
00000000                PORT_3 DBIT   0B
00000000                PORT_4 DBIT   0B
00000000                PORT_5 DBIT   0B
00000000                PORT_6 DBIT   0B
00000000                PORT_7 DBIT   0B
00000001                PORT_8 DBIT   0B
00000001                PORT_9 DBIT   0B

--------                BIT_MAP ENDS

000001000 0000          PERMISSION_BIT_MAP      BIT_MAP <>  ;Die Struktur-Variable PERMISSION_BIT_
                                                            ;MAP enthaelt die I/O Permission Map
00000102 FF                             DB      0FFH        ;Terminator-Byte

--------                ALIAS_URTASK_TSS ENDS

                                END CS:INIT,DS:URTASK_DATEN,SS:URTASK_STACK
ASSEMBLY COMPLETE, NO WARNINGS, NO ERRORS.
```

4.14 Multitasking in page-orientierten Systemen

Wenn der Prozessor die augenblickliche aktive Task beendet, um anschließend eine neue Task zu starten, überträgt er zunächst den augenblicklichen CPU-Status in das Task-Status-Segment der ausscheidenden Task und initialisiert anschließend die CPU-Register mit bestimmten Task-Status-Segment-Feldern der eintretenden Task.



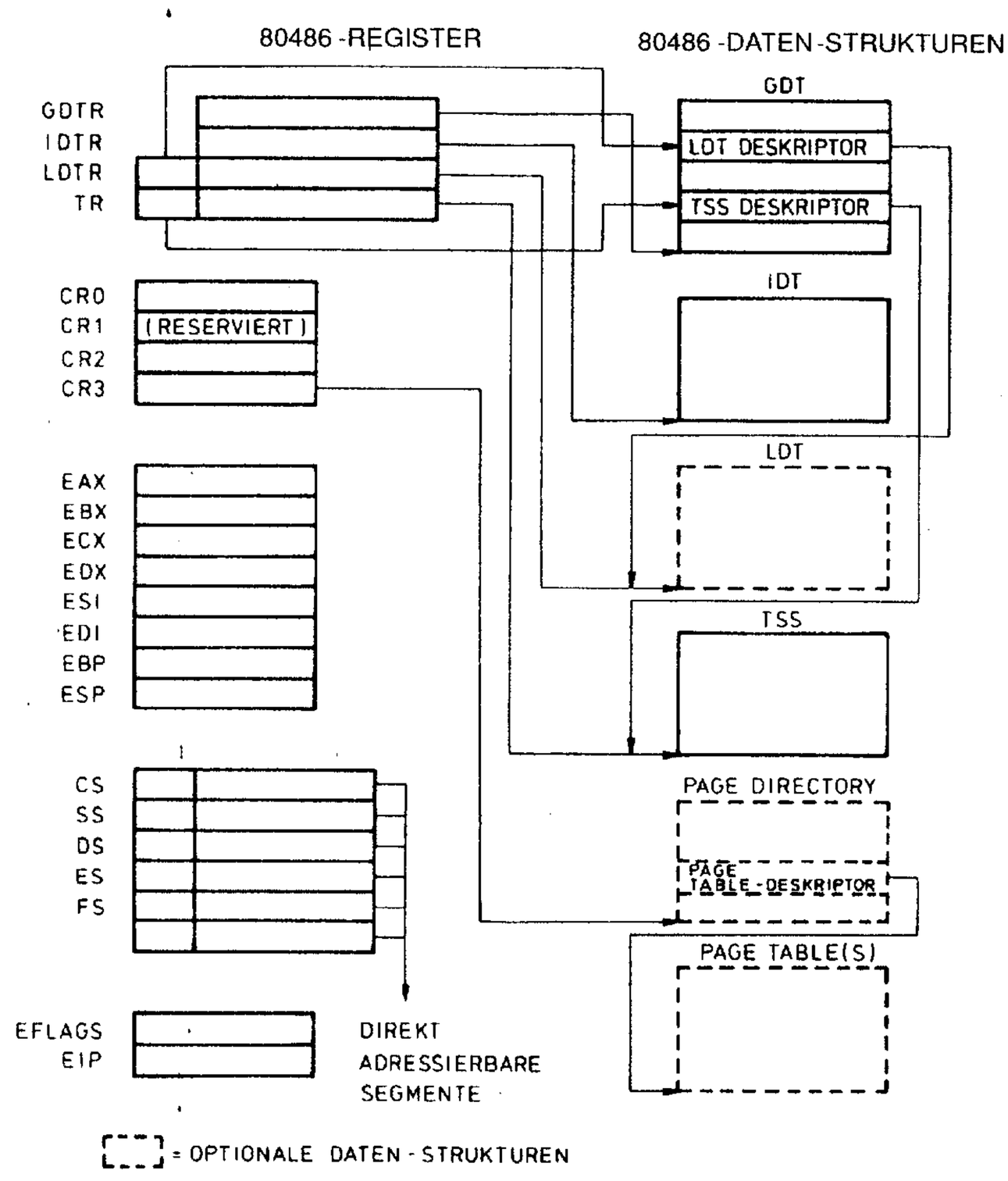
Zu den Feldern, die dabei in die CPU-Register übertragen werden, können ***optional*** der Task-LDT-Selektor und die Page-Directory-Basis gehören. Werden sie benützt, initialisiert das LDT-Selektor-Feld das LDT-Register und PDB-Feld das CR3-Register der CPU ①.

Beide Felder geben dem System-Entwickler ein hohes Maß an Flexibilität im Gebrauch der Segment- und Paging-Leistungsmerkmale des 80486.

Dabei bestimmt das für jede Task gewählte Arrangement,

- ob sich mehrere Tasks ihre Adreßräume teilen,
- ob die Adreßräume der Tasks streng voneinander isoliert sind oder
- ob die Tasks irgendeinen Grand in der Nutzung gemeinsamer Adreßräume zwischen diesen beiden Extremen haben.

Wie die komplette Ausführungs-Umgebung einer Task aussieht, wenn sie auch die optionalen Datenstrukturen wie LDT, „Page Directory“ und „Page Tables“ benützt, illustriert das folgende Bild:



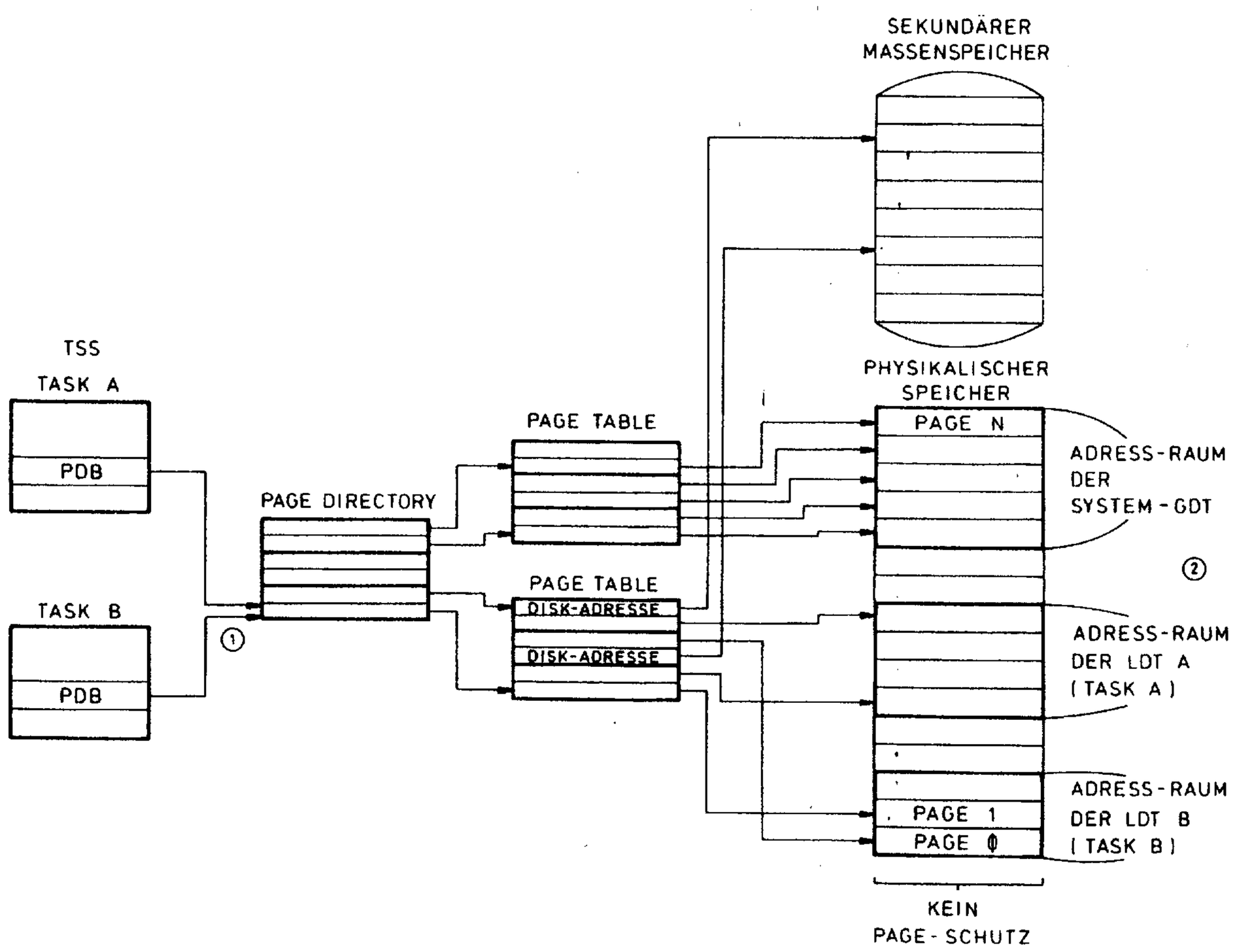
Der System-Entwickler kann für die Abbildung des linearen Adreßraums auf den physikalischen Speicher zwischen verschiedenen Arrangements wählen, die in zwei allgemeine Klassen eingeteilt sind:

- Abbildungs-Arrangement, das von allen „Tasks" gemeinsam benützt wird.
- Abbildungs-Arrangement, das einige Teile der linearen Adreßräume aller Tasks auf die gleichen physikalischen Adressen abbildet.

Die ***erste*** Art der Implementierung ist typisch in segment-orientierten Systemen zu finden. Hier werden die Zugriffsrechte ausschließlich über den Segment-Schutz geregelt und die Adreßräume der Tasks durch die System-GDT und durch ihre LDT's definiert.

Wenn in einem segment-orientierten System auch der Paging-Mechanismus freigegeben ist, kann ein von Pages unterstützter virtueller Speicher implementiert werden. In diesem Fall sind dann beschreibbare „User Pages" (kein Page-Schutz) zu definieren, die lediglich zum Ein- und Auslagern dienen.

Wie das folgende Bild am Beispiel von zwei Tasks zeigt, benützen bei einem solchen Arrangement alle Tasks die ***gleiche*** Page Directory-Tabelle.



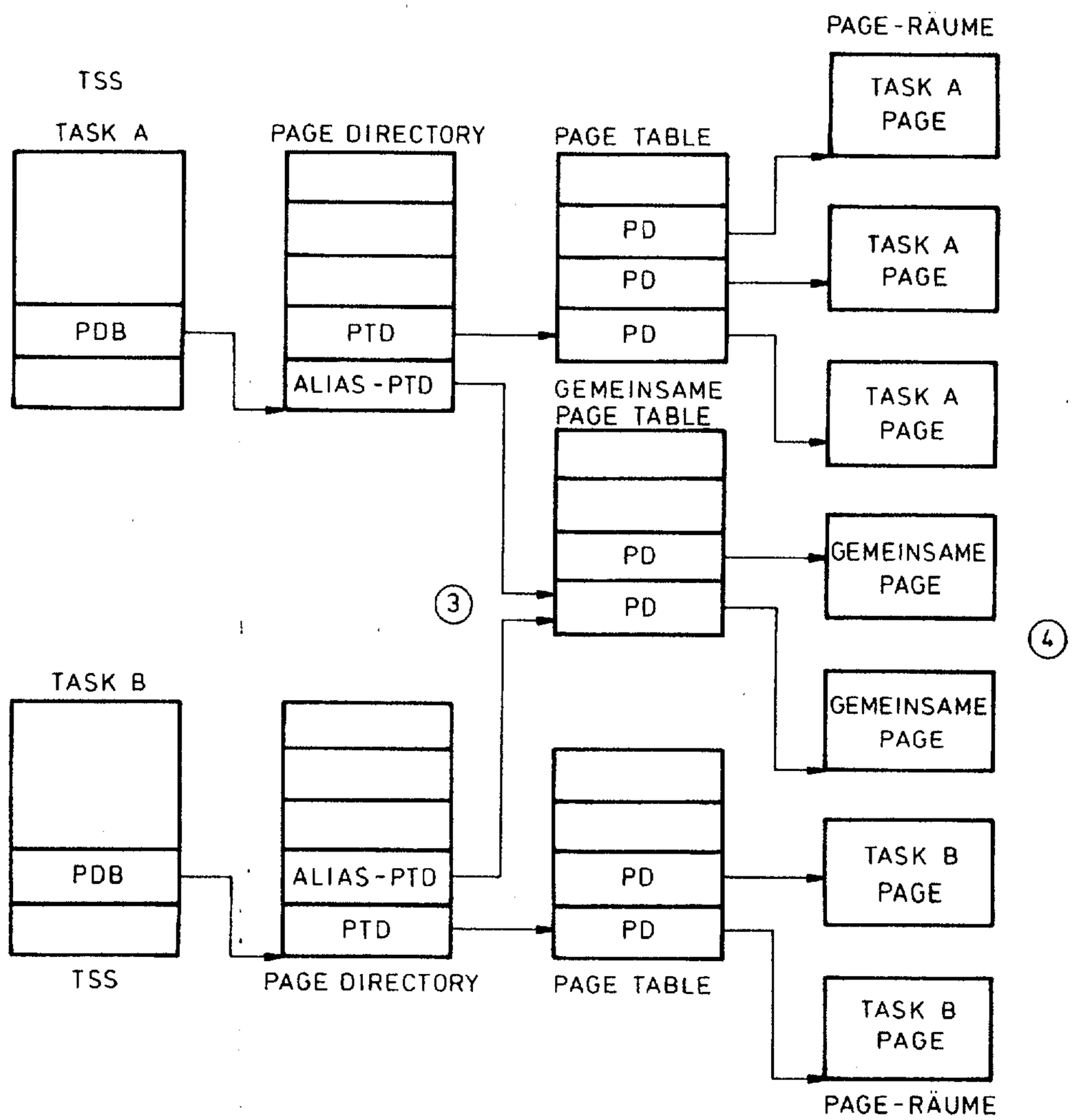
Dies bedeutet, daß alle Page Directory-Basis-Felder (PDB's) in den Task-Status-Segmenten der Tasks den gleichen Adreßwert enthalten ①. Bei einem Task-Wechsel wird daher das CR3-Register ***nicht*** umgeladen.

Da alle Tasks die gleiche „Page Directory" benützen, teilen sie sich auch ihre Pages. Da aber Paging unterhalb der Segmentierung definiert ist, haben Tasks, die sich eine „Page Directory" teilen, nur Zugriff auf Adressen, die durch die System-GDT und die LDT's der Tasks definiert sind ②.

Bei der ***zweiten*** Art der Implementierung wird für jede Task eine eigene Page-Directory-Tabelle benützt. Dies ist möglich, weil das PDB-Feld („Page Directory"-Basis) im Task-Status-Segment jeder Task einen individuellen Adreßwert enthalten kann.

In der ***Theorie*** werden die linearen Adreßräume der verschiedenen Tasks auf komplett unterschiedliche physikalische Adressen abgebildet. Dies ist arrangierbar, wenn die Page-Table-Deskriptoren in den verschiedenen Page-Directory-Tabellen auf unterschiedliche „Page Tables" und die Page-Deskriptoren auf unterschiedliche Anwender-Pages im physikalischen Speicher zeigen. Bei einem solchen Arrangement haben die Tasks ***keine gemeinsamen*** physikalischen Adressen.

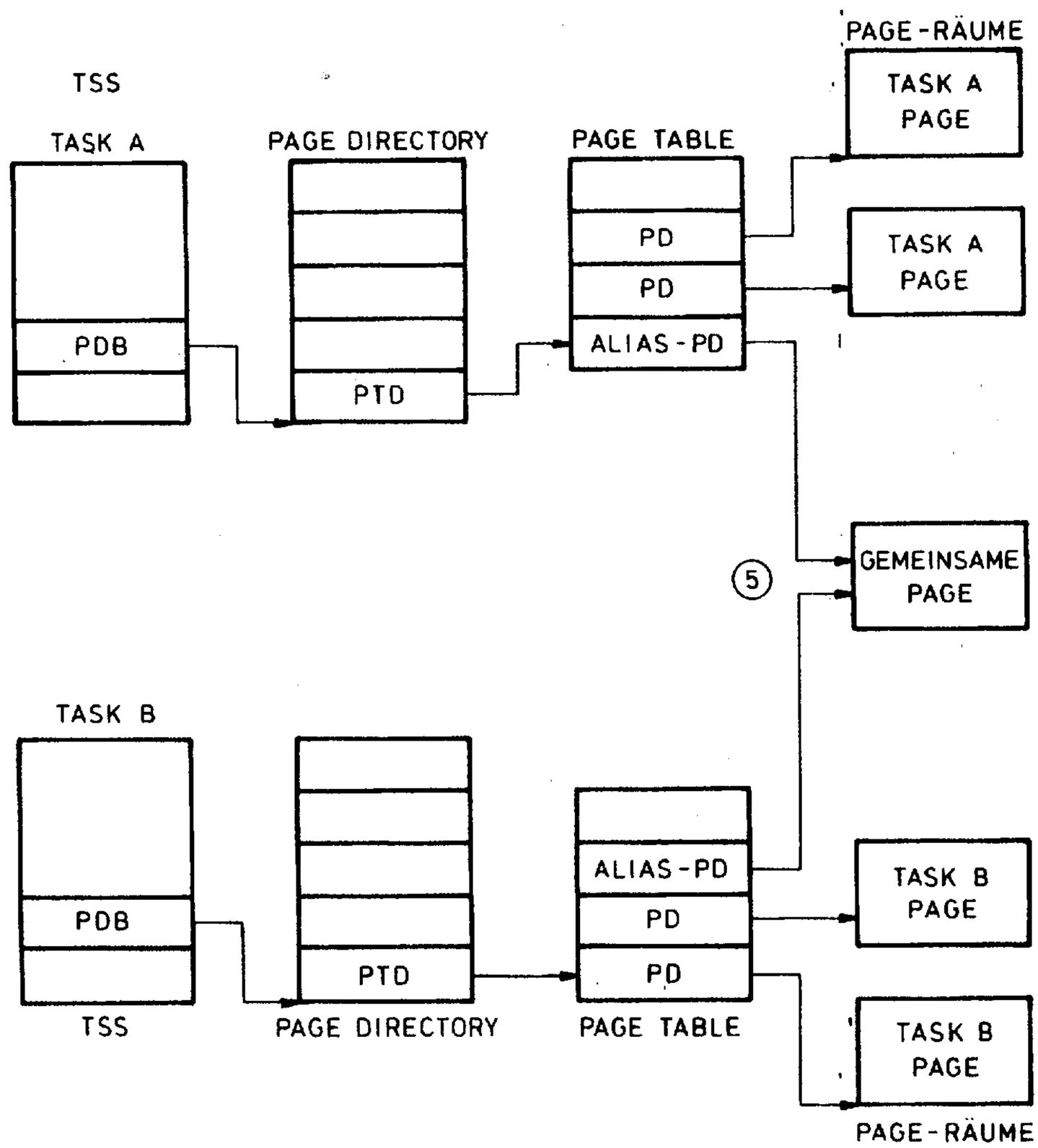
In der ***Praxis*** müssen aber einige Teile der linearen Adreßräume aller Tasks auf die ***gleichen*** physikalischen Adressen abgebildet werden. Wie dies zu realisieren ist, illustriert das folgende Bild am Beispiel von zwei Tasks:



PDB: Page Directory-Basis
PTD: Page Table-Deskriptor
PD: Page-Deskriptor

Es ist zu erkennen, daß jede Task ihre eigene Page-Directory-Tabelle benützt. Sie enthalten sogenannte ***Alias-Page-Table-Deskriptoren,*** die beide auf die gleiche und damit gemeinsame „Page Table" zeigen ③. Auf diese Weise wird ein Teil des linearen Adreßraums beider Tasks auf die gemeinsamen Pages und damit auf die gleichen physikalischen Adressen abgebildet ④.

Eine andere Möglichkeit für Tasks, sich gemeinsame physikalische Adressen zu teilen, wird von sogenannten ***Alias-Page-Deskriptoren*** unterstützt.



PDB: Page Directory-Basis
PTD: Page Table-Deskriptor
PD: Page-Deskriptor

Wie das Bild illustriert, benützt jede Task ihre eigene Page-Directory-Tabelle und ihre eigene „Page Table". Dabei zeigen die in den „Page Tables" enthaltenen Alias-Page-Deskriptoren auf die gemeinsame Anwender-Page ⑤.

> **HINWEIS:** Ein Page- oder Page-Table-Deskriptor kann vom Betriebssystem durch einen frei wählbaren Code im AVAIL-Feld als Alias-Deskriptor gekennzeichnet werden.

Im allgemeinen sind Page-Attribute unbeständiger als Page-Table-Attribute. Daher ist die Benützung gemeinsamer Page-Tables attraktiver. Die Differenz in der Unbeständigkeit der Page- und der Page-Table-Attribute ist besonders in virtuellen Speichersystemen relevant.

Immer dann, wenn der „Swapper" eine Page oder eine „Page Table" ein- bzw. auslagert, werden ihre Attribute (z. B. das Present-Bit) verändert.

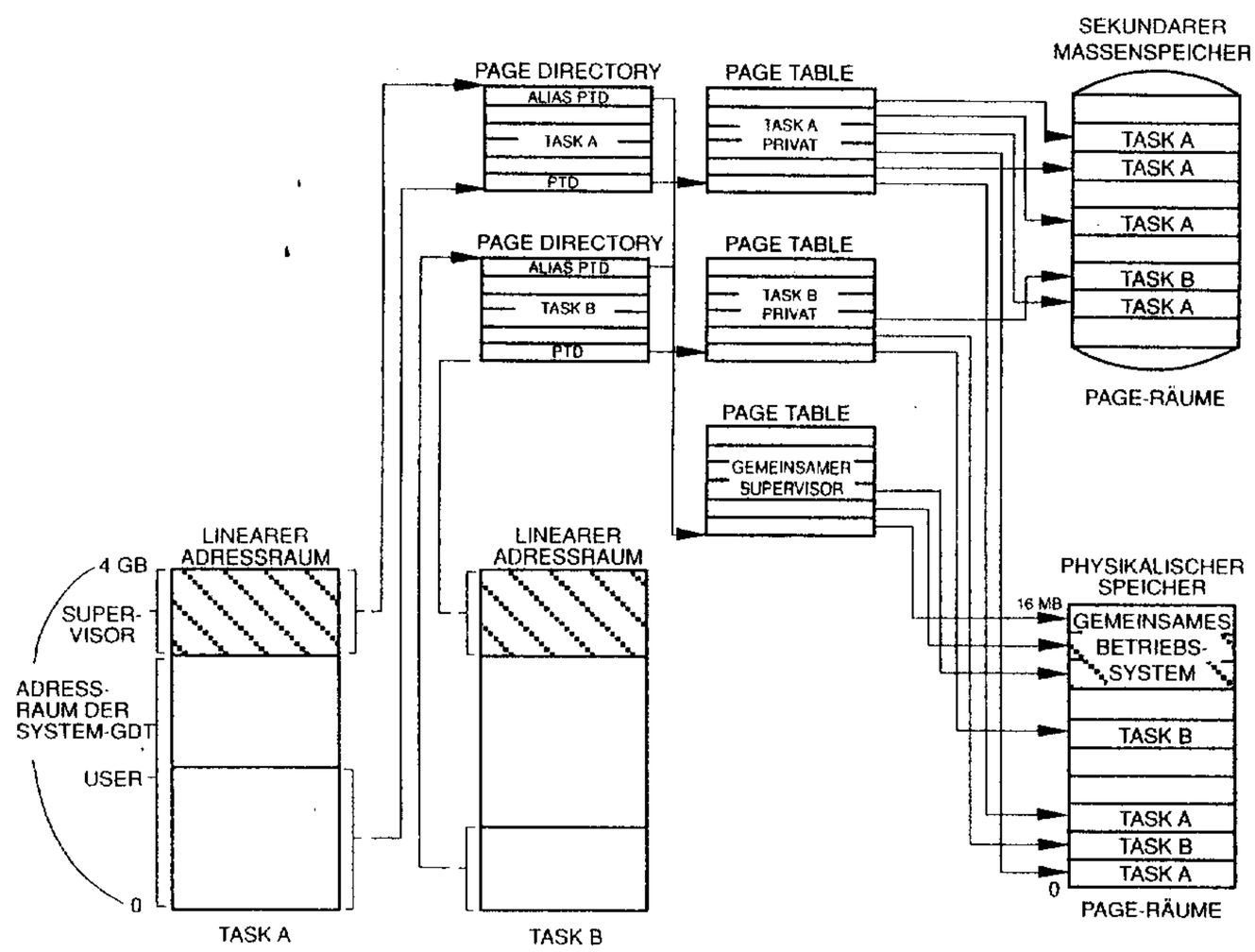
Benützt das System Alias-Page-Deskriptoren für eine gemeinsame Page, müssen nach dem Transfer die Attribute in ***allen*** Deskriptoren aktualisiert werden.

Dies ist ***nicht*** erforderlich, wenn das System eine gemeinsame „Page Table" benützt, da hier die Pages durch normale individuelle Page-Deskriptoren beschrieben werden. Wenn

allerdings die gemeinsame „Page Table" ebenfalls ein- bzw. ausgelagert wird, müssen auch hier die Attribute in ***allen*** Alias-Page-Table-Deskriptoren aktualisiert werden. Dieses Problem kann ein virtuelles Speicher-System aber eliminieren, wenn es auf den Transfer der gemeinsamen „Page Table" zwischen dem sekundären Massenspeicher und dem Arbeitsspeicher verzichtet.

Es liegt daher auf der Hand, daß die System-Software am einfachsten gehalten werden kann, wenn mehrere Tasks die gleiche speicher-resistente „Page Table" benützen.

Wie die Anwendung eines solchen Arrangements dann aussieht, illustriert die folgende Implementation zweier Tasks, die sich über die gleiche „Page Table" das gemeinsame Betriebssystem teilen.



Der 4 Gigabyte lineare Adreßraum jeder Task ist in einen „Supervisor"- und einen „User"-Bereich unterteilt, wobei sich im Supervisor-Bereich jeder Task das Betriebssystem aufhält. Obwohl die Segmentierungs-Einheit für jede Task die gleichen linearen Adressen definiert, werden diese von der Paging-Einheit auf unterschiedliche physikalische Adressen abgebildet.

Dabei sind allerdings Betriebssystem-Referenzen ***ausgenommen***. Wie zu erkennen ist, hat jede Task eine eigene „Page Directory", die jeweils eine ***private*** „Page Table" definiert. Zusätzlich enthalten die Page-Directory-Tabellen Alias-Page-Table-Deskriptoren, die beide auf die gemeinsame „Supervisor-Page-Table" zeigen. Auf diese Weise wird das gemeinsame Betriebssystem in den Adreßraum jeder Task abgebildet.

Im angegebenen Arrangement ist der lineare Adreßraum der beiden Tasks in einen globalen Betriebssystem-Bereich („Supervisor") und einen privaten Anwender-Bereich („User") unterteilt. Dabei werden die Zugriffsrechte auf beide Bereiche ausschließlich über die Page-Schutz-Attribute geregelt. Dies bedeutet, daß der Segment-Schutz ausgeschaltet ist und die System-***GDT alleine*** den linearen Adreßraum der beiden Tasks definiert.