3 Schutz

3.1 Schutztypen

Der 80486 kennt drei wesentliche Anwendungsfälle des Schutzes:

- Die Isolierung von System-Software gegen Benutzer-Software.
- Die Isolierung mehrerer Benutzer gegeneinander (Intertask-Schutz).
- Die Überprüfung von Datentypen.

Der erste Fall ist besonders in einer Umgebung wichtig, in der mehrere Benutzer gleichzeitig unterstützt werden. In dieser Umgebung sind Software-Module, die sich in einer Überwacher-Ebene befinden, vor Modulen in der Benutzer-Ebene geschützt.
Schränkt man die Adressierfähigkeit von Benutzer-Software ein, kann das Betriebssystem die System-Ressourcen ungestört überwachen und steuern, was insbesondere in Multi-User-, Multi-Tasking- und verteilten Verarbeitungs-Systemen eine Grundvoraussetzung bildet.

Der zweite Fall des Schutzes betrifft die Isolierung von Benutzern. Ohne diese Isolierung könnte ein Fehler in einem Benutzer-Programm die Operationen eines anderen fehlerfreien Benutzer-Programms beeinflussen. Solche Interaktionen sind schwierig zu diagnostizieren und zu beseitigen. Eine wesentliche Verbesserung der Zuverlässigkeit von Benutzer-Programmen ist nur dann möglich, wenn die Benutzer voneinander isoliert sind.

Im dritten Fall des Schutzes gewährleistet die 80486-CPU, daß innerhalb eines System- oder Benutzer-Programms alle Code- und Daten-Segmente richtig benützt werden. So ist es nicht möglich, daß Daten ausgeführt und Code modifiziert werden. Zusätzlich müssen sich „Offsets“ innerhalb der definierten Segmentobergrenzen aufhalten usw. Solche Überprüfungen werden bei jedem Speicherzugriff durchgeführt.

3.2 Schutz-Implementierung

Die Schutzhardware des 80486 stellt den Umfang mit dem Speicher, den I/O-Geräten und den Befehlen unter Regeln. Praktisch ist zwar die Anzahl der möglichen Interaktionen zwischen Befehlen, dem Speicher und den I/O-Geräten unbegrenzt; durch den Schutzmechanismus wird sie aber auf eine bestimmte Teilmenge eingeschränkt.

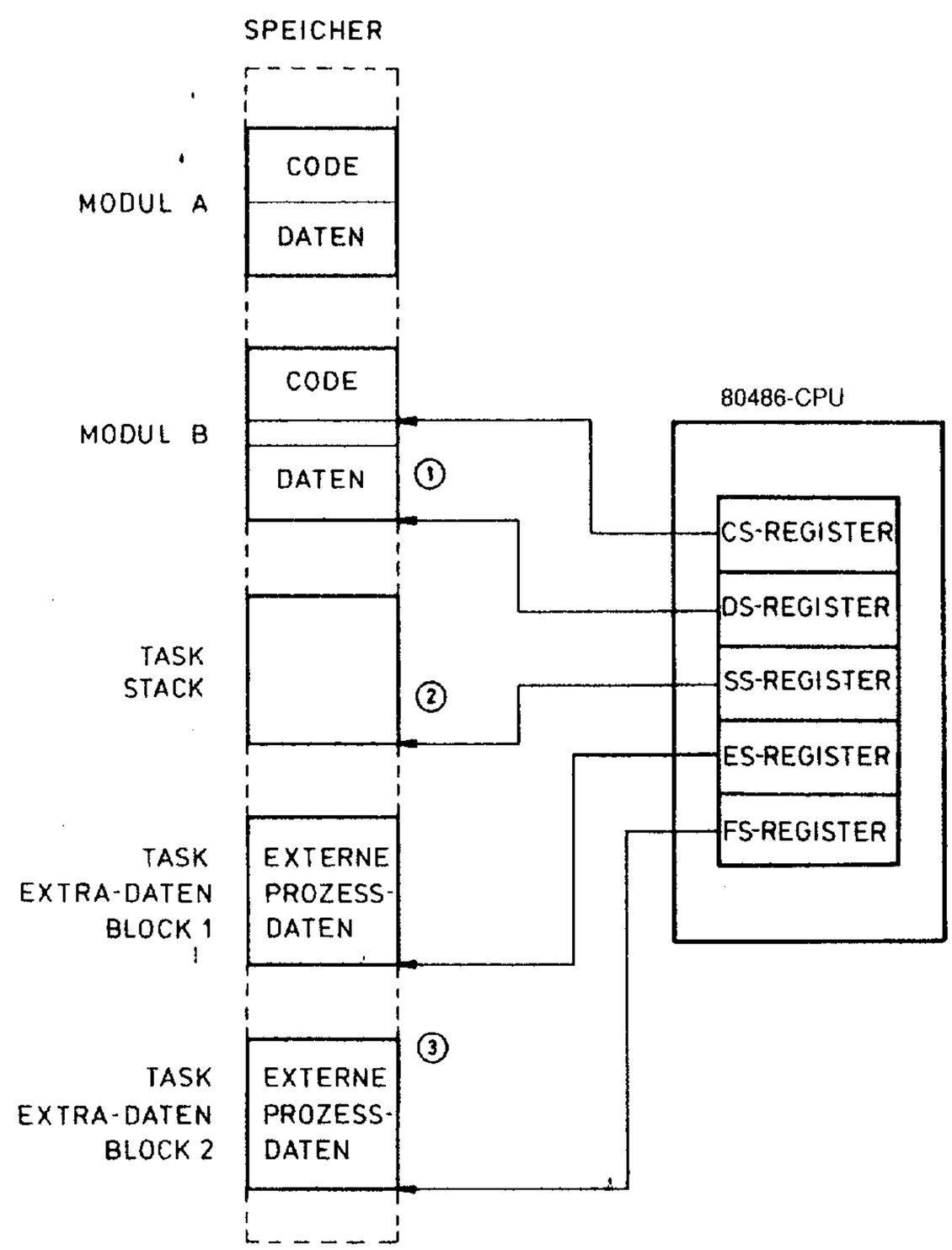
Nur Operationen aus dieser Teilmenge sind korrekt. Jede Operation, die nicht zu dieser Teilmenge gehört, wird von der Schutzhardware festgestellt und als Schutzverletzung signalisiert.

Um den Schutzmechanismus des 80486 zu verstehen, sollen zunächst die Grundbegriffe „Segmente“ und „Tasks“ erläutert werden.

Segmente sind die kleinsten zusammenhängenden Regionen des Speichers mit eigenen Schutz-Attributen. Modulare Programmierung ergibt automatisch Segmente, deren Inhalte als Einheit behandelt werden können.

Im Kapitel „Basis-Programmier-Modell" wurde bereits darauf hingewiesen, daß Segmente den funktionellen Aufbau eines Programms wiedergeben. So existieren z. B. für ein Modul ein Code-Segment, ein Daten-Segment und für die Task ein Stack-Segment.

Das folgende Bild illustriert die Adressierung von Segmenten eines Moduls innerhalb einer Task.



Es ist ersichtlich, daß 2 Module (A und B) existieren, die je ein Code- und Daten-Segment enthalten. Die augenblicklich adressierbaren Segmente in Modul B sind durch einen entsprechenden CS-Selektor und den zugehörigen DS-Selektor ausgewählt ①.

Da sich Modul A und B in der augenblicklich aktiven Task aufhalten, beziehen sie sich auf ein gemeinsames Stack-Segment, das durch einen SS-Selektor ausgewählt ist ②.

Das Bild zeigt weiterhin zwei zusätzliche Datenblöcke, sogenannte Extra-Daten-Segmente, die nur innerhalb der aktiven Task verfügbar sind und z. B. externe Prozeßdaten enthalten. Die Extra-Daten-Segmente können jedem Modul zugewiesen werden. Zu diesem Zweck werden z. B. der ES- und der FS-Selektor benützt, um diese Segmente auszuwählen ③.

Die Speicher-Segmentierung des 80486 wurde für eine optimale Ausführung des Codes unabhängiger Module konzipiert.

Jedes Segment wird durch einen speicher-residenten Deskriptor angesprochen. Zugriffe auf Segmente, die ***nicht*** der Definition des Deskriptors entsprechen, können deswegen von der Schutzhardware verhindert werden. Die Fähigkeit, Zugriffe auf Segmente überwachen zu können, ist Voraussetzung des Softwareschutzes, der durch:

- separate Adreßräume
- Privileg-Mechanismen

verwirklicht wird.

Separate Adreßräume trennen die Anwender-Programme voneinander, Privileg-Mechanismen isolieren System-Software von Anwender-Software.
Software, die das gesamte System steuert, muß einen hohen Vertrauensgrad haben und wird sich typischerweise in der privilegiertesten Ebene aufhalten.
Eine „normale" Anwender-Software hat mit diesen Steuer-Mechanismen nichts zu tun. Sie kommen nur dann zur Anwendung, wenn Task-Wechsel erfolgen oder Betriebssystem-Routinen aufgerufen werden müssen.

Einen wichtigen Unterschied gibt es zwischen Tasks und Programmen. Programme sind ***statisch*** und bestehen aus einem festen Satz von Code- und Daten-Segmenten. Dabei hält sich jedes dieser Segmente in einer bestimmten Privileg-Ebene auf.

Im Gegensatz dazu sind Tasks ***dynamisch***. Sie führen ein oder mehrere Programme aus, wenn sich diese im Adreßraum der Task aufhalten. Zusätzlich bestimmen die von der Schutzart her vorgegebenen Regeln, wann ein Programm ausgeführt werden darf, und nach der Ausführung, was das Programm tun darf.

3.3 Memory Management und Schutz

Die Schutzhardware des 80486 ist eng mit der Memory Management-Hardware verbunden. Schutz-Attribute werden zusammen mit der Memory Management-Information (wie Basisadresse und Limit) im Segment-Deskriptor gespeichert.

Die Schutz-Attribute eines Segments werden spezifiziert, wenn durch den „System-Builder" BLD386/486 das Segment eingerichtet wird. Diese Attribute kennzeichnen unter anderem die Privileg-Ebene des Segments und definieren den Segment-Typ (z. B. Code-Segment, Daten-Segment usw.).

Somit besteht die Schutzinformation für ein Segment aus:

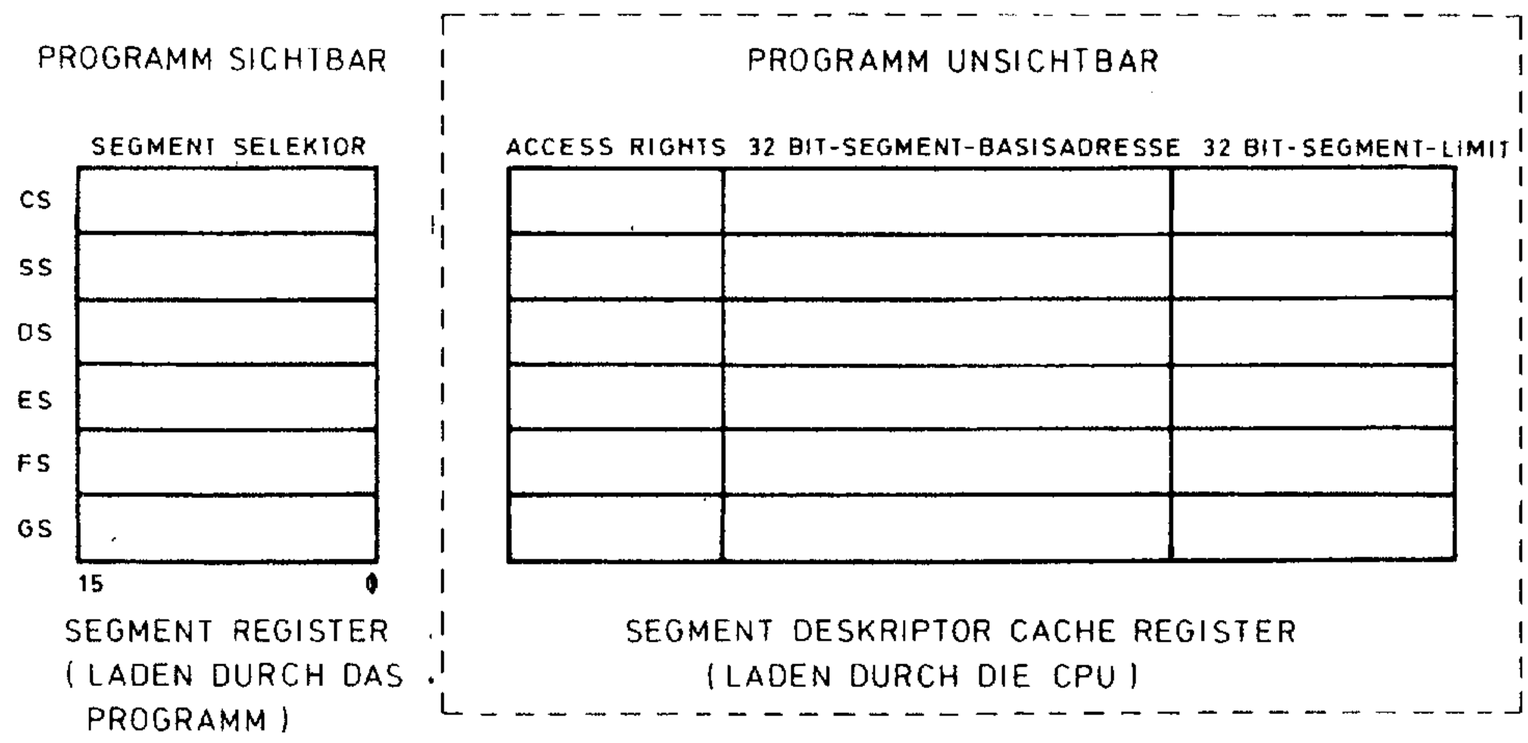
- Segment-Typ
- Privileg-Ebene
- Segment-Größe.

Segment-Typ und Privileg-Ebene sind im Access-Byte des Segment-Deskriptors codiert. Die Segment-Größe ist im Limit-Feld gespeichert.

Geändert werden Schutzinformationen nur, wenn die Selektor-Komponente eines Segment-Registers vom Programm geladen wird. Immer dann initialisiert die CPU den

unsichtbaren Teil des Segment-Registers mit einem neuen Deskriptor. Da sich dieses Deskriptor-Register während der Zugriffe auf die Segment-Eintragungen für den Programmierer „versteckt“ hält, wird es als Segment-Deskriptor-**Cache**-Register bezeichnet.

Das folgende Bild soll dies verdeutlichen.



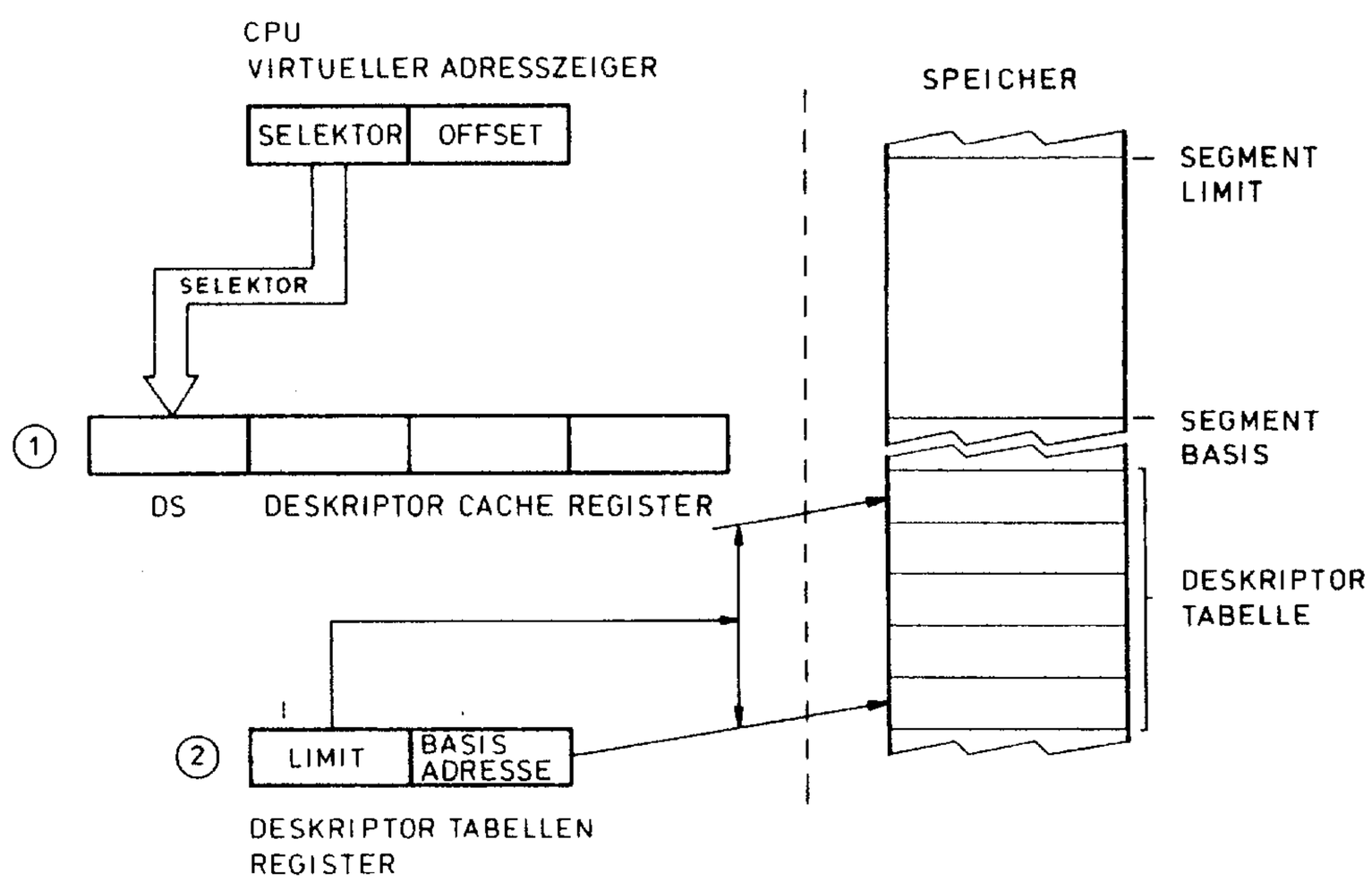
Auf die Schutzdaten Segment-Typ, Privileg-Ebene und Segment-Größe bezieht sich die CPU *zweimal*, nämlich

- bei jedem Laden eines Segment-Registers und
- bei jedem Zugriff auf Eintragungen im selektierten Segment.

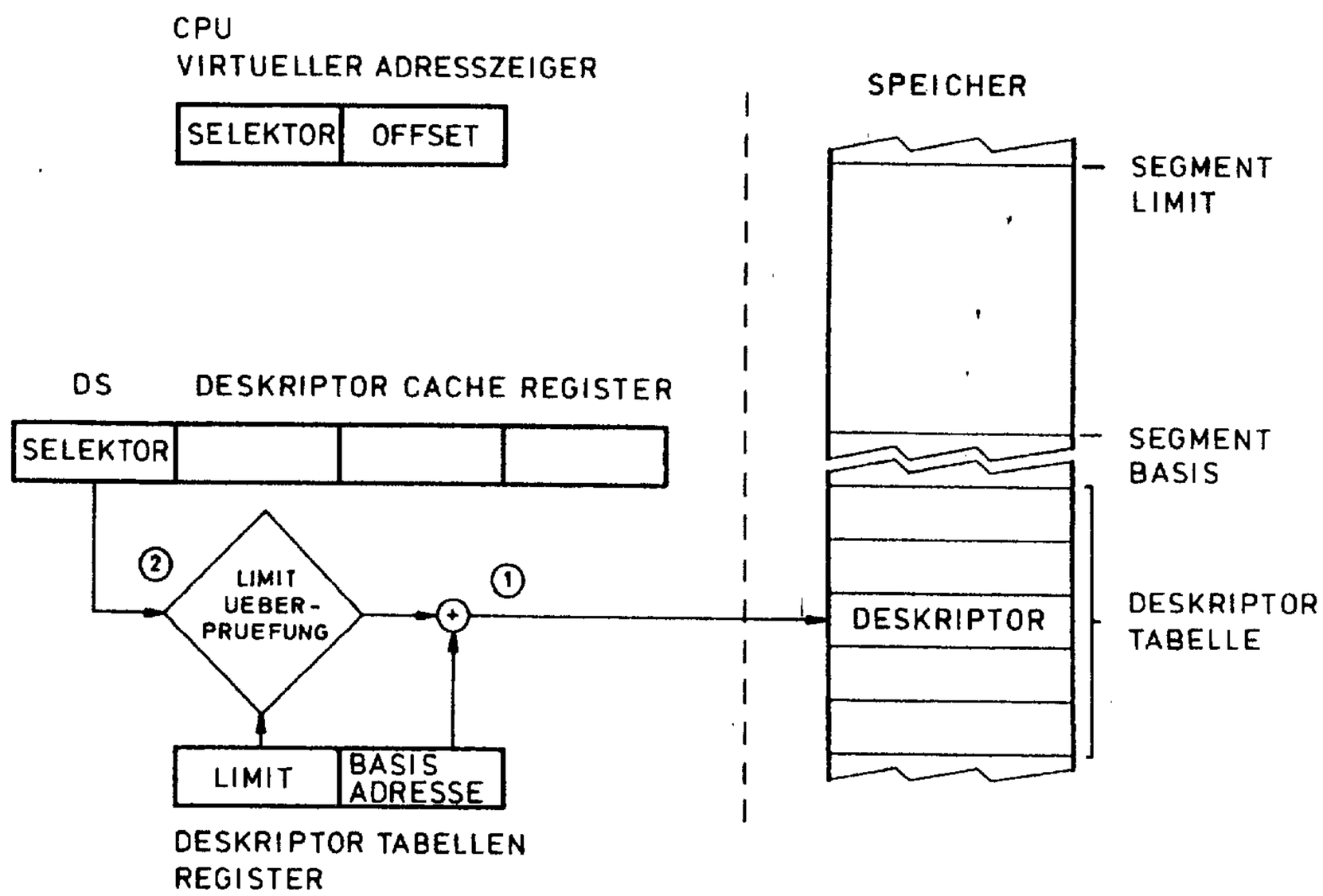
Die Hardware nimmt dabei verschiedene Überprüfungen vor, die von den Schutzregeln vorgegeben sind.

Anhand der nachstehenden Bilderfolge sollen diese Überprüfungen erläutert werden.

1) Zunächst lädt das Programm einen Selektor in eines der Segment-Register. Im Beispiel ist dies das Register DS ①. Die Position und die Größe der Deskriptor-Tabelle sei entweder durch das „Global Descriptor Table Register“ GDTR oder das „Local Descriptor Table Register“ LDTR gekennzeichnet ②.

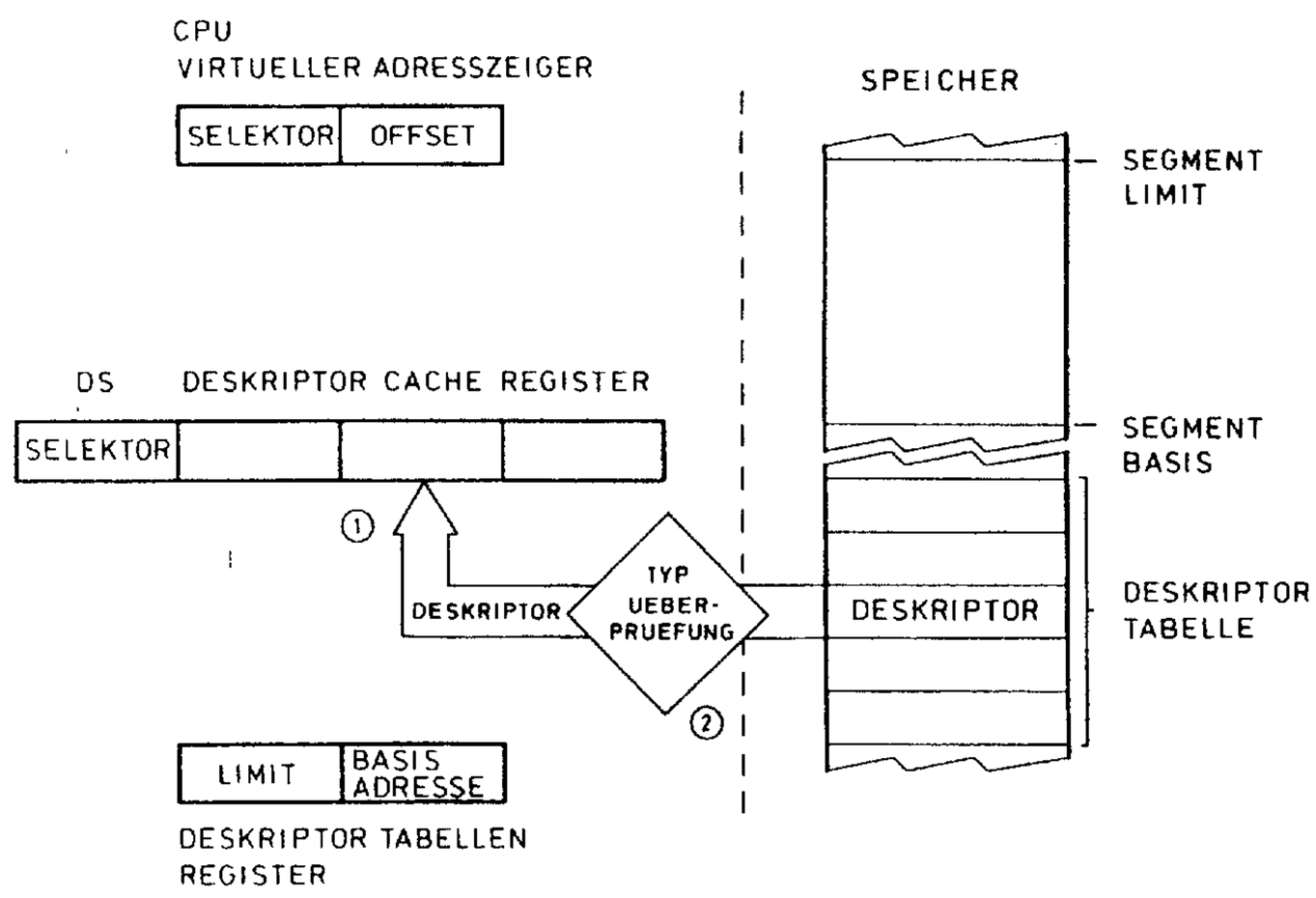


2) Ist der Selektor geladen, addiert die CPU den Selektor-Index zur Basisadresse der Deskriptor-Tabelle (GDT oder LDT), um einen Deskriptor auszuwählen ①.



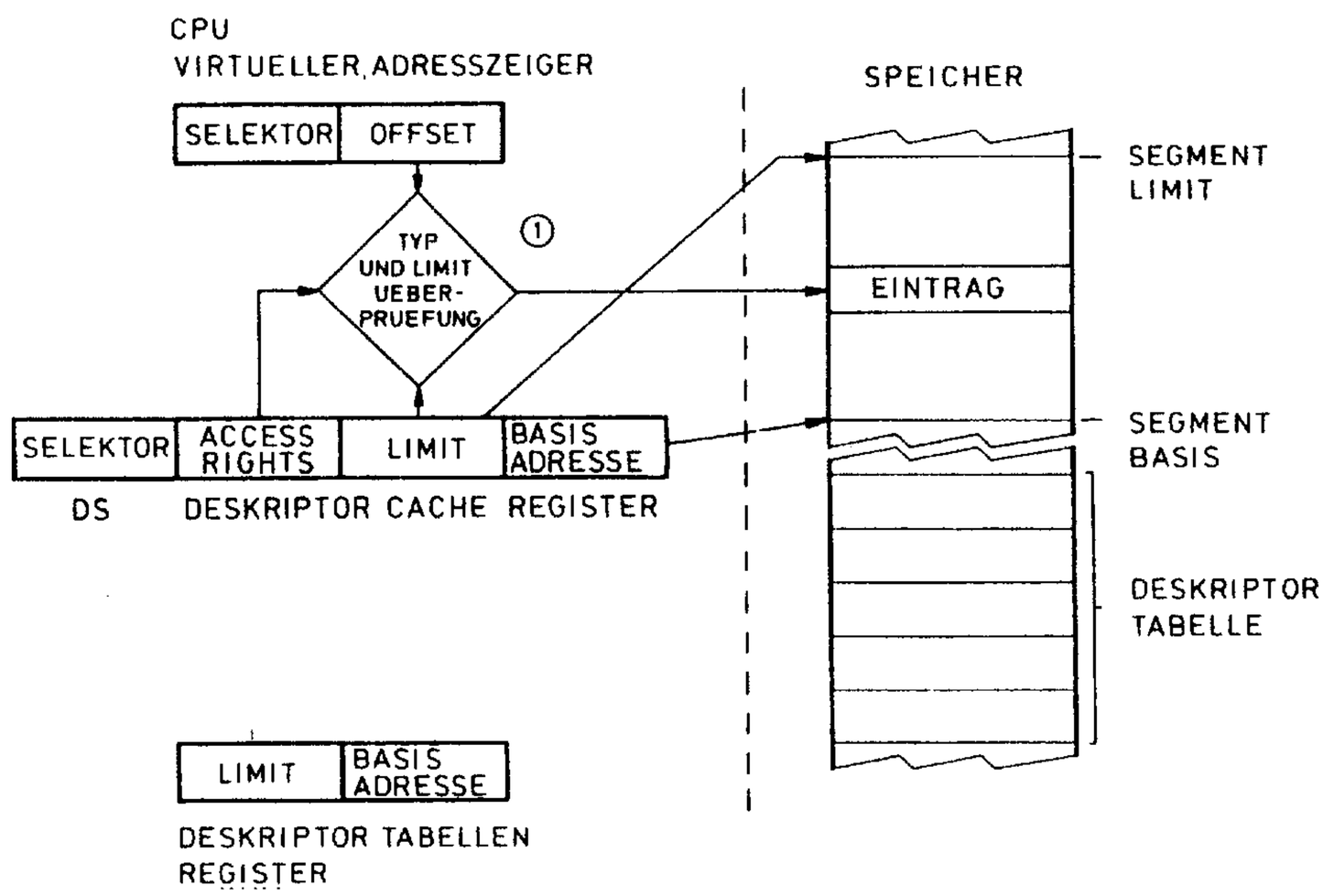
- Alle Deskriptor-Tabellen haben eine spezifische Obergrenze (Limit). Sie wird von der Schutzhardware benützt, um die Adreßraum-Trennung der Tasks sicherzustellen.
- Jeder Versuch des Programms, ein Segment-Register oder auch das „Local Descriptor Table Register" LDTR mit einem Selektor zu laden, der sich auf einen Deskriptor ***außerhalb*** der korrespondierenden Tabellen-Obergrenze bezieht, führt zwangsläufig zur Unterbrechung des Programms.
  Eine solche Situation wird als Ausnahme bzw. „Exception" bezeichnet. Die CPU nimmt also bei jedem Zugriff auf eine Deskriptor-Tabelle eine Limit-Überprüfung vor ②.
- Es sei darauf hingewiesen, daß die GDT oder LDT nur gültige Deskriptoren enthalten dürfen.
  Allerdings sind Löcher bzw. „leere" Deskriptoren erlaubt. Leere Deskriptoren bieten die Möglichkeit, Segmente dynamisch zuzuweisen oder auch zu tilgen, ohne die Größen der GDT oder LDT ändern zu müssen.
  Der Versuch, einen Selektor zu laden, der sich auf einen leeren Deskriptor bezieht, führt aber zur Unterbrechung des Programms.

3) Hat der Zugriff auf einen Deskriptor zu keiner „Exception" geführt, kopiert die CPU den Deskriptor in das DS-Cache-Register ①.



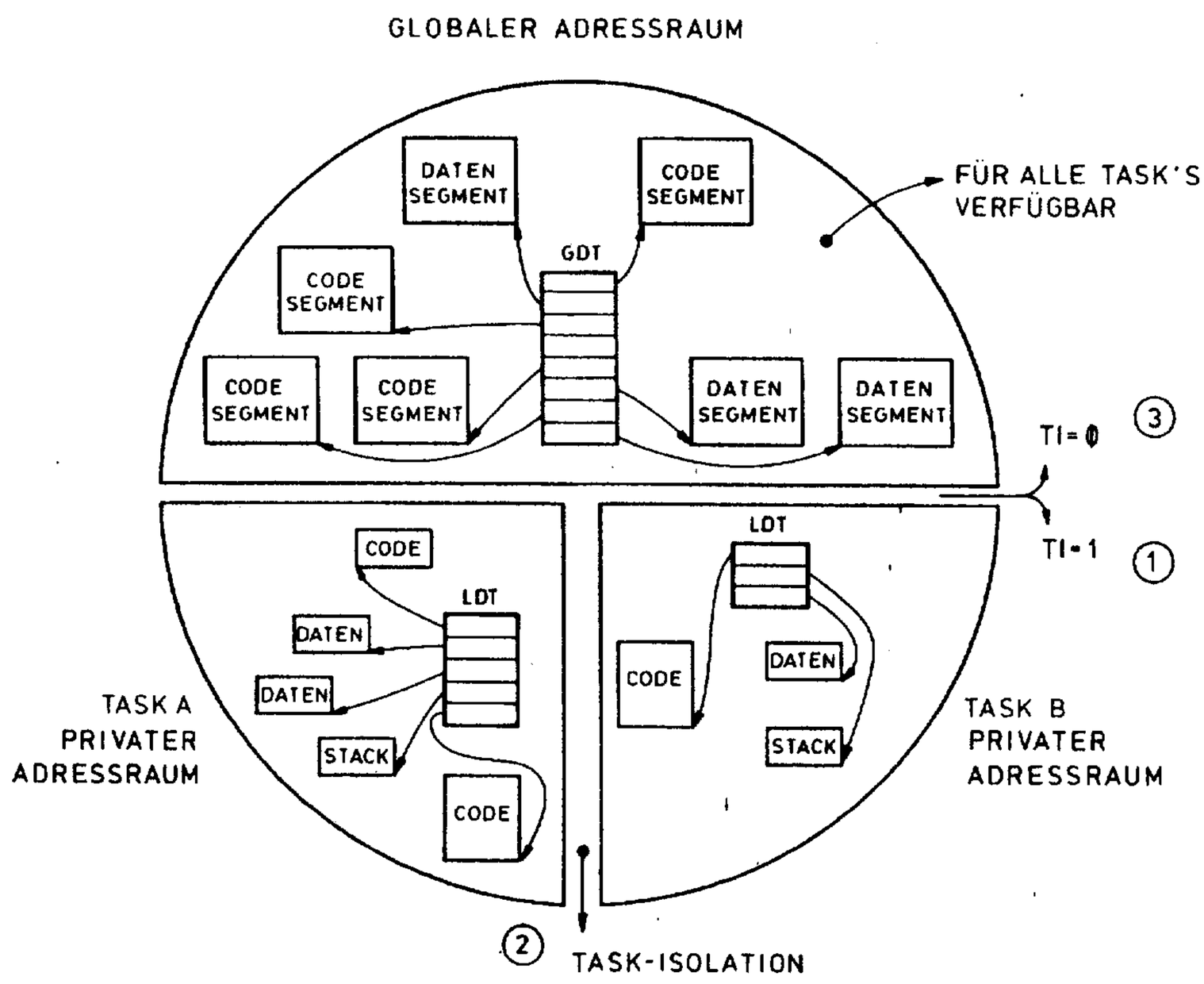
- Jedesmal, wenn der unsichtbare Teil eines Segment-Registers mit einem Deskriptor geladen wird, überprüft die Schutzhardware, ob dieser Deskriptor zum Typ des Segment-Registers paßt ②.
- So ist es z. B. nicht möglich, daß der Deskriptor eines „Read Only"-Segments in das SS-Cache-Register geladen wird, da ein in diesem Register enthaltener Deskriptor ein Stack-Segment spezifiziert, das auch beschreibbar sein muß (Typ „Read/Write").
- Die gleiche Typüberprüfung wird auch beim Laden des „Local Descriptor Table Registers" LDTR durchgeführt. Hier darf nur ein LDT-Deskriptor in den unsichtbaren Teil des Registers kopiert werden.
- Jede Verletzung dieser Schutzregeln stellt eine „Exception" dar und führt zur Unterbrechung des Programms.

4) Ist ein Deskriptor erfolgreich geladen worden, benützt nun die Schutzhardware die Deskriptor-Informationen, um den Typ und die Obergrenze des ausgewählten Segments zu überprüfen ①.



- Während die Adreßeinheit den Offset auf die Basisadresse des Segments addiert, prüft gleichzeitig die Schutzhardware, ob dieser Offset nicht größer als die Segmentlänge ist. Sie verhindert dadurch, daß ein Zugriff über das Ende des Segments hinaus und damit eventuell auf ein anderes Segment erfolgt.
- Parallel dazu wird überprüft, ob der Zugriff mit dem Typ des Segments vereinbar ist. Dadurch wird z. B. verhindert, daß in ein „Execute Only"- oder „Read Only"-Segment geschrieben oder aus einem „Execute Only"-Segment gelesen wird.
- Jede Verletzung dieser Schutzregeln stellt eine „Exception" dar und führt zur Unterbrechung des Programms.

Die erläuterten Überprüfungen, die bei jedem Speicherzugriff durchgeführt werden, gehören zum einfacheren Typ des im 80486 implementierten Schutzmechanismus. Durch diese Überprüfungen kann sichergestellt werden, daß mehrere Benutzer bzw. Tasks voreinander geschützt sind.
Dies soll das folgende Bild illustrieren.



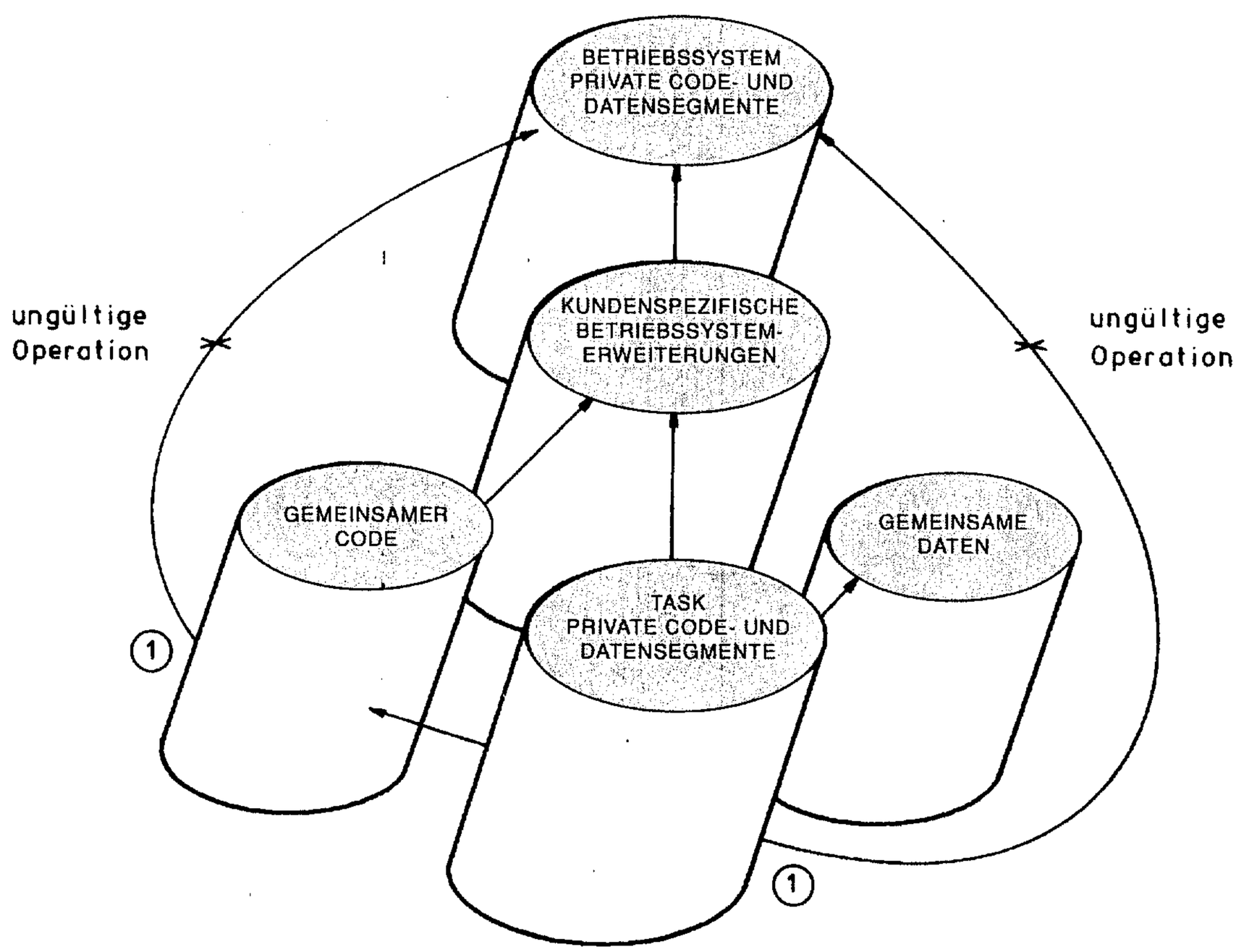
Der gültige Adreßraum einer Task wird durch ihre LDT und die GDT spezifiziert. Während die LDT den privaten Adreßraum einer Task festlegt, definiert die GDT einen Adreßraum, der für alle Tasks verfügbar ist.
Bei jedem Zugriff auf einen Deskriptor in der LDT (Table Indikator = 1) ① stellt die Schutzhardware fest, ob sich der Deskriptor jenseits der adressierbaren LDT befindet (Index-Limit-Überprüfung). Wenn ja, käme dies einem Zugriff auf einen Adreßraum gleich, der außerhalb der aktiven Task liegt. Eine Programmunterbrechung verhindert dies und garantiert damit die Isolierung der Tasks ②.

Die gleiche Limit-Überprüfung wird bei einem Zugriff auf einen Deskriptor in der GDT (Table Indikator = 0) durchgeführt ③. Auch hier verhindert eine Programmunterbrechung, daß ein Zugriff auf eine Speicher-Position erfolgt, die außerhalb des globalen Adreßraums liegt.

Im gültigen Adreßraum einer Task halten sich die Segmente auf, die mit bestimmten Zugriffsrechten ausgestattet sind. Bei jedem Zugriff auf einen Eintrag überprüft die Schutzhardware, ob eine Verletzung dieser Zugriffsrechte vorliegt, und stellt zusätzlich fest, ob der Eintrag ungültig ist, sich also jenseits des adressierbaren Segments befindet (Offset-Limit-Überprüfung).
Ist das Ergebnis der Überprüfung positiv, löst die Schutzhardware in beiden Fällen eine Programmunterbrechung aus.

3.4 Privilegebenen und Schutz

Um innerhalb der aktiven Task Betriebssystem-Software von Anwender-Software zu isolieren, ist eine weitere Dimension des Schutzes notwendig.
Das folgende Bild zeigt das Modell einer typischen Task-Umgebung zusammen mit dem erforderlichen Zugriffsschutz.



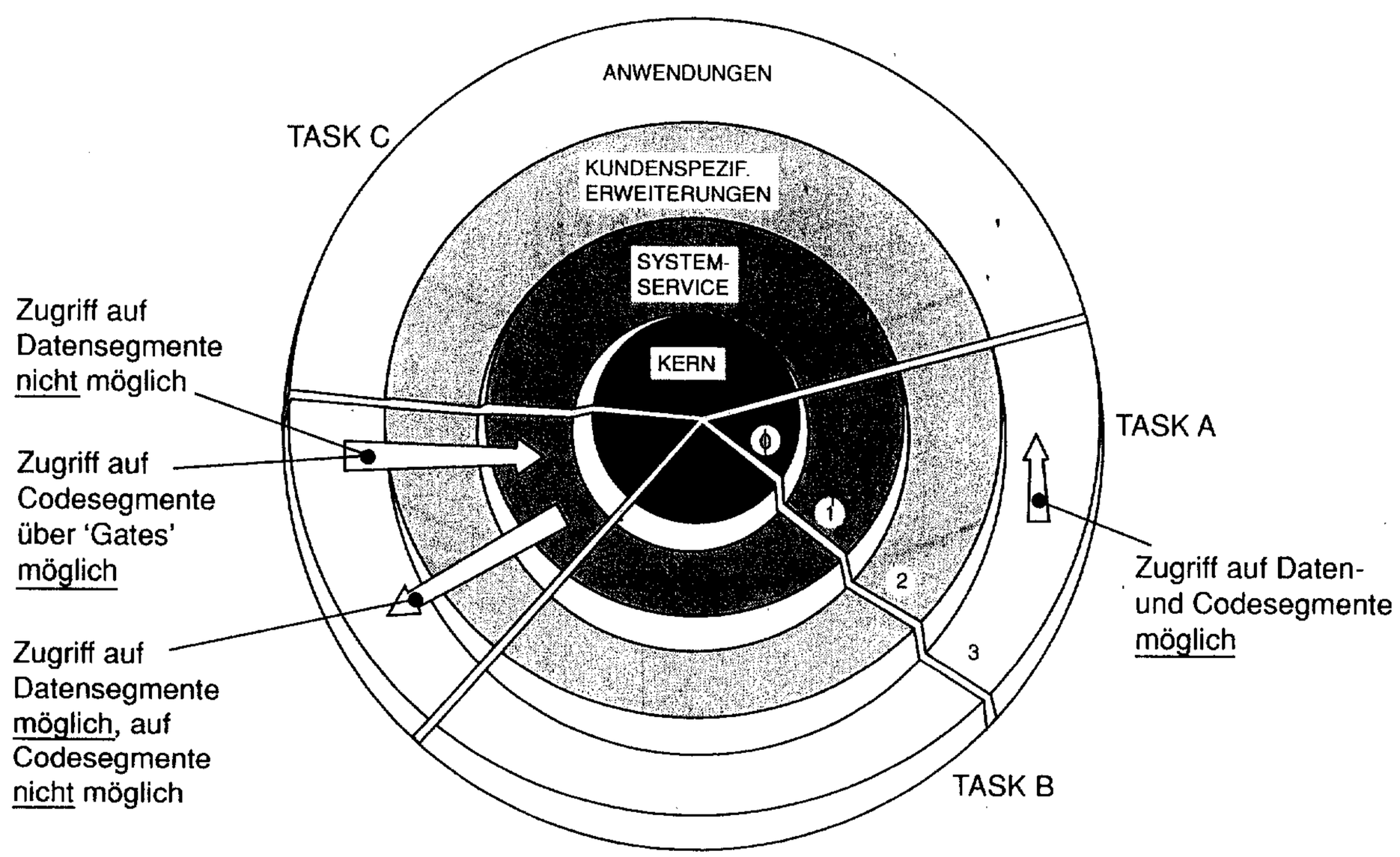
Im lokalen Adreßraum der Task liegen deren private Code- und Datensegmente.

Im globalen Adreßraum der Task liegen

- gemeinsame Code- und Datensegmente, die für alle Tasks verfügbar sind,
- kundenspezifische Betriebssystem-Erweiterungen, die für alle Benutzer verfügbar sind, und
- private Code- und Datensegmente des Betriebssystems.

Die Betriebssystem-Software ist nur dann vor Anwender-Software zu schützen, wenn direkte Zugriffe einer Task oder eines globalen Codes auf Routinen des Betriebssystems als ungültige Operationen erkannt werden ①.
Im Gegensatz dazu ist diese Auflage bei Zugriffen auf die Betriebssystem-Erweiterungen oder globale Code- und Datensegmente nicht erforderlich.

Damit innerhalb einer Task der erforderliche Schutz realisiert werden kann, stellt der 80486 eine Privileg-Hierarchie mit vier Ebenen zur Verfügung. Jedes Code- oder Datensegment hält sich auf einer dieser vier Privilegebenen auf. Dabei ist Ebene 0 die privilegierteste Ebene mit dem höchsten Vertrauensgrad, während Ebene 3 den niedrigsten Vertrauensgrad hat, d. h. am ehesten Übergriffen und Störmöglichkeiten anderer Prozeduren ausgesetzt ist.
Durch diese hierarchische Struktur können Schutzmechanismen implementiert werden, die anhand des folgenden Bildes erläutert werden sollen.

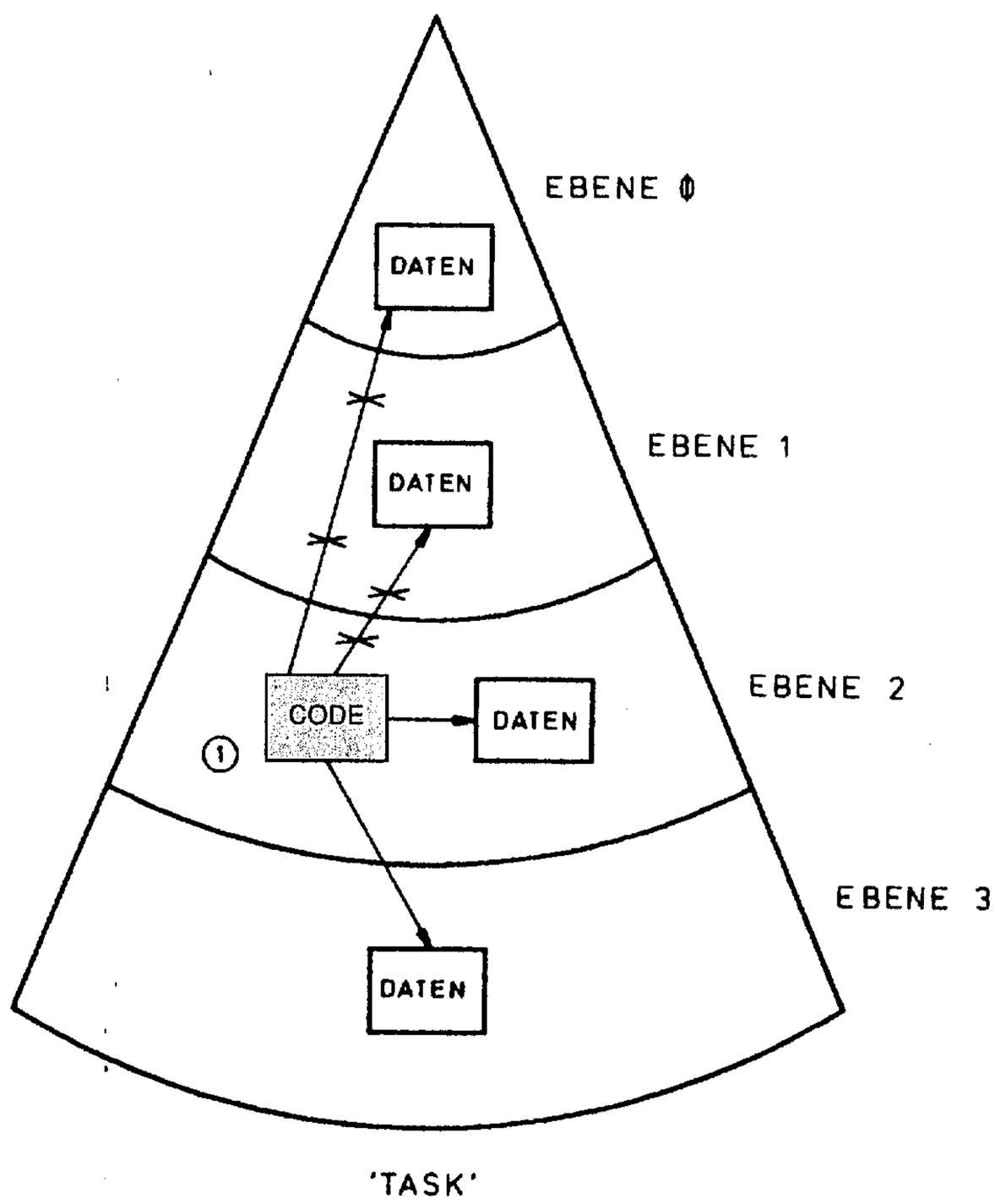


So benützt die 80486-Architektur Privilegebenen, um die Zugriffsfähigkeit eines Codes auf einen anderen Code oder auf Daten zu steuern.

- Da der Vertrauensgrad und damit die Zuverlässigkeit von Programmen immer geringer wird, je niedriger ihre Privilegebene ist, können solche Programme auf Datensegmente in höheren Privilegebenen ***nicht*** zugreifen.

- Allerdings ist der Zugriff eines Programms auf Datensegmente erlaubt, die sich entweder auf der gleichen Privilegebene wie das Programm oder auf einer niedrigeren Privilegebene aufhalten.
- Auf Codesegmente in einer höheren Privilegebene kann nur indirekt über einen programm-transparenten Steuermechanismus, ein sogenanntes „Gate", zugegriffen werden.
- Weiterhin ist es möglich, auf Codesegmente zuzugreifen, die mit dem aufrufenden Programm auf der gleichen Privilegebene liegen.
- Auf Codesegmente in einer niedrigeren Privilegebene kann ***nicht*** zugegriffen werden, weil ihr Vertrauensgrad geringer ist als der des aufrufenden Programms.

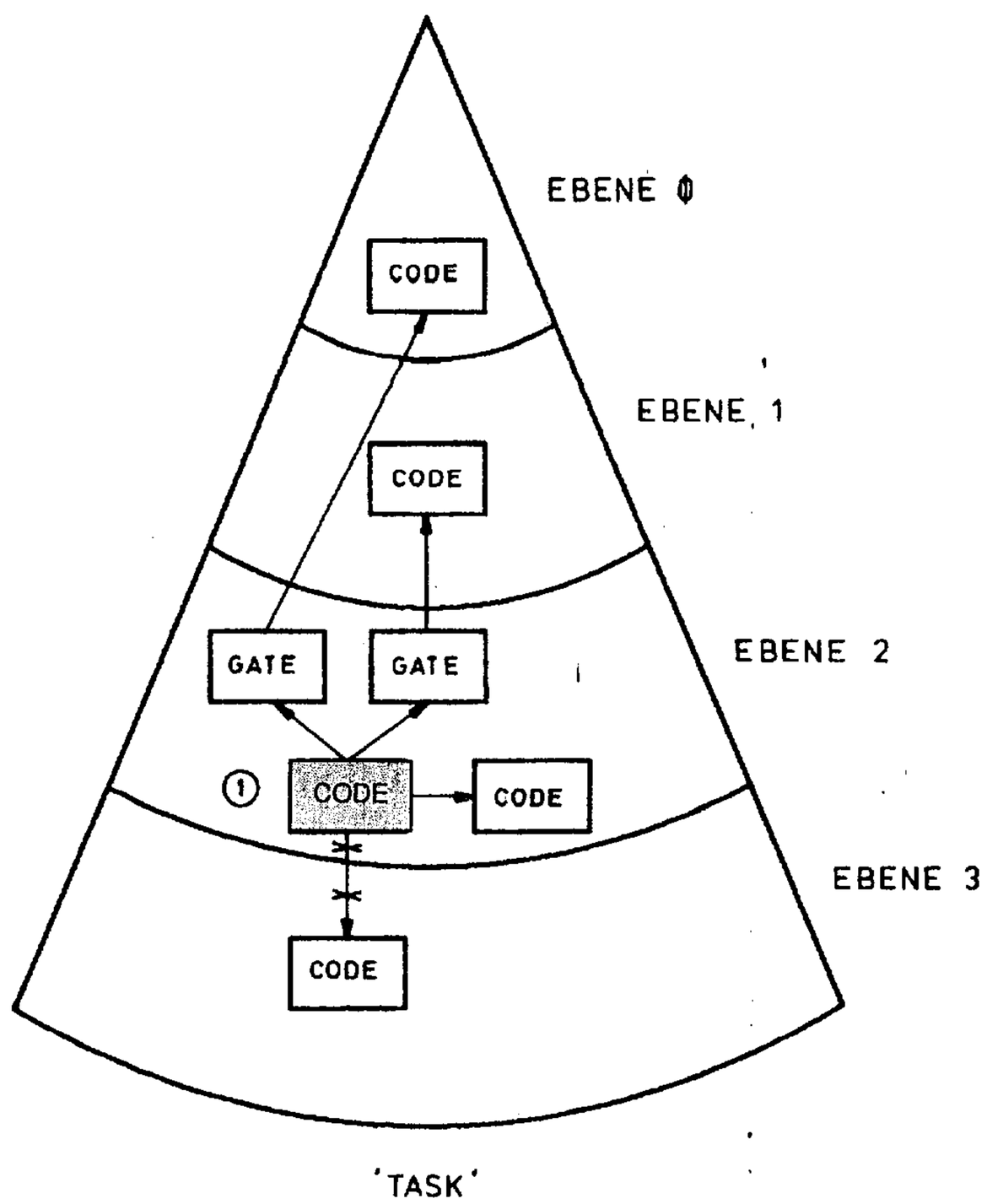
Wie der Vier-Ebenen-Mechanismus die Manipulation von Daten einschränkt, soll das folgende Beispiel verdeutlichen.



Es ist ersichtlich, daß die augenblicklich aktive Task einen Code in der Privilegebene 2 ausführt ①.
Auf die Daten in Ebene 2 und Ebene 3 kann der Code ohne Einschränkung zugreifen, während ein Zugriff auf die Daten in Ebene 1 und Ebene 0 ***nicht*** möglich ist.

Die Aussage dieses Beispiels kann auf den Schutz privater Betriebssystem-Daten übertragen werden.
Da sich die privaten Daten eines Betriebssystems typischerweise in der Privilegebene 0 befinden, sind diese vor Anwenderprogrammen, die sich im Normalfall in der Privilegebene 3 aufhalten, geschützt.

Wie der Vier-Ebenen-Mechanismus den unkontrollierten Zugriff auf Codesegmente verhindert, soll das folgende Beispiel verdeutlichen.



Es ist ersichtlich, daß die augenblicklich aktive Task einen Code in der Privilegebene 2 ausführt ①.
Diese Code kann einen anderen Code in der gleichen Privilegebene (im Beispiel Ebene 2) aufrufen.

Da die Zuverlässigkeit des Codes in Ebene 3 geringer ist als die des Codes in Ebene 2, ist ein Zugriff auf den Code in Ebene 3 ***nicht*** möglich.
Damit ein Programm mit geringer Zuverlässigkeit nicht „wild" in höherprivilegierte Routinen springen kann, ist der direkte Zugriff auf diese Routinen ***nicht*** erlaubt.
So kann der Code in Ebene 2 die Codes in Ebene 1 und 0 nur über „Gates" erreichen, wobei diese „Gates" ***feste*** Einsprungadressen vorgeben. Dadurch ist ein unkontrollierter Zugriff ausgeschlossen.
Die Aussage dieses Beispiels kann auf den Schutz von privatem Betriebssystem-Code übertragen werden.
Da sich der private Code eines Betriebssystems typischerweise in der Privilegebene 0 befindet, ist dieser vor unkontrollierten Zugriffen der Anwenderprogramme, die sich im Normalfall in der Privilegebene 3 aufhalten, geschützt.

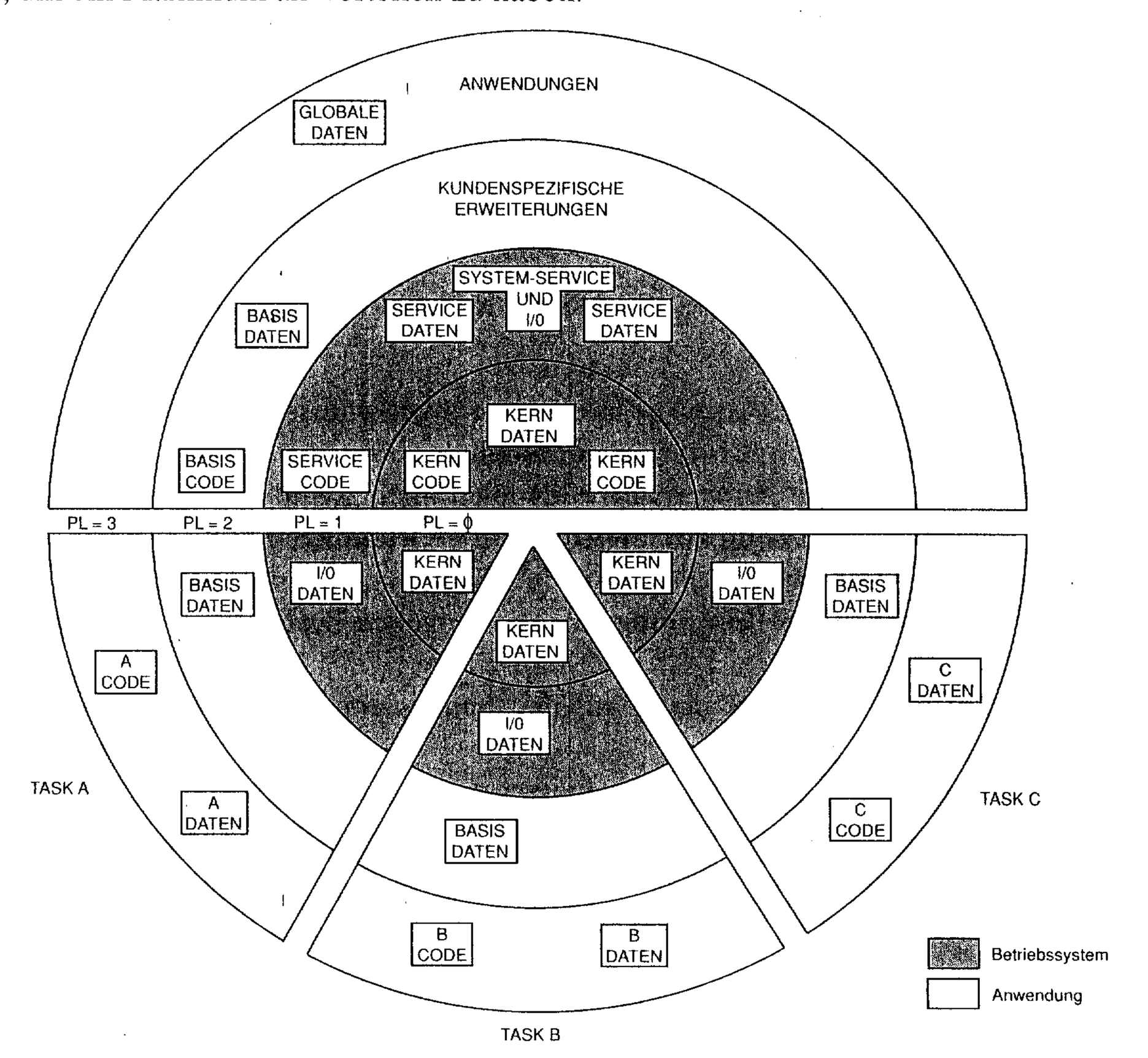
3.5 Struktur des Software-Systems

Die Privileg-Hierarchie der 80486-Architektur gibt dem System-Entwickler die Möglichkeit, Code und Daten auf die verschiedenen Privilegebenen und Tasks zu verteilen. Die daraus resultierende Software-Struktur beeinflußt direkt die Zuverlässigkeit, die Leistungsfähigkeit und die Anpassungsfähigkeit des Systems.

Wenn Software-Module auf verschiedene Privilegebenen und separate Tasks verteilt werden, ergeben sich hierfür einige Vorteile:

1) Es wird eine feinere Körnung des Schutzes erreicht.
2) Da jede Task nur eine bestimmte Aufgabe erledigen kann (z. B. Lesen einer Tastatur), erfolgt die Umschaltung auf andere Tasks automatisch. Dadurch ist die Synchronisation der anfordernden Tasks nicht notwendig.
3) Die Kommunikation zwischen Anwender-Software und Betriebssystem ist schneller.
4) Für alle Tasks ist es möglich, die gleichen Betriebssystem-Funktionen parallel auszuführen.

Das folgende Bild illustriert eine 80486-Software-Struktur, die alle vier Privilegebenen benützt, um ein Maximum an Vorteilen zu haben.

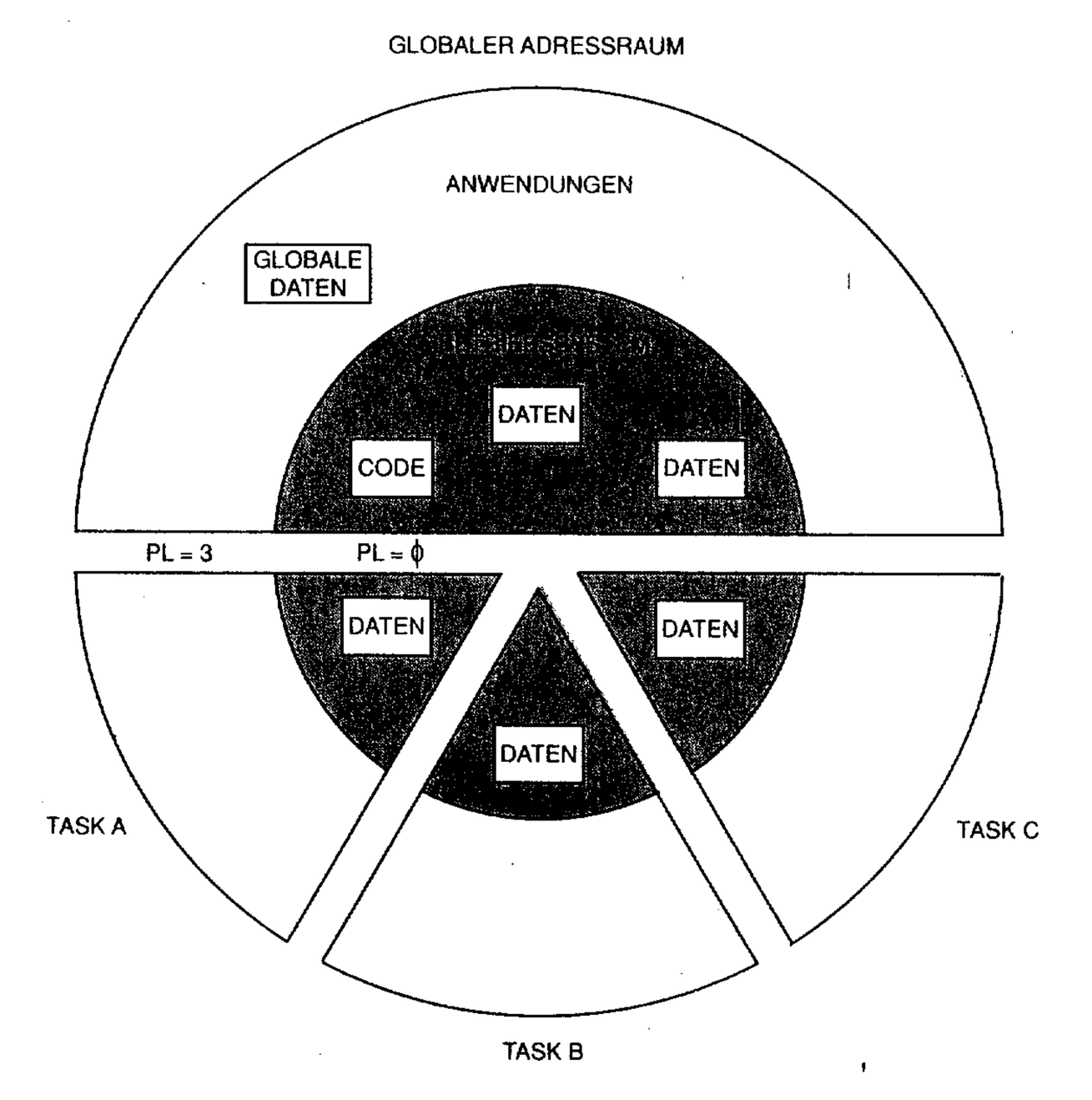


Die kritischen Prozeduren und Daten des Betriebssystem-Kerns sind vor allen anderen Prozeduren im System geschützt. Zu diesen kritischen Regionen gehören z. B. Deskriptor-Tabellen und Speicher-Zuweisungs-Prozeduren oder Informationen über Tasks. Sie gehören daher zum Kern des Betriebssystems und liegen auf der Privilegebene 0 (PL = 0).

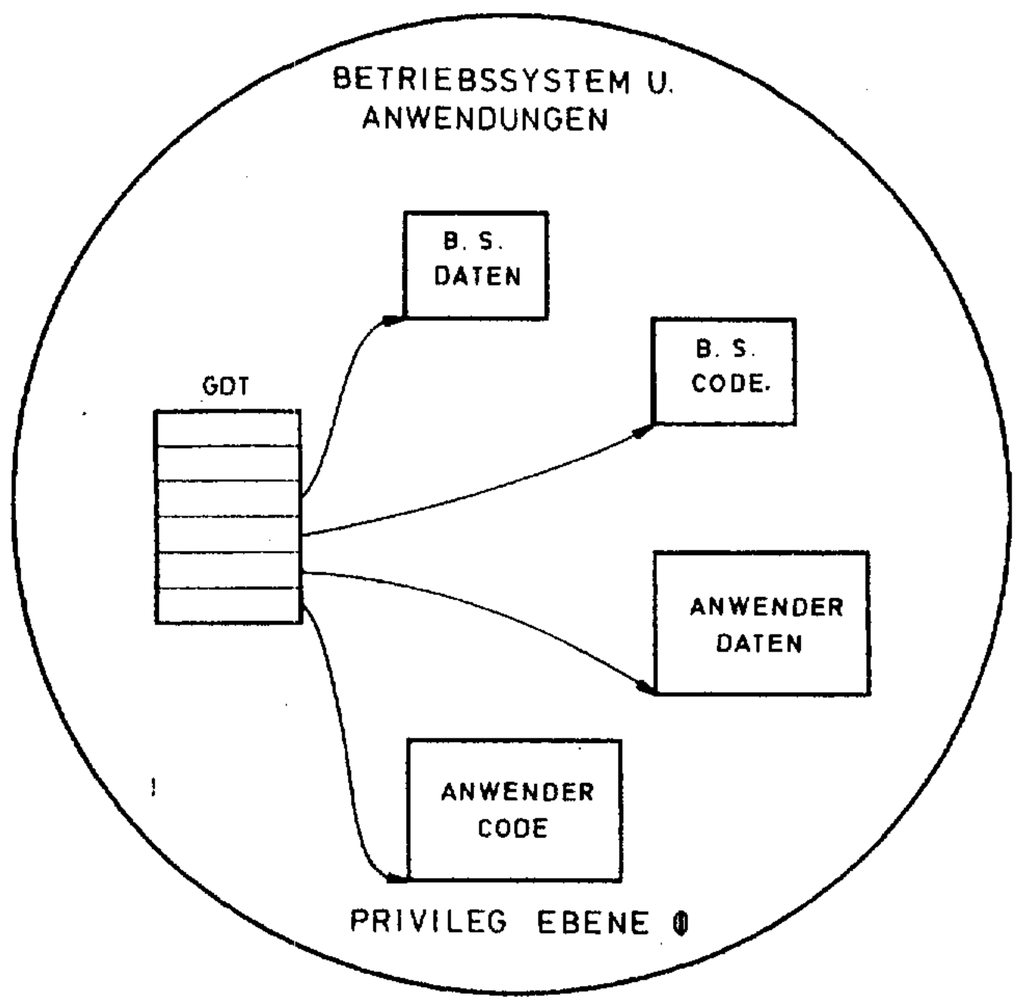
Jene Teile des Betriebssystems, die weniger zuverlässig sind, liegen auf der Privilegebene 1 (PL = 1) und sind deshalb noch vor Anwenderebenen geschützt. Die geringere Zuverlässigkeit dieser Betriebssystem-Teile ist entweder auf ihre natürliche Komplexität zurückzuführen, die z. B. für I/O-Subsysteme charakteristisch ist, oder auf gelegentlich durch sie ausgelöste Maßnahmen, die den Datendurchsatz erhöhen sollen.

Die Anwender-Logik hat zwei Ebenen, so daß kundenspezifische Betriebssystem-Erweiterungen wie z. B. das Management von Basisdaten oder die Verwaltung von Dateien vom Zugriff des reinen Anwender-Codes geschützt werden können.

Wie das folgende Bild zeigt, ist es ***nicht*** unbedingt notwendig, alle vier Privilegebenen zu benützen. So kann der Anwender die angegebenen zwei Privilegebenen bevorzugen, wenn er sein System an die traditionellen Multi-Tasking-Systeme angleichen will. Solche Systeme bieten nur zwischen zwei Ebenen Schutz, wobei die eine Ebene durch die Anwendungen und die andere Ebene durch das Betriebssystem definiert ist.



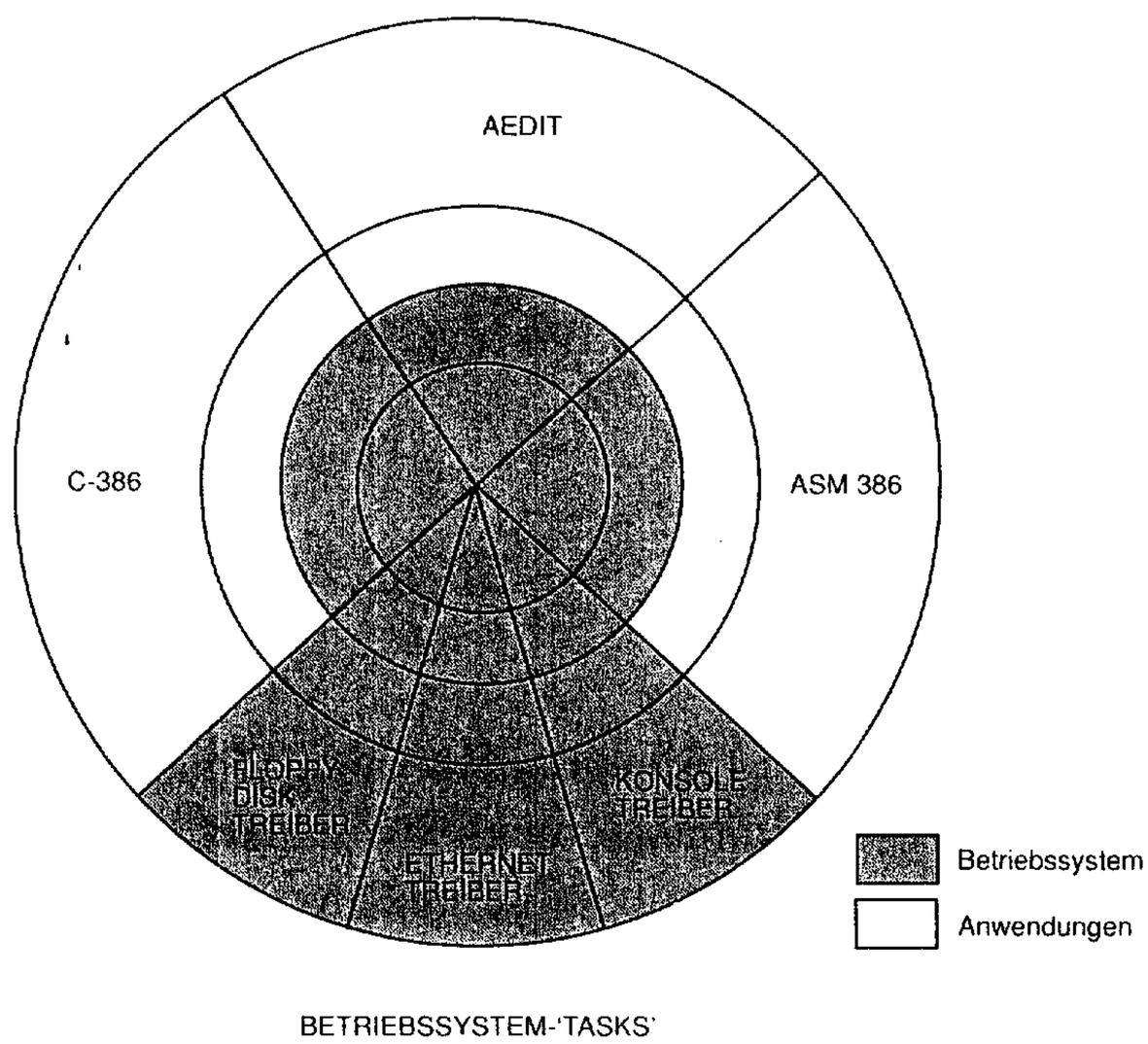
Wie das folgende Bild illustriert, kann der 80486 auch ein Ein-Ebenen-System emulieren. Dieses System kann in der Initialisierungs-Phase nützlich sein, wenn der Wechsel von einer ungeschützten Umgebung in eine geschützte erfolgt. Allerdings nimmt ein solches System die vielen Vorteile des Schutzes, die der 80486 bietet, nicht in Anspruch. So ist beispielsweise das Betriebssystem vor Anwender-Software ***nicht*** geschützt.



B.S. = Betriebssystem

Manchmal ist es besser, einige Betriebs-System-Funktionen als unabhängige Betriebs-System-Tasks und nicht als privilegierte Prozeduren innerhalb einer Task zu strukturieren. Dafür eignen sich bestimmte I/O-Funktionen. Da diese Funktionen erst durch I/O-Bausteine aktiviert werden, ist es vorteilhaft, sie als separate Tasks zu behandeln, die völlig asynchron ablaufen können.

Damit die Zuverlässigkeit des Systems durch die Komplexität der I/O-Routinen nicht beeinträchtigt wird, ist ihr gegenseitiger Schutz durch die Task-Grenzen gegeben. Das nachstehende Bild illustriert ein solches System.



3.6 Daten-Segment/Executable-Segment-Deskriptoren

Jeder Zugriff eines Segments auf ein anderes Segment ist nur indirekt über einen Deskriptor möglich. Er enthält alle Informationen über das Segment, auf das der Zugriff erfolgen soll. Jedes Segment muß mindestens einen Deskriptor haben. Im anderen Fall würde das bedeuten, daß das Segment im Speicherraum nicht vorhanden ist.

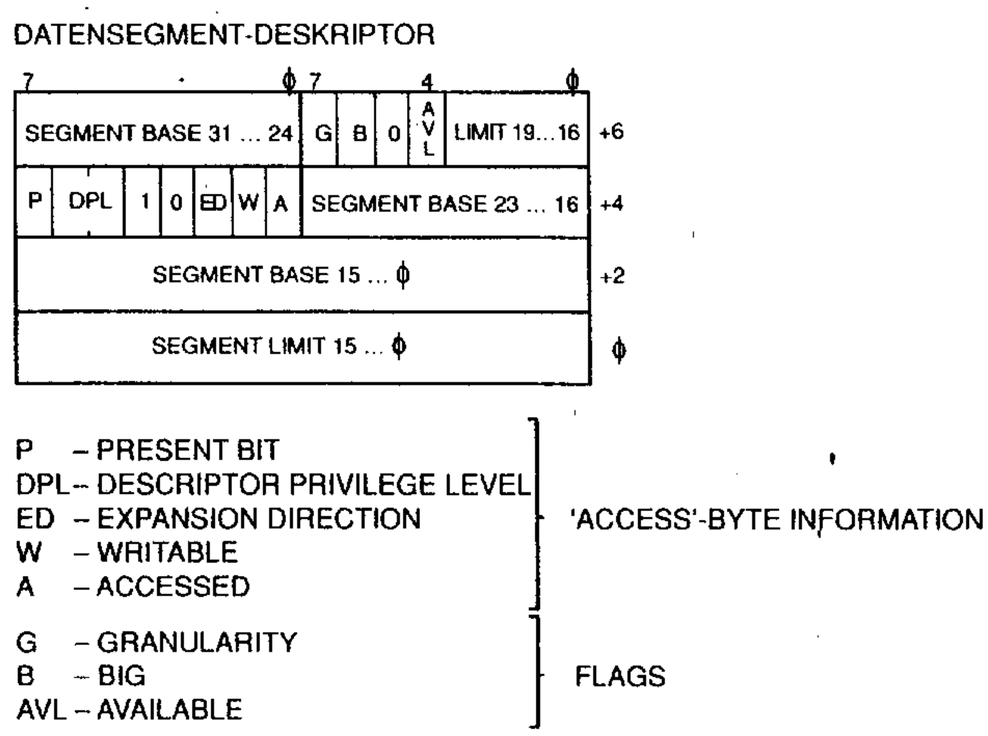
Die Deskriptoren werden entweder vom „System Builder" BLD386/486 für statische Systeme oder von Betriebssystem-Prozeduren für dynamische Systeme festgelegt. Dabei sind ihre Existenz und Funktion für den Anwender-Programmierer völlig „unsichtbar".

Die in einem 80486-System bekannten Deskriptor-Formate lassen sich in vier Klassen einteilen, wobei zunächst die Klasse der

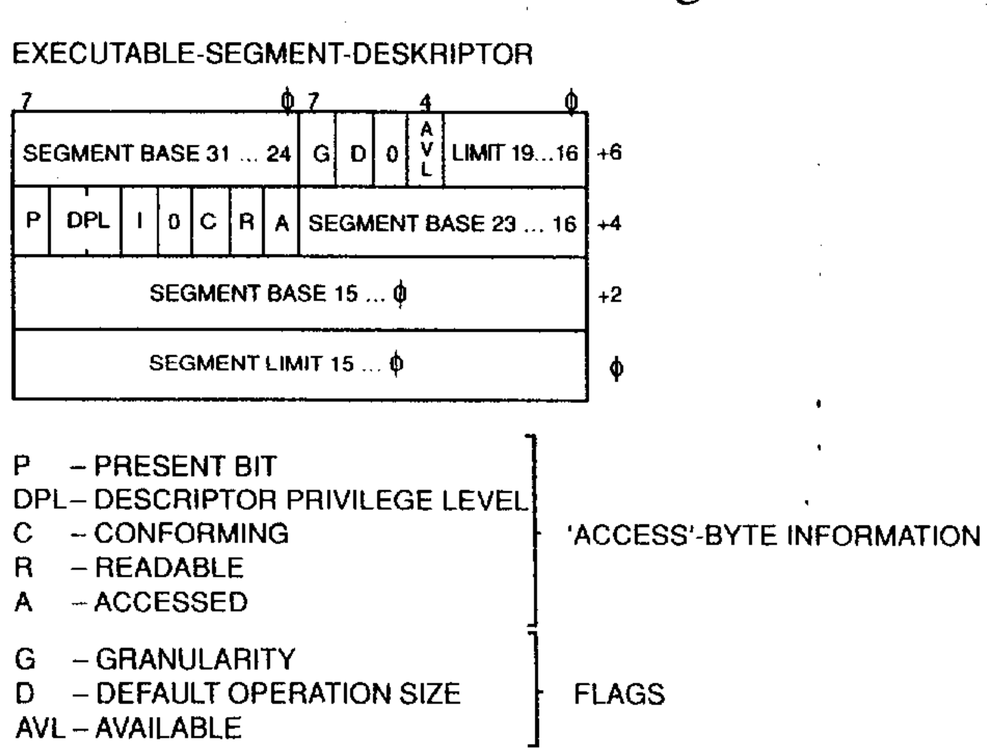
- Daten-Segment-Deskriptoren und der
- Executable-Segment-Deskriptoren

betrachtet werden sollen.

Daten-Segment-Deskriptoren beziehen sich ausschließlich auf solche Segmente, die entweder System- oder Anwenderdaten enthalten. Dazu gehören ***auch*** Stack-Segmente.
Das folgende Bild zeigt das Format eines Daten-Segment-Dseskriptors.



Executable-Segment-Deskriptoren beziehen sich ausschließlich auf Segmente, die ausführbare Befehle enthalten.
Das folgende Bild zeigt das Format eines Executable-Segment-Deskriptors.



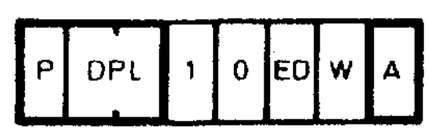
Es ist zu erkennen, daß in beiden Deskriptor-Formaten einige gemeinsame Felder existieren. Diese sind

- das Limit-Feld (Segment-Obergrenze),
- das Base-Feld (Segment-Basisgrenze),
- die Bit-Felder P (Present), DPL (Descriptor Privilege Level) und A (Accessed) im Access-Byte und
- die Bits G (Granularity) und AVL (Available) im Flag-Feld.

Da das G-Bit und die Limit- und Base-Felder bereits früher besprochen worden sind, sollen zunächst die gemeinsamen Bit-Informationen im Access-Byte der beiden Deskriptoren diskutiert werden.

ZUR ERINNERUNG:
Das Access-Byte legt die Zugriffsrechte eines Segments auf ein anderes Segment fest.

3.6.1 Gemeinsamkeiten der „Access-Bytes“



'ACCESS-BYTE' DES DATEN-SEGMENT-DESKRIPTORS

'ACCESS BYTE' DES EXECUTABLE-SEGMENT-DESKRIPTORS

Present-Bit (P)
Dieses Bit ist gesetzt (P = 1), wenn das durch einen Deskriptor adressierte Segment im Speicher gegenwärtig ist. Im anderen Fall ist dieses Bit gelöscht (P = 0). Jeder Zugriff auf ein Segment, dessen Present-Bit gelöscht ist, stellt eine Ausnahme-Situation dar, die allerdings einem Betriebssystem für dynamische Anwendungen die Gelegenheit gibt, das Segment vom virtuellen Speicher zu laden.

Accessed-Bit (A)
Der Prozessor setzt dieses Bit (A = 1) immer dann, wenn ein Deskriptor in ein Segment-Register geladen wird. Damit A manipuliert werden kann, muß sich der Deskriptor im RAM-Bereich aufhalten, da ein EPROM-Deskriptor-Feld ***nicht*** beschreibbar ist. Das Betriebssystem hat die Möglichkeit, dieses Bit periodisch zu testen und zu löschen und kann dadurch die Frequenz des Segmentzugriffs verfolgen.
Dieses Bit zeigt auch an, ob ein Segment in den Hintergrund-Speicher ausgelagert werden muß, bevor der vom Segment besetzte RAM-Bereich wiederbenützt wird.

Descriptor Privilege Level (DPL)
Der Wert in diesem Feld definiert die Privilegebene des durch den Deskriptor adressierten Segments.

TABELLE DER INITIALISIERUNGSWERTE

DPL		SEGMENT-PRIVILEG-EBENE
Φ	Φ	Φ
Φ	1	1
1	Φ	2
1	1	3

Das DPL-Feld kann für ein statisches System oder für den residenten Teil eines dynamischen Systems vom „System Builder“ BLD386/486 initialisiert werden.
Wenn Segmente dynamisch geladen werden, übernehmen Prozeduren des Betriebssystems die Initialisierung von DPL.

3.7 Das „Access Byte“ des Executable-Segment-Deskriptors

'ACCESS-BYTE'

3.7.1 Das Readable-Bit (R)

Wenn dieses Bit gelöscht ist (R = 0), können Prozeduren in anderen Segmenten dieses Code-Segment ***nicht*** lesen. Dies bedeutet, daß der Inhalt dieses Segments nur ausgeführt werden kann und als „Execute Only“-Segment (Typ EO) spezifiziert ist.

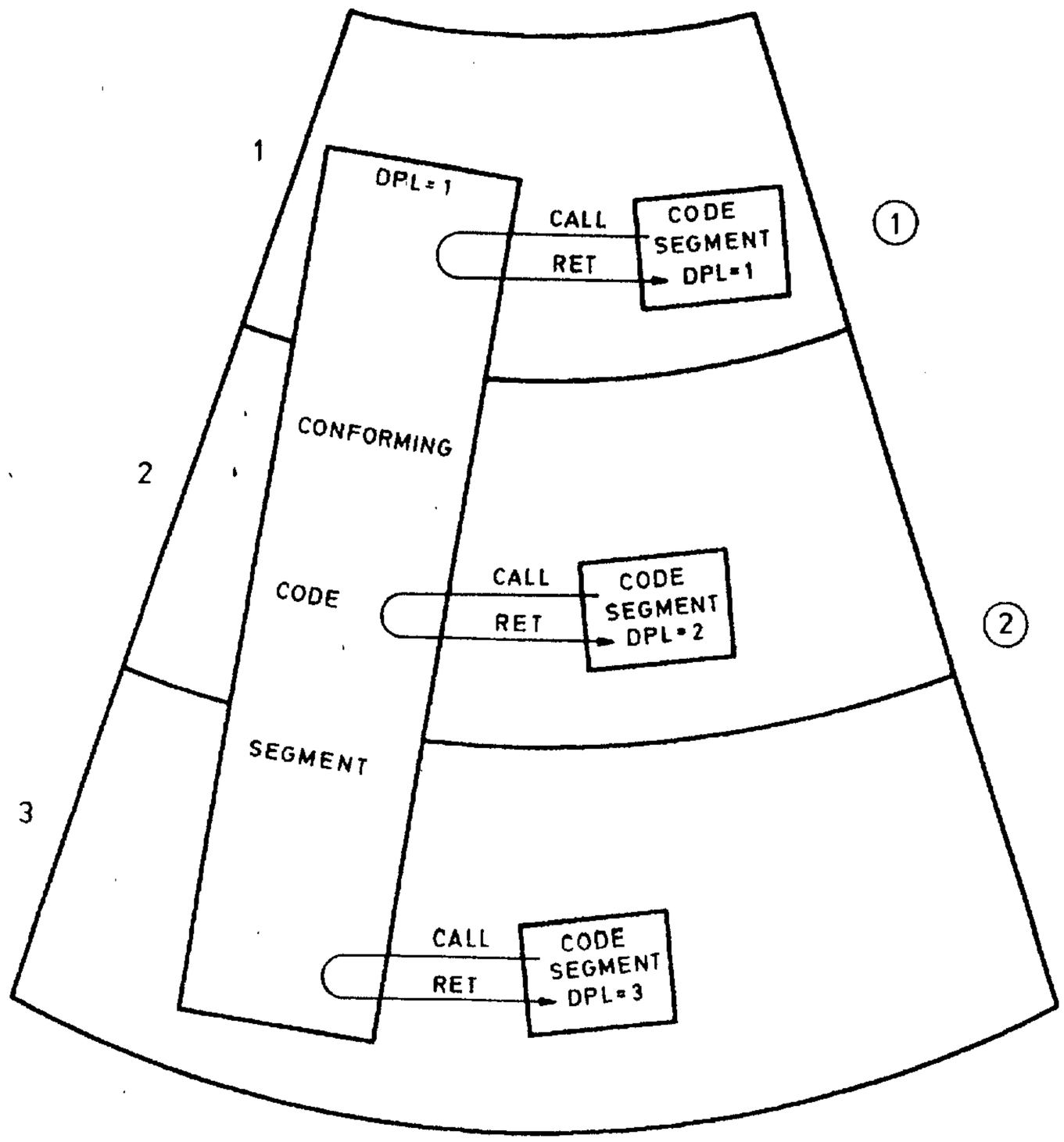
Sollte ein ausführbares Segment auch lesbare Daten (z. B. Konstanten) enthalten, muß das R-Bit gesetzt sein (R = 1). In diesem Fall ist das Segment als „Execute- and Readable“-Segment (Typ ER) spezifiziert.

3.7.2 Das Conforming-Bit (C)

Dieses Bit wird nur für „Execute Only“-Segment-Deskriptoren benützt. Ist das Conforming-Bit des aufgerufenen Segments gelöscht (C = 0), liegt die einfachere Situation vor. Die aufgerufene Prozedur hält sich in genau dem Segment auf, dessen Privilegebene durch das DPL-Feld des zugehörigen Deskriptors definiert ist.

Ist das Conforming-Bit des aufgerufenen Segments gesetzt (C = 1), wird in diesem Fall die aufgerufene Prozedur auf der gleichen Privilegebene wie die aufrufende Prozedur ausgeführt.

Die folgende Graphik soll die Arbeitsweise eines Conforming-Segments illustrieren.

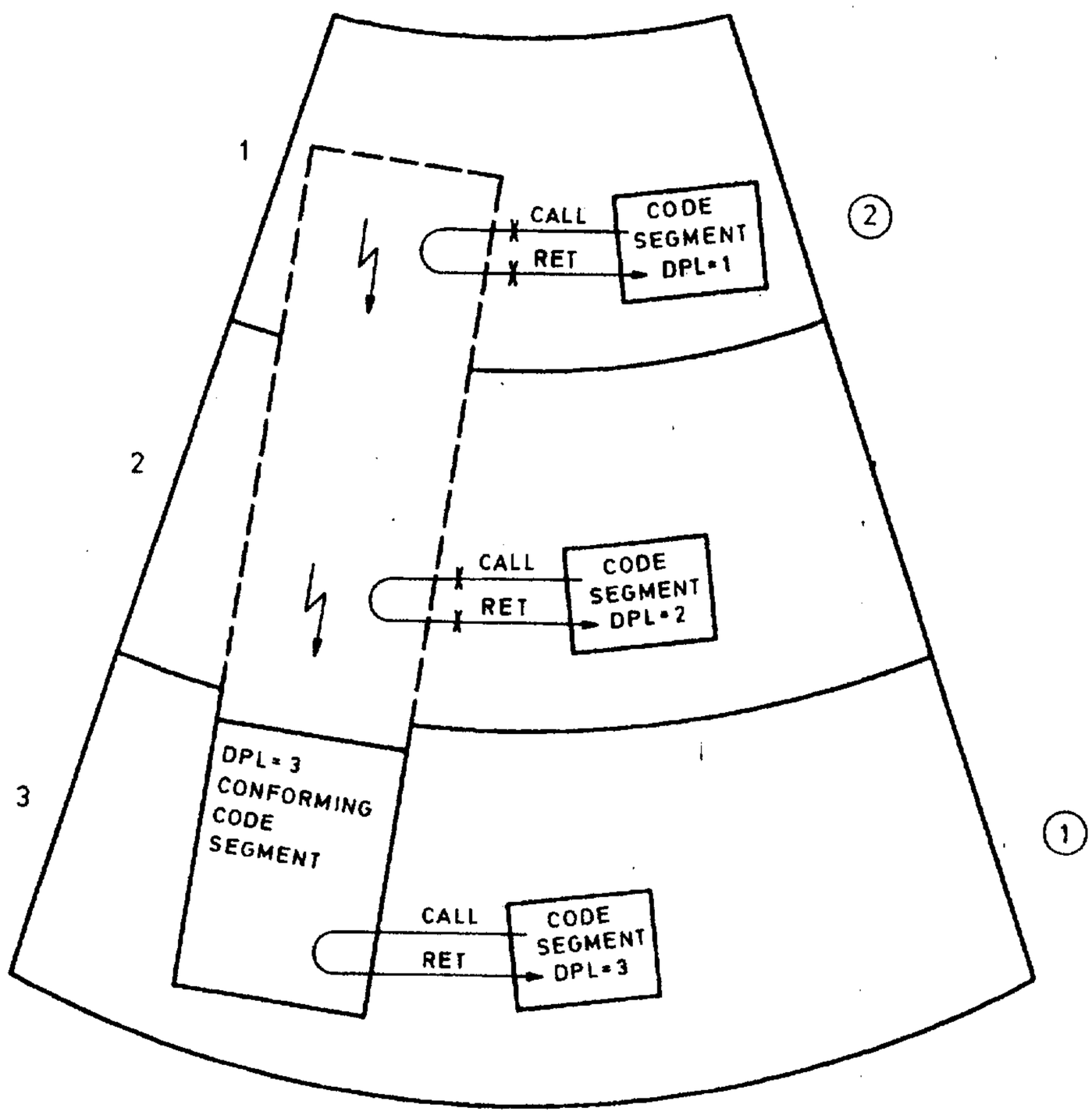


Es ist ersichtlich, daß sich das „Conforming Code"-Segment auf der Privilegebene 1 (DPL = 1) befindet. Die Privilegebene des Conforming-Segments ändert sich nicht, wenn sich das aufrufende Code-Segment auf der gleichen Ebene befindet (im Beispiel DPL = 1) ①.

Sind aber die Privilegebenen der aufrufenden Code-Segmente numerisch ***größer*** (im Beispiel DPL = 2 und DPL = 3) als die Ebenen des aufgerufenen Conforming-Segments, wird das Conforming-Segment auf diesen Ebenen ausgeführt ②.

Die Leistungsmerkmale eines Conforming-Segments können nicht ausgenützt werden, wenn die Privilegebene des aufrufenden Code-Segments numerisch ***kleiner*** ist als die Ebene des aufzurufenden Conforming-Segments.

Das folgende Bild soll dies verdeutlichen.



Es ist ersichtlich, daß sich das „Conforming Code“-Segment auf der Privilegebene 3 (DPL = 3) befindet. Seine Privilegebene bleibt erhalten, wenn es von einem Code-Segment auf der gleichen Ebene aufgerufen wird (im Beispiel DPL = 3) ①.

Der Versuch eines Zugriffs auf das Conforming-Segment durch die Code-Segmente in den Privilegebenen 1 und 2 stellt eine Ausnahme-Situation dar und ist daher nicht erlaubt ②.

Die Leistungsmerkmale der Conforming-Segmente sind immer dann nützlich, wenn bestimmte Prozeduren (z. B. mathematische Routinen) für andere Prozeduren, die auf niedrigeren Privilegebenen laufen, ***ohne*** Privilegwechsel verfügbar sein sollen.

3.8 Das „Access Byte“ des Daten-Segment-Deskriptors

3.8.1 Das Writable-Bit (W)

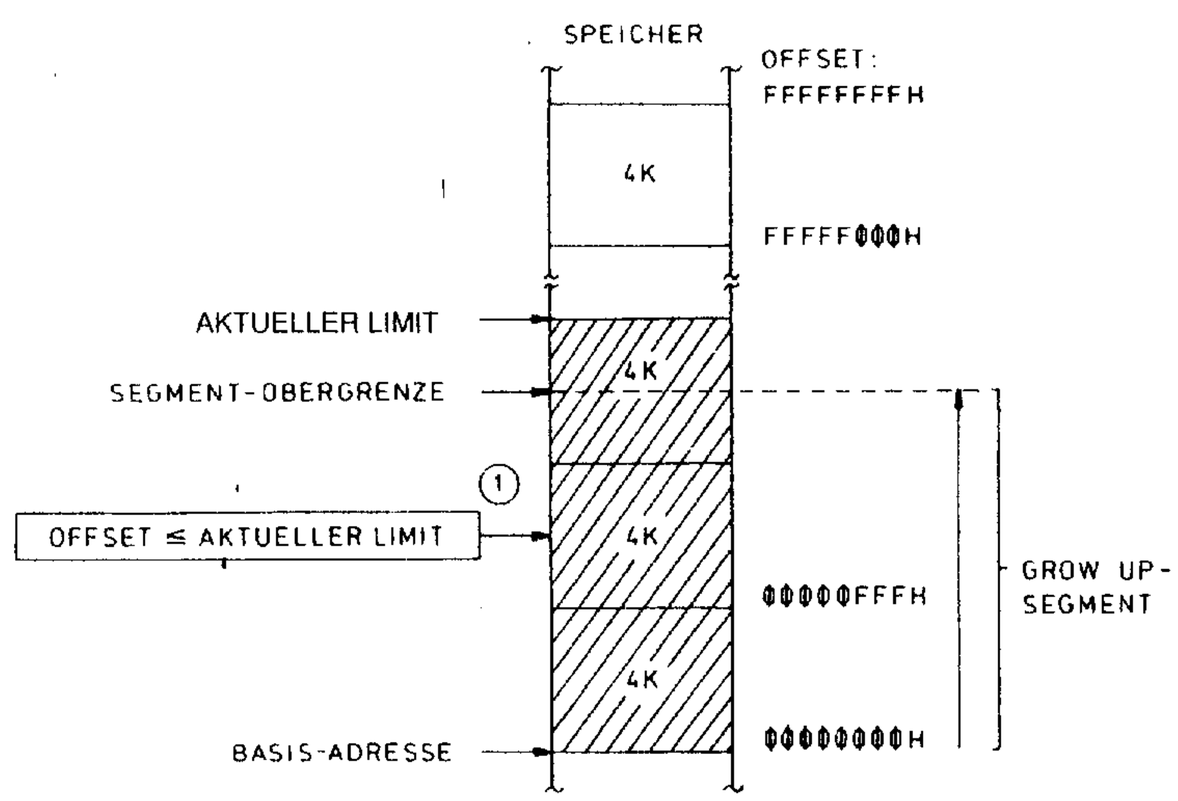
Wenn dieses Bit gesetzt ist (W = 1), hat die CPU die Erlaubnis, in das Segment zu schreiben. Soll das Segment vor einem Schreibversuch geschützt werden, muß dieses Bit gelöscht sein (W = 0). In diesem Fall handelt es sich um ein „Read Only“-Segment (Typ RO).

So sind Übersetzungs-Tabellen ein Beispiel für Daten, die durch ihren besonderen Anspruch auf Schutz in „Read Only"-Segmenten gespeichert werden.

3.8.2 Das Expansion-Direction-Bit (ED)

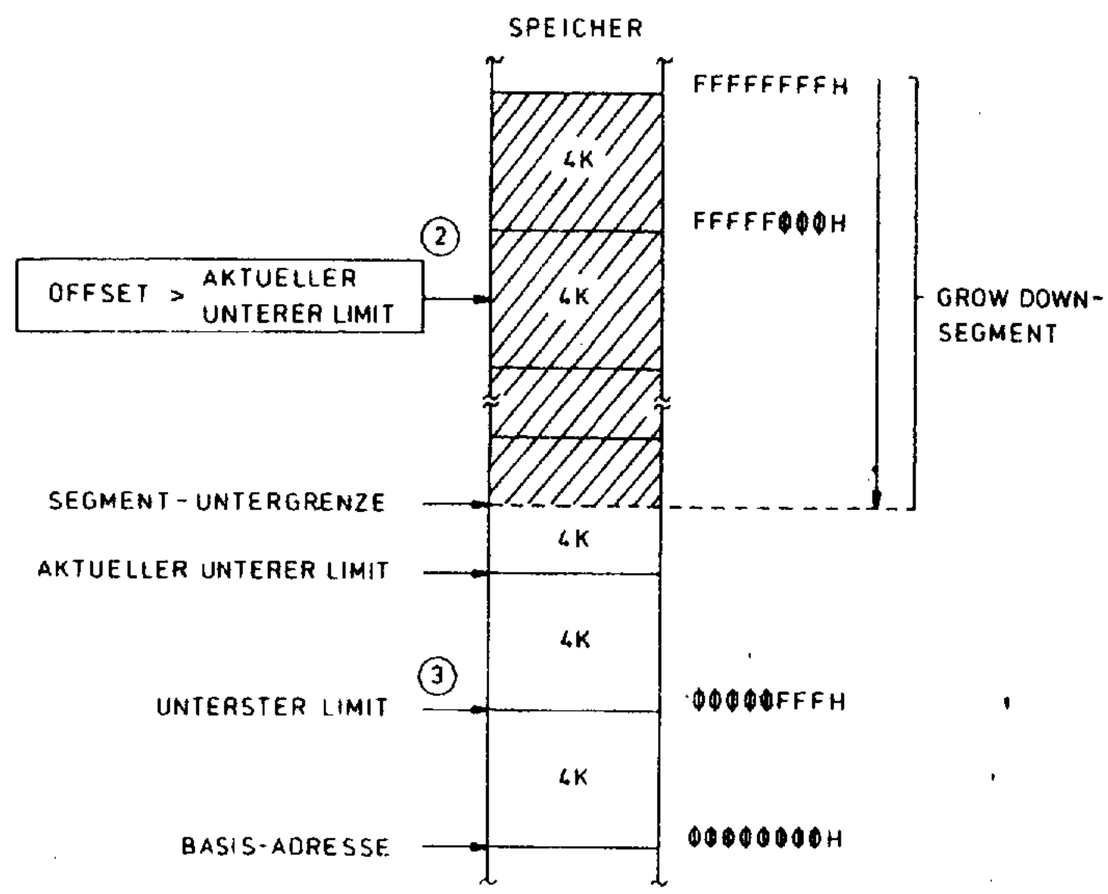
Daten-Segmente können sowohl Stack-Daten als auch andere Datenstrukturen enthalten. Ein „normales" Daten-Segment wird durch Offsetwerte adressiert, die innerhalb des Bereichs von 0 und der Größe des Segments liegen. Ist das ED-Bit gelöscht (ED = 0), sind Offsetwerte zwischen 0 bis einschließlich dem Segment-Obergrenzwert (Limit) erlaubt. Ist das Granularity-Bit gesetzt (G = 1), schiebt der Prozessor das 20-Bit-LIMIT-Feld im Daten-Segment-Deskriptor um 12 Bits nach links und besetzt anschließend die niederwertigen 12 Stellen mit 1-Bits. Bei ED = 0 kann dann der aktuelle Limit-Wert im Bereich von 0FFFH (2^{12}-1 oder 4 KBytes) ... 0FFFFFFFH (2^{32}-1 oder 4 Gigabytes) liegen.

Der Wert 0 im ED-Feld gibt an, daß das Daten-Segment, beginnend bei seiner Basisadresse, in Richtung Segment-Obergrenze wächst. Daher wird ein solches Segment als ***„Grow UP"***-Segment bezeichnet. Alle Offsets in „Grow Up"-Segmenten müssen ***kleiner*** oder ***gleich*** dem aktuellen Limit-Wert sein. Dies verdeutlicht das folgende Bild ①.



Während sich die meisten Datenstrukturen in Richtung höherer Adressen ausbreiten, expandieren Stack-Daten in Richtung niedrigerer Adressen.
Damit das Verhalten des Stacks im Daten-Segment berücksichtigt werden kann, muß das ED-Bit gesetzt sein. Ein mit ED = 1 gekennzeichnetes Daten-Segment wird als ***„Grow Down"***-Segment bezeichnet.

Alle Offsets, die auf ein solches Segment Bezug nehmen, müssen eindeutig ***größer*** sein als der aktuelle untere Limit-Wert. Dies verdeutlicht das folgende Bild ②.



Ist das Granularity-Bit gesetzt (G = 1), schiebt ***auch*** hier der Prozessor das 20-Bit-LIMIT-Feld im Daten-Segment-Deskriptor um 12 Bits nach links und besetzt anschließend die niederwertigen 12 Stellen mit 1-Bits.

Bei ED = 1 kann dann der aktuelle untere Limit-Wert im Bereich von 0FFFFFFFFH (2^{32}-1 oder 4 Gigabytes) ... 0FFFH (2^{12}-1 oder 4 KBytes) liegen.

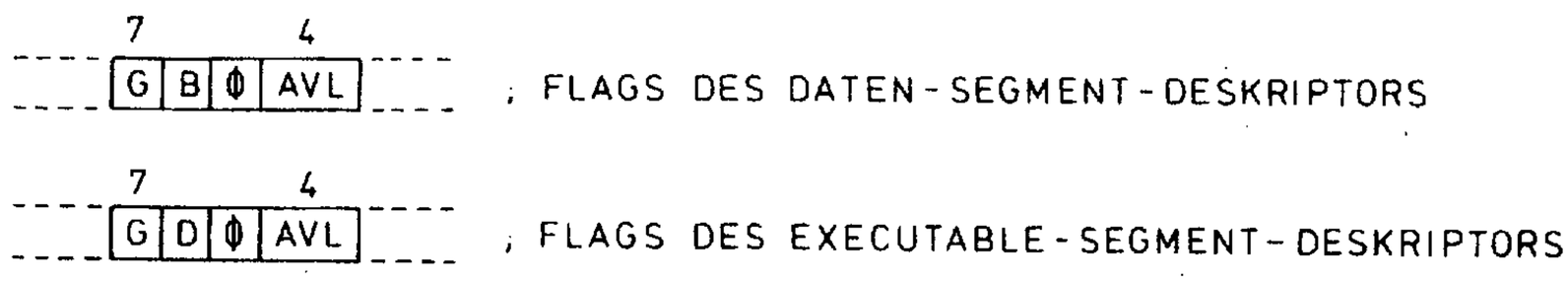
Exakt dieser Bereich steht auch den Offsetwerten zur Verfügung, da der ***unterste*** Limit-Wert 0FFFH wegen G = 1 ***niemals*** unterschritten werden kann ③.

Dies bedeutet, daß ein „Grow Down"-Segment die maximale Größe von 4 Gigabytes – 4 KBytes hat. Die volle Größe eines Daten-Segments (4 Gigabytes) ist also nur dann zu bekommen, wenn es als „Grow Up"-Daten-Segment deklariert ist.

Es liegt daher im Ermessen des System-Programmierers, ob er den Stack in einem „Grow Up"- oder einem „Grow Down"-Daten-Segment aufbewahrt.

HINWEIS:
Ein Daten-Segment, das einen Stack enthält, wird immer durch einen ***SS-Selektor*** ausgewählt. Daher wird ein solches Segment auch als Stack-Segment bezeichnet.

3.9 Das FLAG-Feld im Daten- und Executable-Segment-Deskriptor



Wie die Formate der beiden FLAG-Felder zeigen, haben einige Bit-Positionen identische Bedeutung. So enthalten beide Felder das G-Bit (Granularity) und das AVL-Bit (AVAILABLE). Während das G-Bit die Krönung des Segments bestimmt, steht das AVL-Bit dem System-Programmierer für statistische Zwecke zur Verfügung. Dieses Bit wird vom „System Builder" BLD386/486 auf 0 gesetzt.

Das im FLAG-Feld des Executable-Segment-Deskriptors enthaltene D-Bit (Default Operation Size) wurde bereits besprochen. Es bestimmt die Länge der in einem ***Code***-Segment benutzten Operanden und effektiven Adressen.

ZUR ERINNERUNG:

- Ist ***D = 1***, sind 32-Bit-Operanden und 32-Bit-Adressierungsarten vorausgesetzt.
- Ist ***D = 0***, sind 16-Bit-Operanden und 16-Bit-Adressierungsarten vorausgesetzt.

Wenn also D = 0 ist, können auch 80286-Code-Segmente vom 80486 ausgeführt werden.

3.9.1 Das Big-Bit (B)

Wie im vorherigen Abschnitt gezeigt, können Stacks entweder in

- „Grow Up"-Daten-Segmenten oder in
- „Grow Down"-Daten-Segmenten

untergebracht sein. Dies bedeutet, daß der augenblickliche Stack eines Programms von einem Daten-Segment-Deskriptor beschrieben wird.

Ist das B-Bit in diesem Deskriptor gelöscht (B = 0), hält sich der Stack in einem USE16-Segment auf. Ist das B-Bit in diesem Deskriptor besetzt (B = 1), hält sich der Stack in einem USE32-Segment auf.

Greift nun der Prozessor ***implizit*** auf den Stack zu, teilt das B-Bit im zugehörigen Daten-Segment-Deskriptor dem Prozessor mit, ob er das 16-Bit-Stackpointer-Register SP (bei B = 0) oder das 32-Bit-Stackpointer-Register ESP (bei B = 1) benutzen soll. Das B-Bit bestimmt also bei impliziten Stack-Referenzen die Größe des Stackpointer-Registers. Solche Stack-Referenzen werden generell durch

- Interrupts
- Ausnahme-Situationen
- Befehlen wie PUSH, POP, CALL und RET

veranlaßt.

Alle Befehle, die den Stack benutzen, sind durch ein sogenanntes ***„Stack-Size-Attribut"*** von 16 Bits oder 32 Bits gekennzeichnet. Da sich der Stack in einem Daten-Segment aufhält, ist das Stack-Size-Attribut mit dem USE-Attribut dieses Daten-Segments ***identisch***.

Befehle mit einem Stack-Size-Attribut von 16 benützen das 16-Bit-Stackpointer-Register SP. Befehle mit einem Stack-Size-Attribut von 32 benützen das 32-Bit-Stackpointer-Register ESP.

Die Wirkung des B-Bits soll nun am Beispiel des PUSH-Befehls erläutert werden. Dieser Befehl benützt entweder eine Operanden-Größe von 16 Bits (z. B. PUSH AX) oder eine Operanden-Größe von 32 Bits (z. B. PUSH EBX).

Wie die folgenden Programm-Fragmente zeigen, enthält das USE16-Code-Segment (D-Bit = 0) die Befehle PUSH AX und PUSH EBX und das USE32-Code-Segment (D-Bit = 1) die Befehle PUSH ECX und PUSH DX.

```
CODE SEGMENT USE16

PUSH AX      ; keine „Operand Size Prefix“
PUSH EBX     ; mit „Operand Size Prefix“

CODE ENDS
```

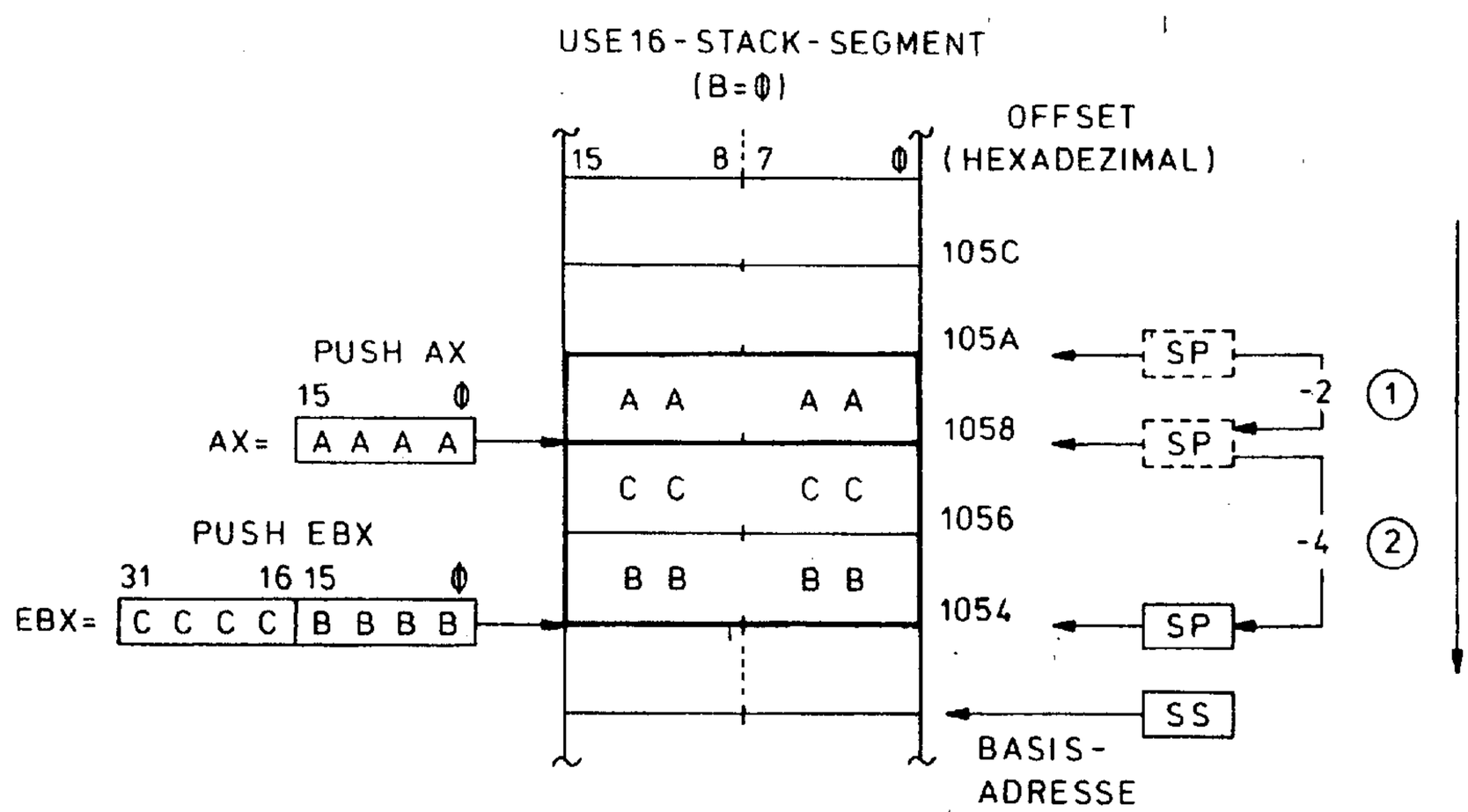
```
CODE_1 SEGMENT USE32

PUSH ECX     ; ohne „Operand Size Prefix“
PUSH DX      ; mit „Operand Size Prefix“

CODE_1 ENDS
```

1. B-Bit = 0:

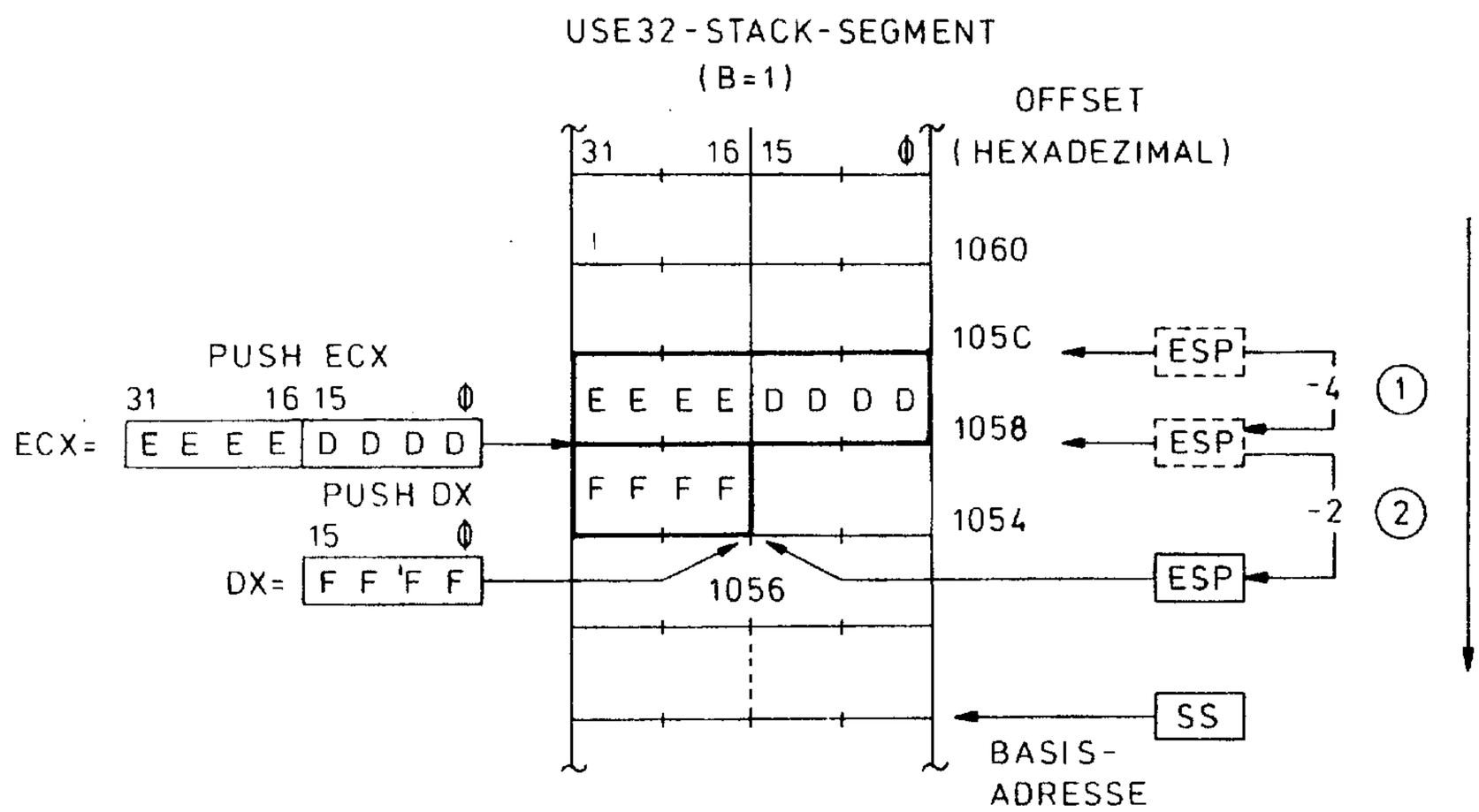
Wenn das B-Bit = 0 ist, benützen beide Befehle ein USE16-Stack-Segment. Das Stack-Size-Attribut beider Befehle ist daher 16. Dies bedeutet, daß der Prozessor für die Befehlsausführungen das 16-Bit-Stackpointer-Register SP benützt.



In der angegebenen Konfiguration zeigt SP zunächst auf die Position 105AH im augenblicklichen Stack-Segment (Top of Stack). Der Befehl PUSH AX bewirkt, daß ***SP mit 2*** dekrementiert und anschließend AX an die Position 1058H geschrieben wird (Word Assignment) ①. Der folgende Befehl PUSH EBX bewirkt, daß ***SP mit 4*** dekrementiert und anschließend EBX an die Position 1054H geschrieben wird (Doubleword Assignment) ②.

2. B-Bit = 1:

Wenn das B-Bit = 1 ist, benützen beide Befehle ein USE32-Stack-Segment. Das Stack-Size-Attribut beider Befehle ist daher 32. Dies bedeutet, daß der Prozessor für die Befehlsausführungen das 32-Bit-Stackpointer-Register ESP benützt.

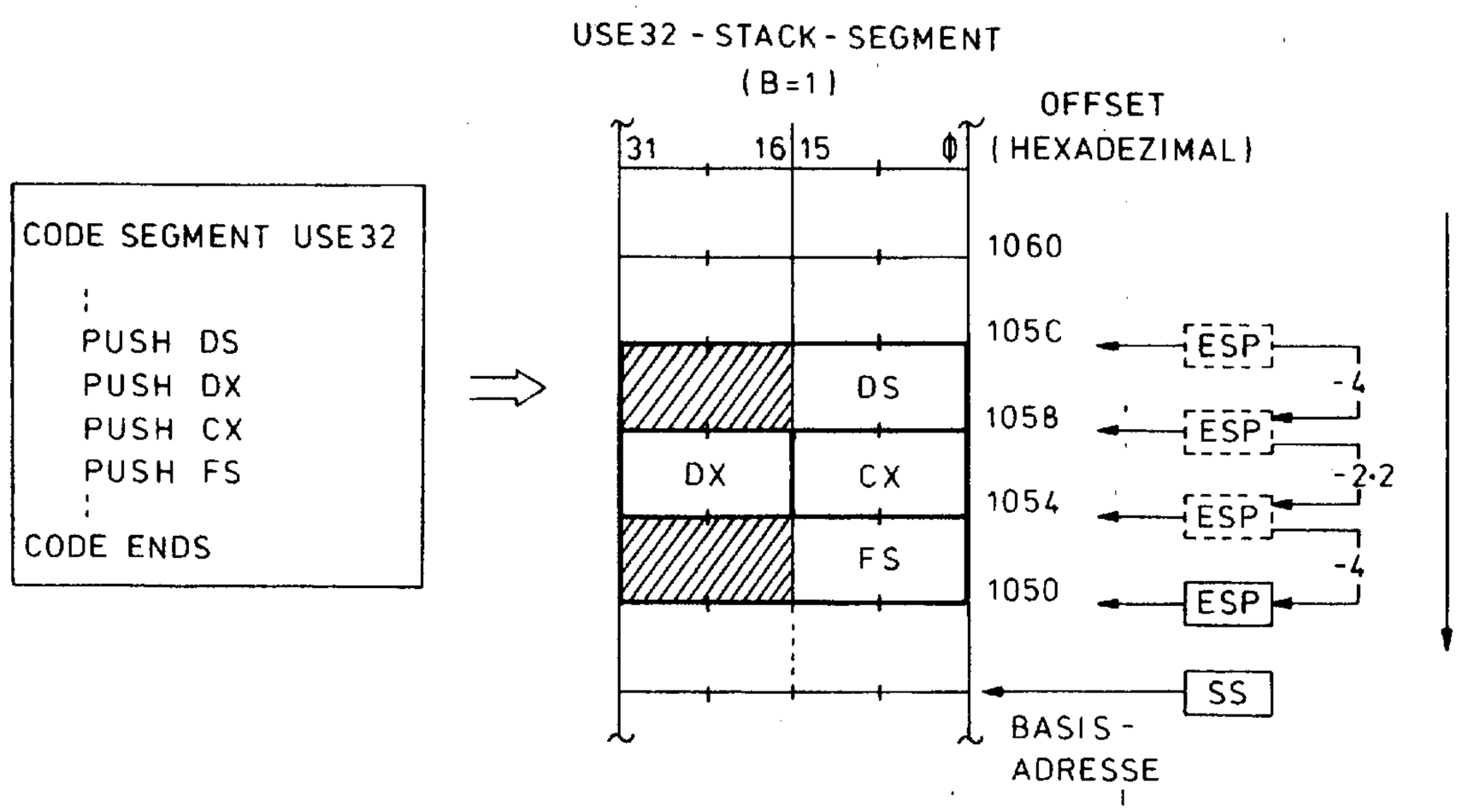


In der angegebenen Konfiguration zeigt ESP zunächst auf die Position 105CH im augenblicklichen Stack-Segment (Top of Stack). Der Befehl PUSH ECX bewirkt, daß ***ESP mit 4*** dekrementiert und anschließend ECX an die Position 1058H geschrieben wird (Doubleword Assignment) ①. Der folgende Befehl PUSH DX bewirkt, daß ***ESP mit 2*** dekrementiert und anschließend DX an die Position 1056 H geschrieben wird (Word Assignment) ②.

Generell gilt, daß der Prozessor das Stackpointer-Register SP oder ESP mit 4 dekrementiert oder inkrementiert, wenn ein PUSH- oder POP-Befehl einen Doubleword-Operanden benützt. Im anderen Fall, wenn sich ein PUSH- oder POP-Befehl auf einen Word-Operanden bezieht, dekrementiert oder inkrementiert der Prozessor das Stackpointer-Register SP oder ESP mit 2.

Von diesen Regeln ausgenommen sind allerdings PUSH- und POP-Operationen, die mit den 16-Bit-Segment-Selektoren DS, ES, FS, GS und SS in USE32-Stack-Segmenten durchgeführt werden. In einer solchen Umgebung werden diese Selektoren als ***„Doublewords“*** behandelt. Dabei hält sich der Selektor in den niederwertigen 16 Bits des Doublewords auf, während die höherwertigen 16 Bits ***undefiniert*** sind. Dies bedeutet, daß der Prozessor ESP mit 4 dekrementiert bzw. mit 4 inkrementiert, wenn er die Befehle PUSH SELEKTOR bzw. POP SELEKTOR ausführt.

Welches Stack-Layout sich ergibt, wenn „normale“ Word-Operanden mit Segment-Selektoren im USE32-Stack-Segment gemischt werden, verdeutlicht das folgende Beispiel:



Es zeigt, daß der Prozessor nach Ausführung der Befehle PUSH DS (ESP-4) und PUSH FS (ESP-4) die 16 Bit-Selektoren DS und FS in „Doublewords" aufbewahrt, während er die „normalen" Word-Register DX und CX nach Ausführung der Befehle PUSH DX (ESP-2) und PUSH CX (ESP-2) in ihrem natürlichen 16 Bit-Format speichert.

3.10 Limit-Überprüfung

Um zu verhindern, daß Programme auf Regionen außerhalb eines Segments zugreifen, benützt der Prozessor das Limit-Feld im Segment-Deskriptor. Dabei ist die Interpretation des Limit-Feldes abhängig vom Granularity-Bit G.

Für Daten-Segmente benützt der Prozessor zur Interpretation des Limit-Feldes noch ***zusätzlich*** das ED-Bit (Expansion Direction) im Access-Byte und das B-Bit (Big) im FLAG-Feld des Segment-Deskriptors.

Die folgende Tabelle zeigt sinnvolle Kombinationen der G-, ED- und B-Bits und ihre Auswirkungen auf die Segment-Grenzen und ihre Größen.

Fall	**1**	**2**	**3**	**4**
Expansion Direction-Bit ED	0 (Grow UP)	0 (Grow UP)	1 (Grow Down)	1 (Grow Down)
Granularity-Bit G	0	1	0	1
Big-Bit B	0	1	0	1
Untere Grenze ist:				
0	x	x		
LIMIT +1			x	
Shift (LIMIT,12,1) +1				x
Obere Grenze ist:				
LIMIT	x			
Shift (LIMIT,12,1)		x		
64K-1			x	
4G-1				x
Maximale Segment-Größe ist:				
64K	x			
64K-1			x	
4G-4K		x		x
4G				
Minimale Segment-Größe ist:				
0	x		x	
4K		x		x

HINWEIS:
Shift (LIMIT,12,1) = Schiebe LIMIT um 12 Bits nach links und besetze anschließend die niederwertigen 12 Stellen mit 1-Bits.

Für alle Segment-Typen, mit Ausnahme von „Grow Down"-Segmenten, ist der Limit-Wert um 1 kleiner als die Größe (gemessen in Bytes) des Segments.

Dabei erkennt der Prozessor in folgenden Fällen eine sogenannte „allgemeine Schutzverletzung“:

- beim Versuch, auf ein Speicher-***Byte*** bei einer Adresse > Limit zuzugreifen.
- beim Versuch, auf ein Speicher-***Word*** bei einer Adresse > = Limit zuzugreifen.
- beim Versuch, auf ein Speicher-***Doubleword*** bei einer Adresse < = (Limit -2) zuzugreifen.

Für „Grow Down“-Daten-Segmente hat das Limit-Feld die gleiche Schutzfunktion, es wird aber anders interpretiert. Hier liegen die gültigen Adressen im Bereich

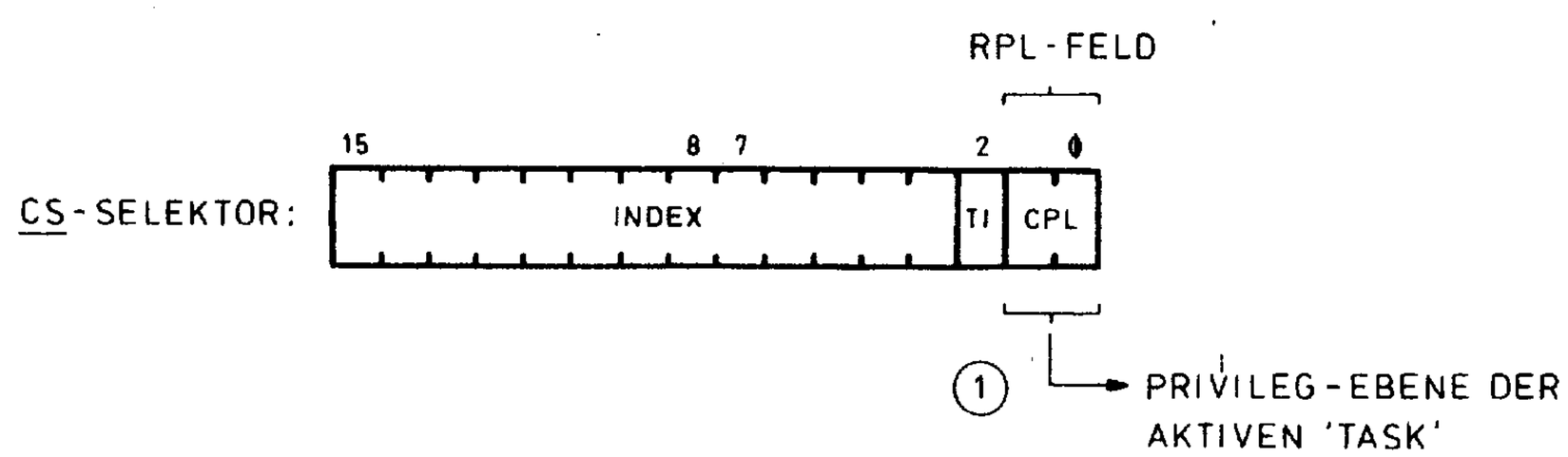
- (Limit + 1) ... (64 KBytes) bei B-Bit = 0
- oder (Limit + 1) ... (4 Gigabytes) bei B-Bit = 1.

Ein „Grow Down“-Segment hat also dann seine maximale Größe, wenn der Limit-Wert 0 ist.

3.11 Ablaufsteuerung eines Programms ohne Privileg-Wechsel

3.11.1 Privilegebene einer Task

Jede Task ist zu einem bestimmten Zeitpunkt nur auf ***einer*** Privilegebene aktiv und führt ein Code-Segment dieser Ebene aus.
Das Privileg-Attribut einer Task ist im RPL-Feld (Requestor's Privilege Level) des ***CS***-Selektors gespeichert und wird als „Current Privilege Level“ (CPL) bezeichnet. CPL spezifiziert die Privilegebene der aktiven Task ①.



Der Zugriff auf ein Code-Segment erfolgt immer dann, wenn das CS-Register mit einem Selektor geladen wird. Innerhalb eines ausgewählten Segments kann der Ablauf des Programms durch einen der folgenden Befehle gesteuert werden:

- **JMP Offset**
- **CALL Offset**
- **RET**; (der Offset wird dem Stack entnommen)

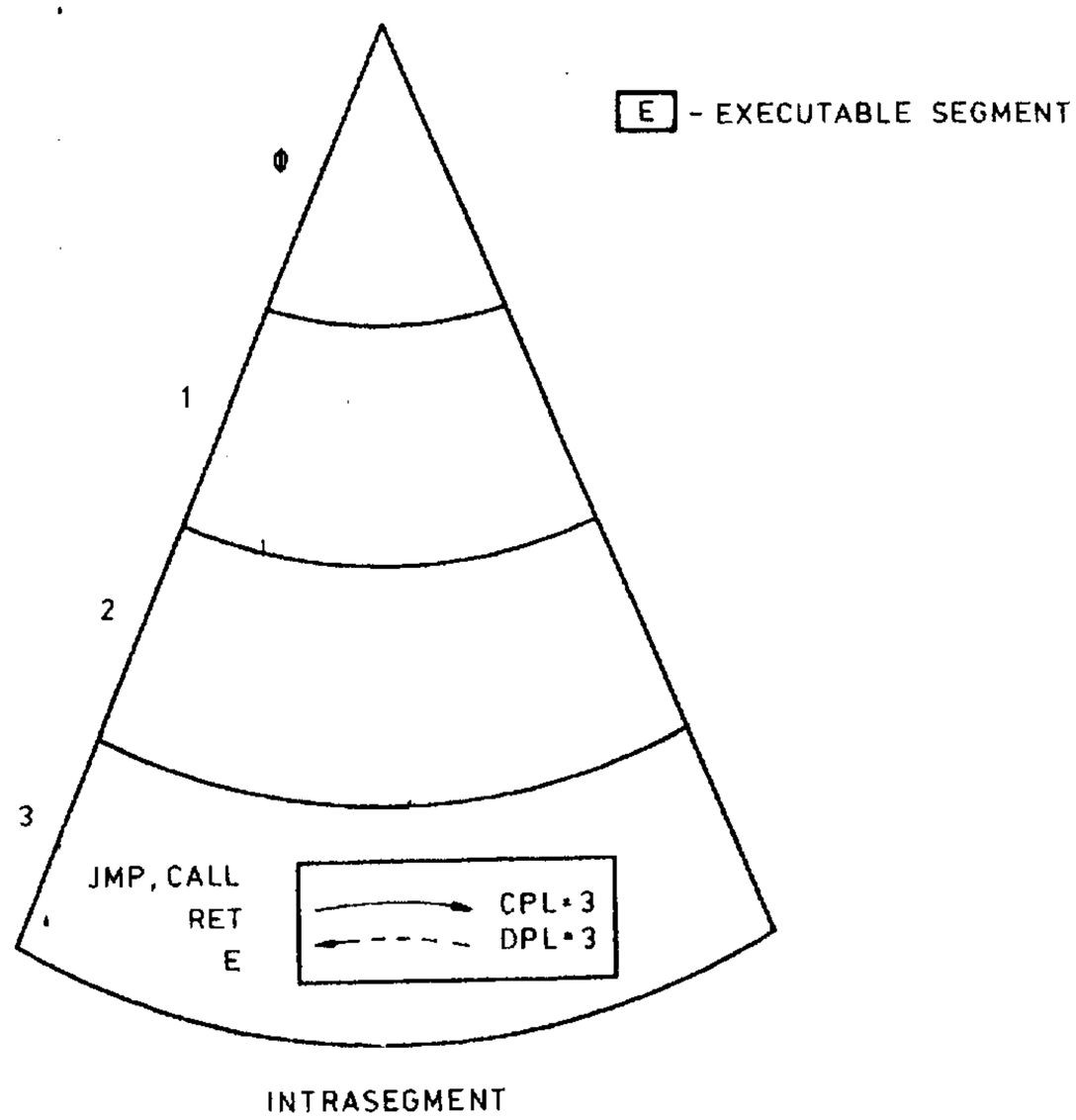
Da diese Befehle nur die Offset-Komponente („Instruction Pointer“ EIP) des virtuellen Adreßzeigers verändern, werden sie als Intra-Segment-Steuerbefehle bezeichnet.

Ist es notwendig, daß während der Ausführung eines Programms von einem Code-Segment auf ein anderes gewechselt werden muß, ist dies durch einen der folgenden Befehle realisierbar:

- **JMP Selektor: Offset**
- **CALL Selektor: Offset**
- **RET**; (der Selektor und der Offset werden dem Stack entnommen)

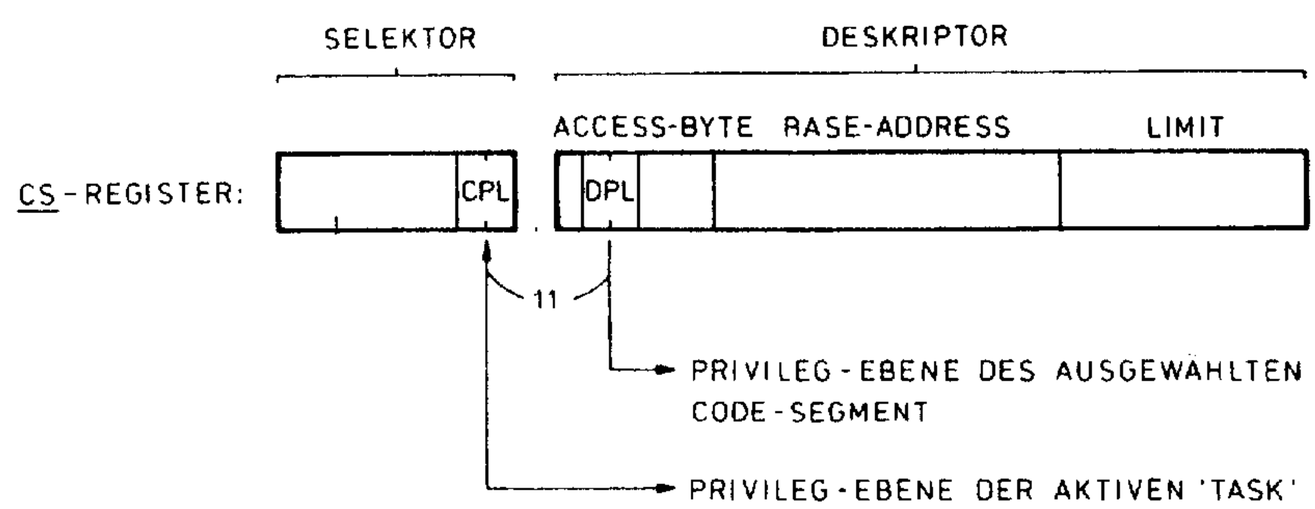
Da diese Befehle sowohl die Selektor-Komponente (Register CS) als auch die Offset-Komponente („Instruction Pointer" EIP) des virtuellen Adreßzeigers verändern, werden sie als Inter-Segment-Steuerbefehle bezeichnet.

Das folgende Bild illustriert den ***Intra***-Segment-Ablauf eines Programms auf der Privilegebene 3.

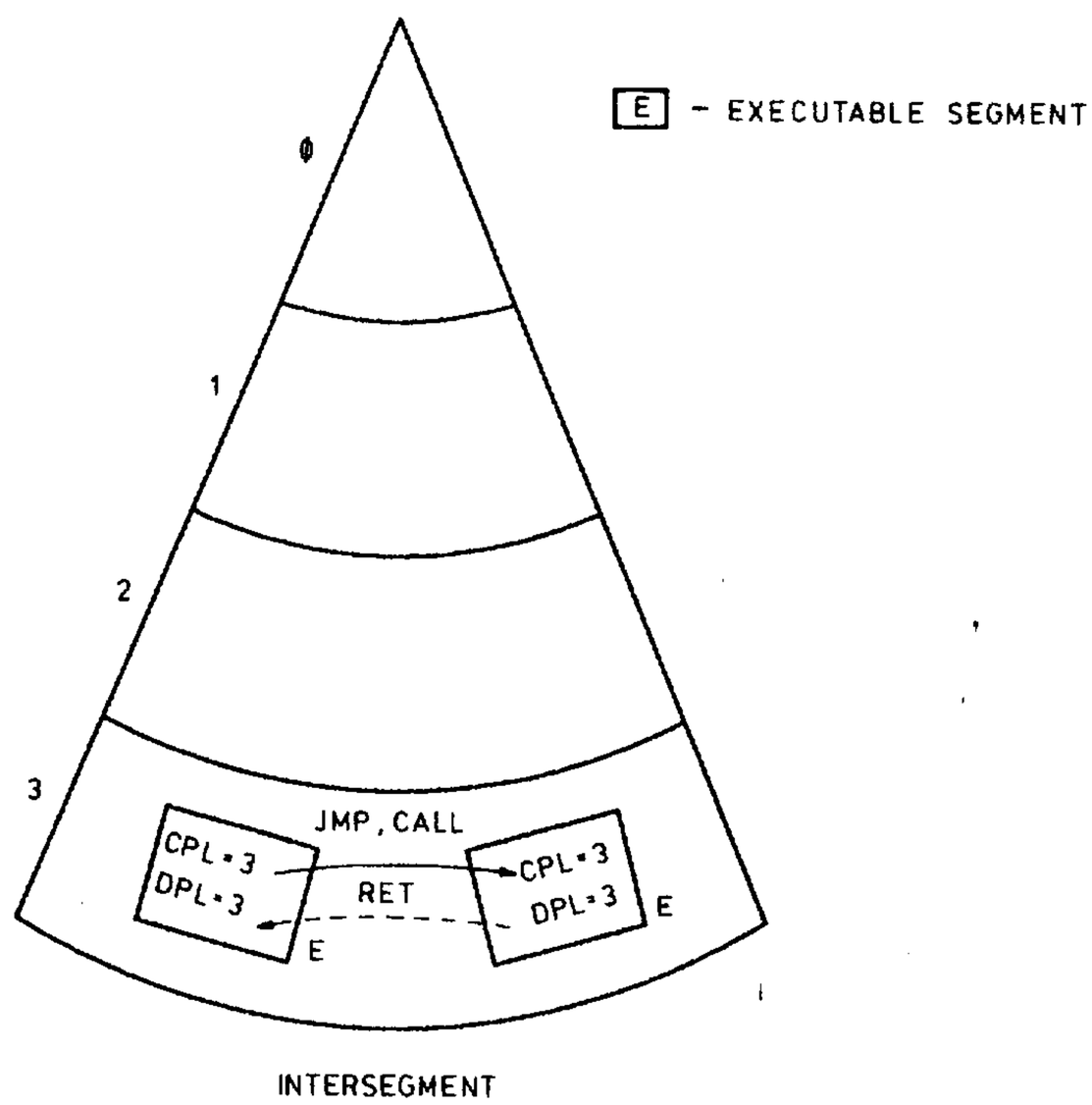


Der Selektor wählt den Deskriptor eines ausführbaren Segments aus. Da die Privilegebene (DPL) im Deskriptor dieses Segments den Wert 3 hat, wird das CPL-Feld des CS-Selektors mit diesem Wert initialisiert. Dadurch spezifiziert CPL die Privilegebene (3) der aktiven Task.

Das folgende Bild verdeutlicht diesen Mechanismus:



Das folgende Bild illustriert den ***Inter***-Segment-Ablauf eines Programms auf der Privilegebene 3.



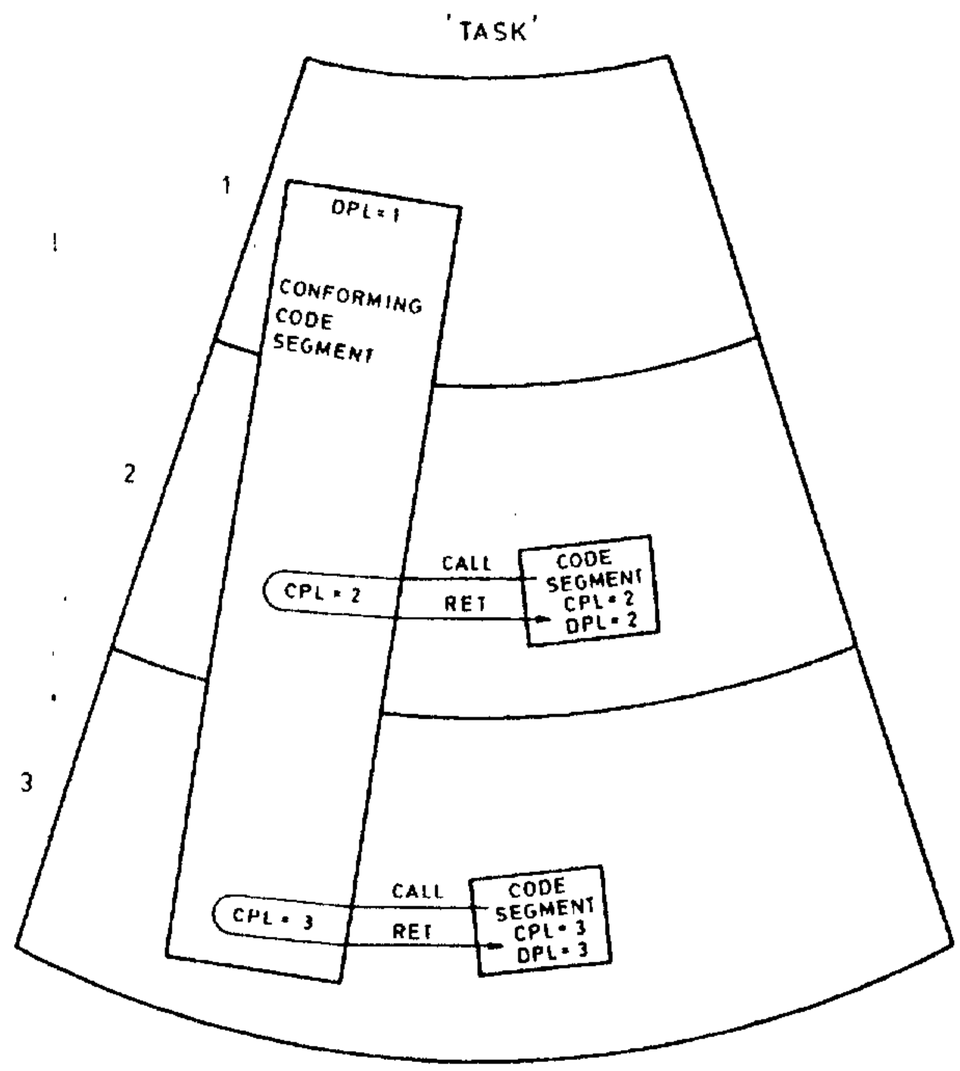
Da der Segment-Wechsel auf der ***gleichen*** Privilegebene erfolgt, wird ein solches Programm als „intralevel-gesteuert" bezeichnet. Der Segment-Wechsel wird eingeleitet, indem der CS-Selektor neu initialisiert wird. Da die Privilegebene (DPL) des Zielsegments mit der des aufrufenden Segments identisch ist, ändert sich das CPL-Feld im neuen CS-Selektor nicht. Die Privilegebene der aktiven Task bleibt daher erhalten.

Jede Task kann Segmente auf der ersten, zweiten, dritten oder vierten Ebene enthalten, die alle zu passenden Zeiten benützt werden können. Da eine Task jeweils nur auf ***einer*** Privilegebene aktiv ist, hat der Übergang auf ein Code-Segment in einer anderen Ebene zur Folge, daß sich die Privilegebene der Task ändert. Solche Übergänge unterliegen strengen Regeln, die sorgfältig eingehalten werden müssen.

Eine „normale" Privileg-Anforderung gehorcht diesen Regeln automatisch, da die augenblickliche Privilegebene (CPL) des anfordernden JMP- oder CALL-Befehls mit der Privilegebene (DPL) des aufgerufenen Segments identisch ist (DPL = CPL).

Wie bereits früher erwähnt, können Code-Segmente in privilegierteren Ebenen nur über sogenannte Gates erreicht werden. Eine Ausnahme stellen allerdings Conforming-Code-Segmente dar. Hier darf die Privilegebene des angeforderten Conforming-Segments (DPL) numerisch kleiner sein als CPL des aufrufenden Segments (DPL $\leqslant$ CPL).
Die Task führt das Conforming-Segment auf den jeweiligen Privilegebenen der aufrufenden Code-Segmente aus.

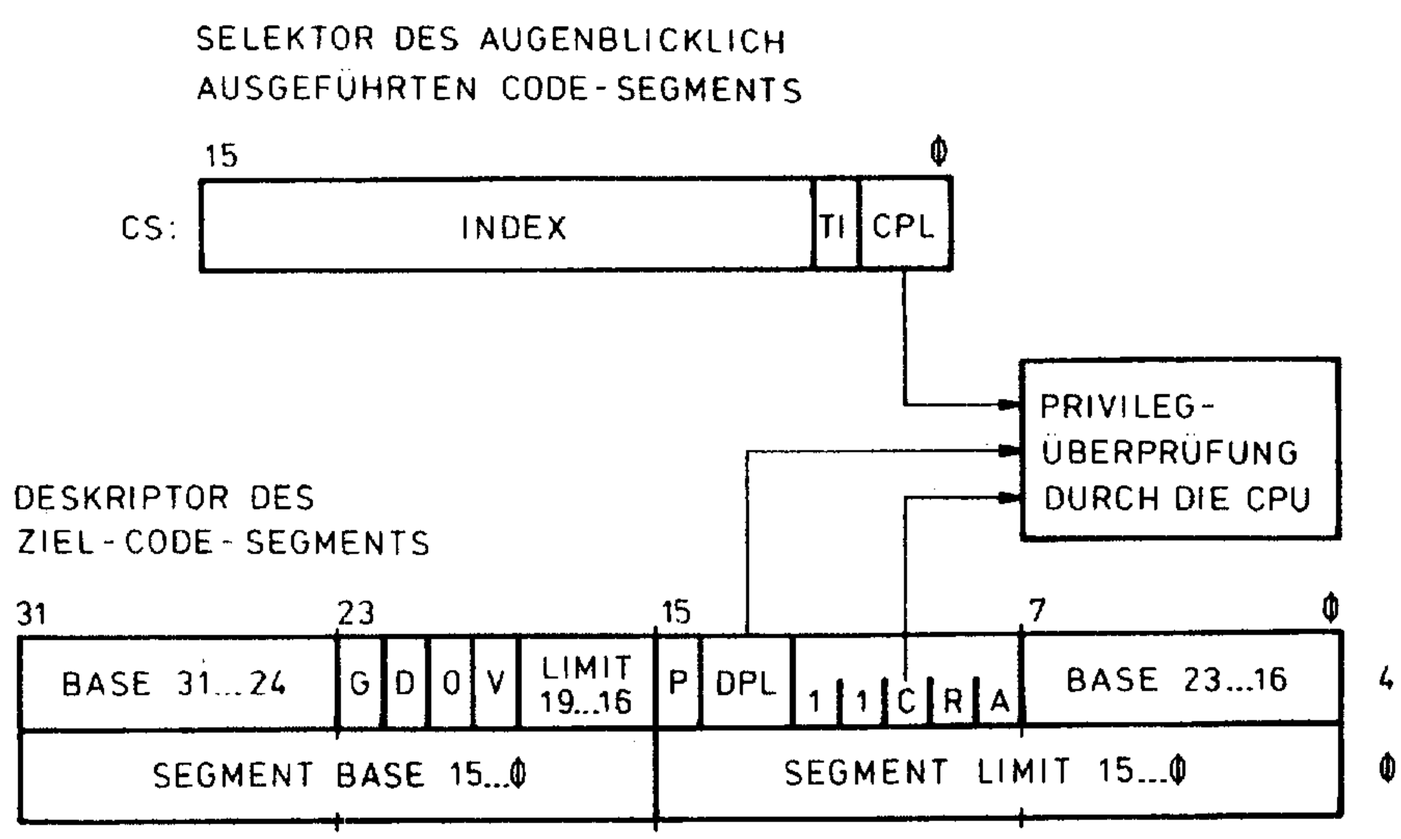
Das folgende Bild soll diese Situationen verdeutlichen:



Es ist ersichtlich, daß sich das Conforming-Segment auf der Privilegebene 1 (DPL = 1) befindet, die Task aber dieses Segment auf den Ebenen 2 (CPL = 2) oder 3 (CPL = 3) ausführt.

Die Privileg-Überprüfungen, die der Prozessor bei einem ***Intra***-Segment- oder einem ***Inter***-Segment-Programmablauf durchführt, zeigt die folgende Zusammenfassung. Wie das angegebene Bild erkennen läßt, kontrolliert dabei die CPU

- die augenblickliche Privileg-Ebene CPL der Task,
- die Deskriptor-Privileg-Ebene DPL des Ziel-Code-Segments und
- das Conforming-Bit C im Deskriptor des Ziel-Code-Segments.



Einen ***direkten CALL-, JMP- oder RET-Befehl*** auf ein anderes Segment erlaubt der Prozessor nur, wenn die folgenden Privileg-Regeln eingehalten werden:

1) Die Deskriptor-Privileg-Ebene DPL ist ***gleich*** der augenblicklichen Privileg-Ebene CPL der Task (DPL = CPL).
2) Das Conforming-Bit im Deskriptor des Ziel-Code-Segments ist ***gesetzt*** (C = 1) und DPL des Ziel-Code-Segments ist ***kleiner*** oder ***gleich*** der augenblicklichen Privileg-Ebene CPL der Task (DPL ≤ CPL).

3.11.2 Programmierbeispiel: Intra-Level-Steuerung

Das folgende Programm-Fragment zeigt, wie ohne Privileg-Wechsel (Intra-Level) ein Programmablauf entweder ***intra***segment oder ***inter***segment gesteuert werden kann.

```
LOC       OBJ                LINE    SOURCE

                              1              NAME      B9
                              2
--------                      3      CODE    SEGMENT EO USE32     ;Nur ausfuehrbares Segment mit
                              4                                   ;Basis-Adresse=1000H und DPL=3
                              5                                   ;(Festlegung durch BLD386/486)
                              6              ;
                              7              ;                    ;Normale Befehlssequenz
                              8              ;
00000000  E800000000          9              CALL      INTRA      ;Prozeduraufruf intrasegment-direkt
                             10              ;
                             11              ;                    ;Normale Befehlssequenz
00000005                     12      INTRA   PROC      NEAR ;****************
                             13              ;
                             14              ;                    ;Prozedurkoerper
                             15              ;
00000005  C3                 16              RET
00000006                     17      INTRA   ENDP            ;****************
                             18              ;
                             19              ;                    ;Normale Befehlssequenz
                             20              ;
00000006  9A00000000----  R  21              CALL FAR PTR INTER   ;Prozeduraufruf intersegment-direkt
                             22              ;
                             23              ;                    ;Normale Befehlssequenz
--------                     24      CODE    ENDS
                             25
                             26
--------                     27      CODE1   SEGMENT EO USE32     ;Nur ausfuehrbares Segment mit
                             28                                   ;Basisadresse=2000H und DPL=3
                             29                                   ;(Festlegung durch BLD386/486)
                             30              ;
                             31              ;                    ;Normale Befehlssequenz
                             32              ;
00000000                     33      INTER   PROC      FAR  ;****************
                             34              ;
                             35              ;                    ;Prozedurkoerper
                             36              ;
00000000  CB                 37              RET
00000001                     38      INTER   ENDP            ;****************
                             39              ;
                             40              ;                    ;Normale Befehlssequenz
--------                     41      CODE1   ENDS
                             42
                             43              END
ASSEMBLY COMPLETE, NO WARNINGS, NO ERRORS.
```

Das Programm enthält die beiden ausführbaren Code-Segmente (Typ EO) CODE und CODE1. Für beide Segmente generiert BLD386/486 je einen Deskriptor. Der eine Deskriptor beschreibt das CODE-Segment mit der Basisadresse 1000H und der Privilegebene 3 (DPL = 3). Der andere Deskriptor beschreibt das CODE1-Segment mit der Basisadresse 2000H und ebenfalls der Privilegebene 3 (DPL = 3).

Beide Segmente enthalten Prozeduren unterschiedlichen Typs. Im CODE-Segment ist die NEAR-Prozedur INTRA, und im CODE1-Segment ist die FAR-Prozedur INTER enthalten.

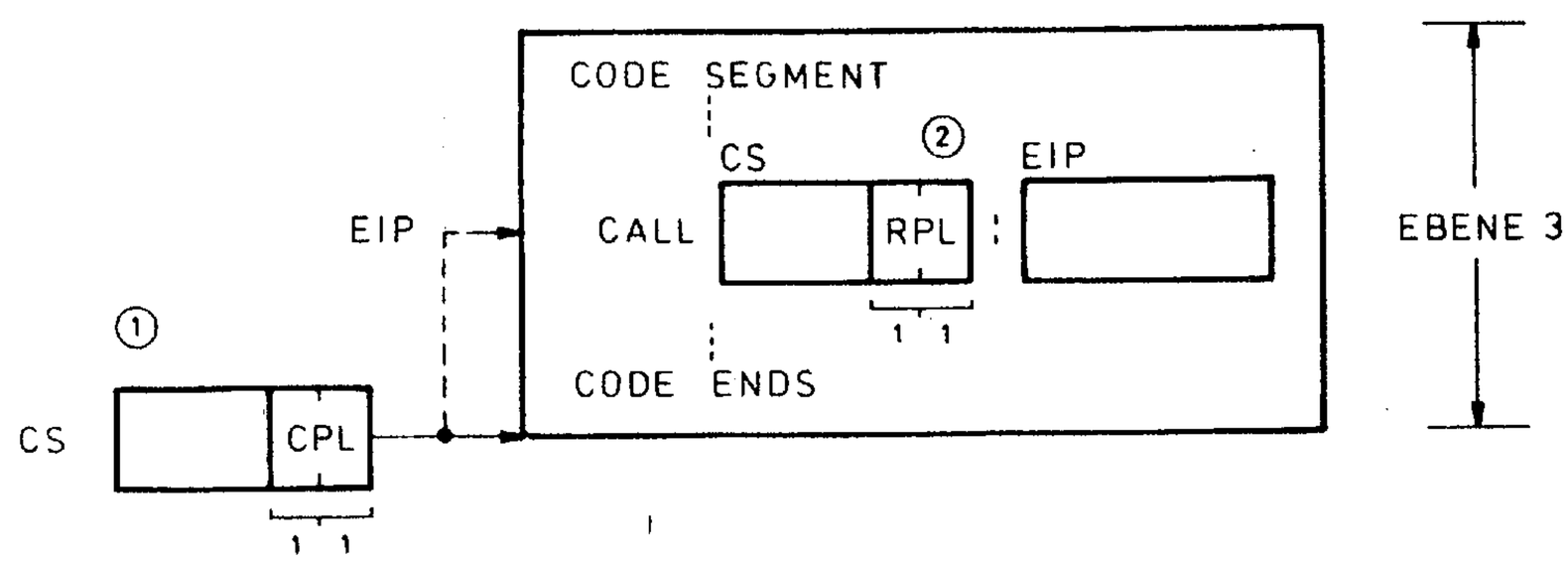
Führt die Task das CODE-Segment auf der Privilegebene 3 (CPL = 3) aus, bewirkt der Befehl CALL INTRA keinen Segment-Wechsel, da nur der „Instruction Pointer" EIP umgeladen wird.

Im Gegensatz dazu bewirkt der Befehl CALL FAR PTR INTER, daß sowohl das CS-Register als auch EIP umgeladen werden. Damit zeigt im CODE-Segment CS:EIP nach CODE1:INTER, wodurch ein Segmentwechsel ausgelöst wird. Die Task führt nun das CODE1-Segment aus, das sich auf der gleichen Privilegebene 3 (CPL = 3) wie das CODE-Segment befindet.

Aus dem Programm-Listing ist durch die Markierung - - - - beim Befehl CALL FAR PTR INTER (CALL Selektor:Offset) ersichtlich, daß ASM386/486 die Selektorkomponente noch nicht kennt. Diese kann vom CALL-Befehl erst dann geliefert werden, wenn der „System Builder" BLD386/486 die Systemkonfiguration festgelegt hat.

3.11.3 Requestor's Privilege Level (RPL)

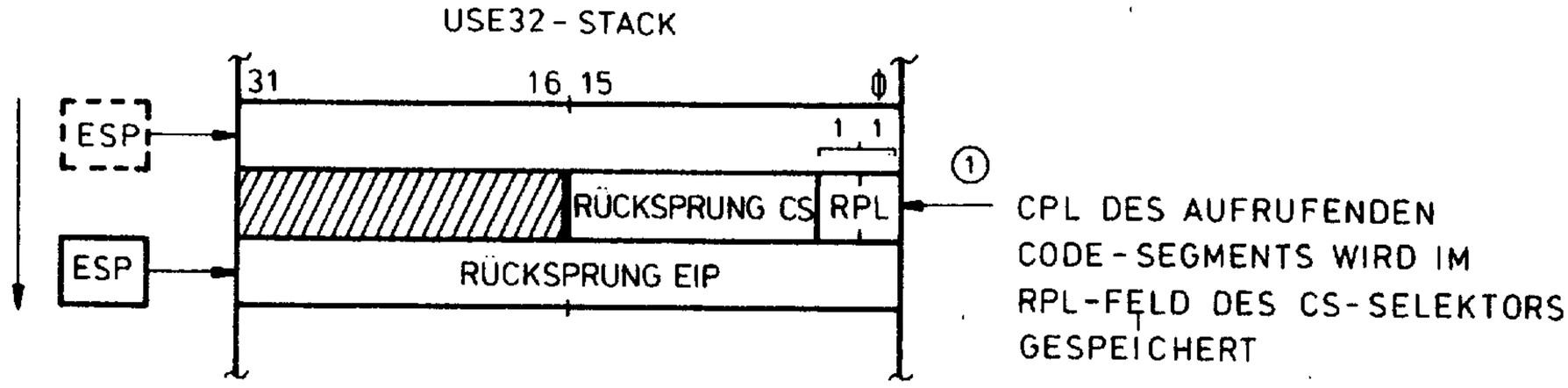
Führt die Task das CODE-Segment auf der Privilegebene 3 aus (siehe vorheriges Programmierbeispiel), liefert der Befehl CALL FAR PTR INTER einen Selektor, dessen Privilegebene in seinem RPL-Feld spezifiziert ist. Diese Situation illustriert das folgende Bild:



Vor der Ausführung des CALL-Befehls enthält das RPL-Feld des CS-Selektors die augenblickliche Privilegebene der Task (CPL = 3) ①.
Führt die Task den Call-Befehl aus, liefert dieser einen Selektor, dessen Privileg-Attribut RPL „normalerweise" die Ebene des Code-Segments kennzeichnet, die den Selektor generiert hat. Im Beispiel ist dies die Ebene 3 (RPL = 3) ②.

Um eine Verletzung der Schutzmechanismen auszuschließen, darf der Vertrauensgrad des erzeugten Selektors ***nicht größer*** sein als der des aufrufenden Code-Segments. Dieser Vertrauensgrad wird im RPL-Feld des Selektors angezeigt. Im Beispiel gibt das Privileg-Attribut RPL = 3 des vom CALL-Befehl generierten Selektors an, daß sein Vertrauensgrad genauso groß ist wie der des aufrufenden CODE-Segments mit CPL = 3.

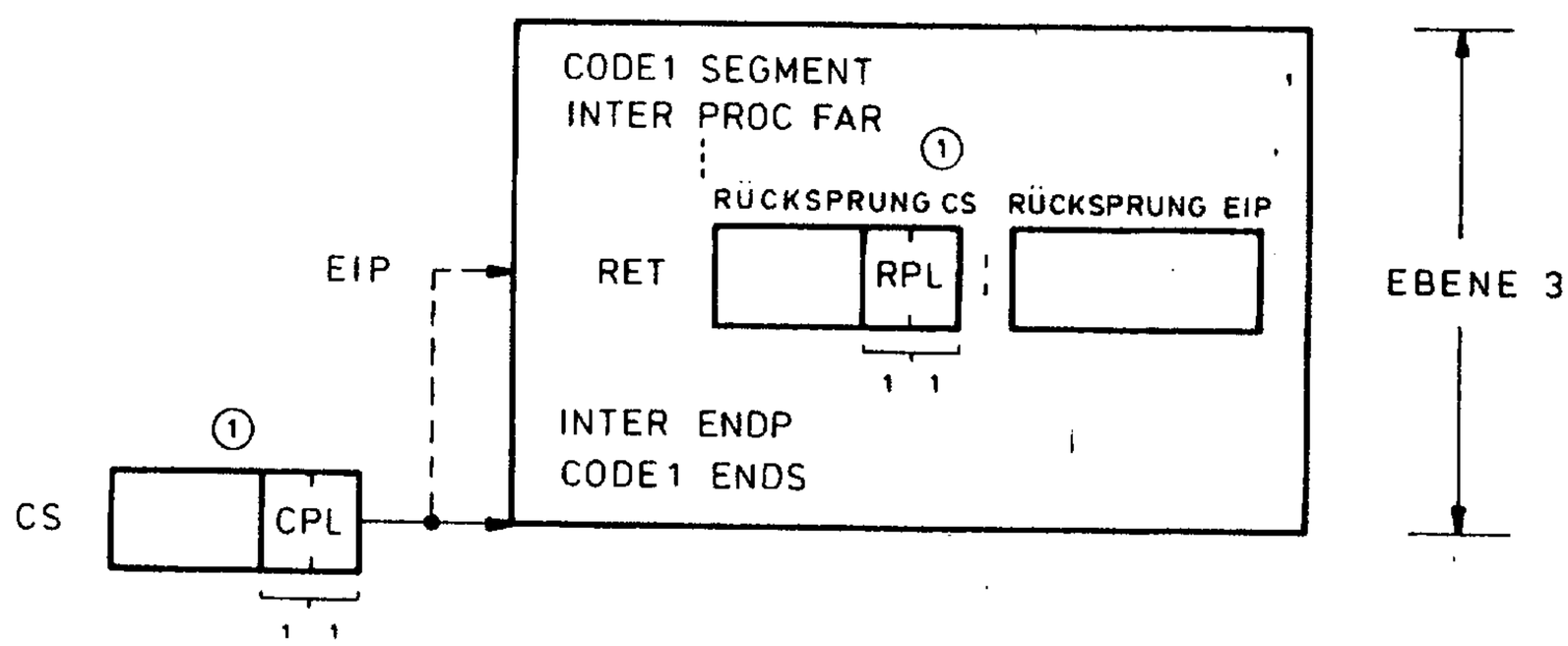
Hat die Task den Befehl CALL FAR PTR INTER ausgeführt, steht die Rücksprungadresse zum aufrufenden CODE-Segment im Stack zur Verfügung. Das folgende Bild illustriert diese Situation:



Es ist ersichtlich, daß als Rücksprungadresse sowohl der CS-Selektor als auch der „Instruction Pointer" EIP des aufrufenden CODE-Segments im Stack gespeichert ist. Dabei zeigt EIP zum ersten Byte nach CALL FAR PTR INTER.

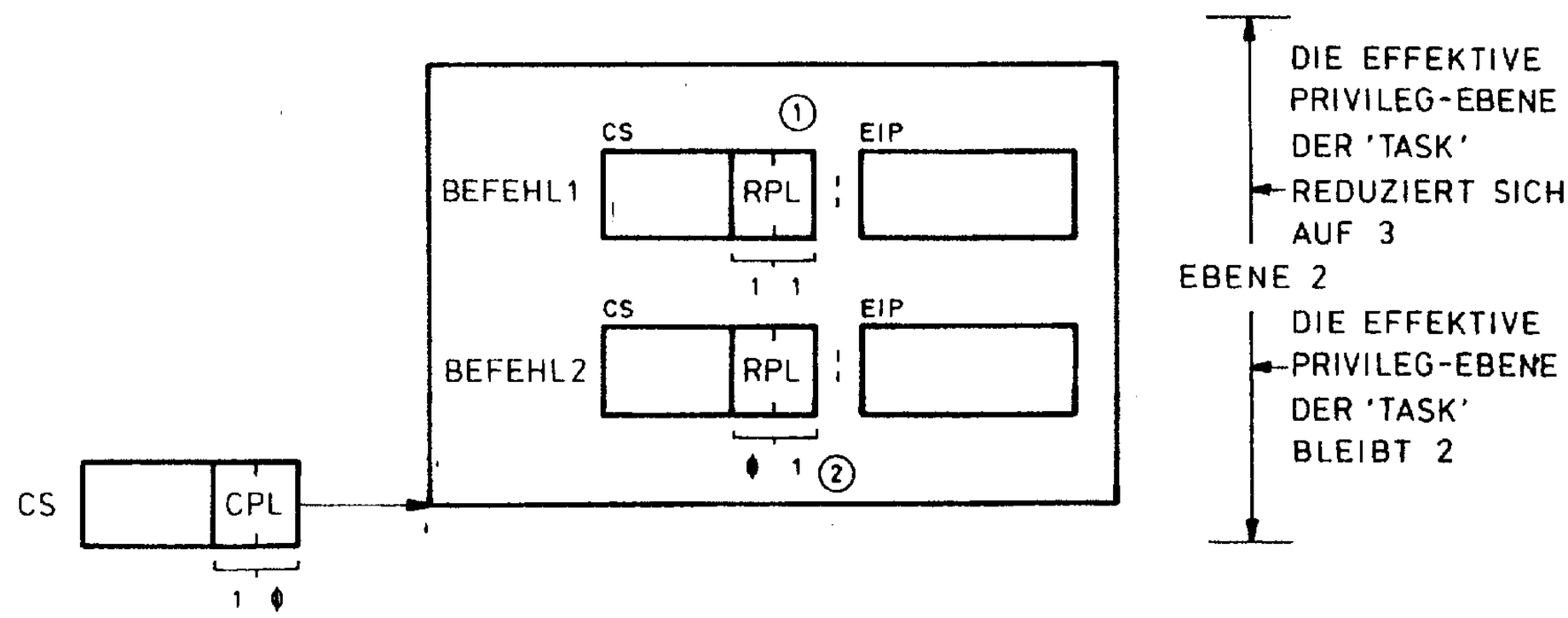
Die Privilegebene der Task, die das Code-Segment ausgeführt hat, wird im RPL-Feld des CS-Rücksprung-Selektors gespeichert. Im Beispiel war dies die Ebene 3 ①.

Nachdem der Segment-Wechsel erfolgt ist, führt die Task die FAR-Prozedur INTER im CODE1-Segment auf der Privilegebene 3 aus (siehe vorheriges Programmierbeispiel).
Dabei liefert der Befehl RET den CS-Selektor, der vor dem Segment-Wechsel im Stack gespeichert worden ist. Dies soll das folgende Bild verdeutlichen:



Es ist ersichtlich, daß der CS-Selektor mit RPL = 3 den gleichen Vertrauensgrad hat, wie das von der Task im Moment ausgeführte CODE1-Segment mit CPL = 3 ①.
Die angegebene Situation liegt nur dann vor, wenn der Befehl RET einen Segment-Wechsel auf der gleichen Privilegebene auslöst.

Der durch RPL spezifizierte Vertrauensgrad eines Selektors kann die effektive Privilegebene einer Task niemals verbessern, sondern nur verschlechtern. Dies soll anhand des folgenden Beispiels erläutert werden:



Es sei vorausgesetzt, daß die Task ein Code-Segment auf der Privilegebene 2 ausführt (CPL = 2). Liefert z. B. BEFEHL1 einen Selektor mit RPL = 3 ① , reduziert dieser RPL-Wert die effektive Privilegebene der Task auf Ebene 3.

Liefert z. B. BEFEHL2 einen Selektor mit RPL = 1 ② , beeinflußt dieser RPL-Wert die effektive Privilegebene der Task nicht. Die Task bleibt auf der Ebene 2.

Die effektive Privilegebene einer Task (EPL = Effective Privilege Level) entspricht daher dem numerischen Maximum von RPL und der augenblicklichen Privilegebene CPL:

EPL = MAX („Task" CPL, Selektor RPL).

3.12 Ablaufsteuerung eines Programms mit Privileg-Wechsel

Innerhalb einer Task kann der Ablauf eines Programms auf ***drei*** Arten gesteuert werden:

- Durch NEAR JMP-, NEAR CALL- und NEAR RET-Befehle innerhalb eines Segments ohne Privileg-Wechsel.
- Durch FAR JMP-, FAR CALL- und FAR RET-Befehle zwischen Segmenten auf der gleichen Privileg-Ebene und
- durch FAR CALL- und FAR RET-Befehle zwischen Segmenten auf unterschiedlichen Privilegebenen.
  (Hinweis: JMP-Befehle auf andere Privilegebenen sind ***nicht*** erlaubt).

Die ersten beiden Typen der Ablaufsteuerung sind bereits diskutiert worden. Es war ersichtlich, daß sie ***keine*** besonderen Steuereinrichtungen benötigten.

Beim dritten Typ der Ablaufsteuerung ist ein spezieller Deskriptor erforderlich, der mittelbar den Wechsel auf eine andere Privilegebene ermöglicht. Dieser spezielle Deskriptor wird als „Call Gate"-Deskriptor bezeichnet und wird von einem FAR CALL-Befehl in der gleichen Weise benützt wie ein Code-Segment-Deskriptor. Allerdings werden die Selektor- und Offsetkomponenten im CALL-Befehl anders behandelt.

Um einen Ablauf eines Programms zwischen Segmenten auf verschiedenen Privilegebenen zu steuern, können ***nur*** die folgenden beiden Befehle benützt werden:

- **CALL Selektor:Offset**
- **RET** ; (Selektor und Offset werden dem Stack entnommen)

Während bei einer Intra-Level-Steuerung mit Segment-Wechsel der Selektor des FAR CALL-Befehls den Deskriptor eines ausführbaren Segments auswählt, selektiert jetzt

- der Selektor einen „Call Gate"-Deskriptor.

Während bei einem „normalen" FAR CALL-Befehl die Offset-Komponente die Einsprungstelle in das Zielsegment markiert, muß auch jetzt

- der Offset-Operand präsent sein, er wird aber ignoriert.

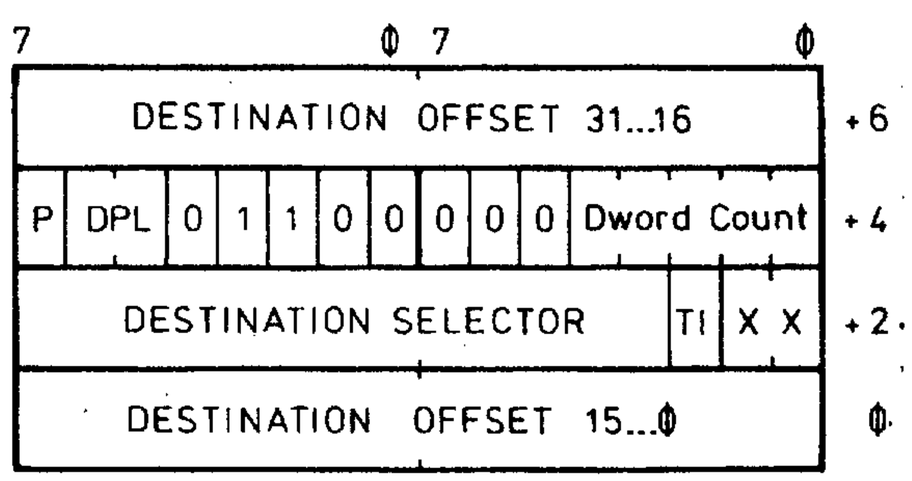
3.12.1 Der „Call Gate"-Deskriptor

Ein „Call Gate"-Deskriptor darf sich entweder

- in der globalen Deskriptor-Tabelle (GDT) oder
- in der lokalen Deskriptor-Tabelle (LDT)

aufhalten, aber ***niemals*** in der Interrupt-Deskriptor-Tabelle (IDT).

Das folgende Bild zeigt das komplette Layout eines „Call Gate"-Deskriptors.



Ein Programm ruft mit

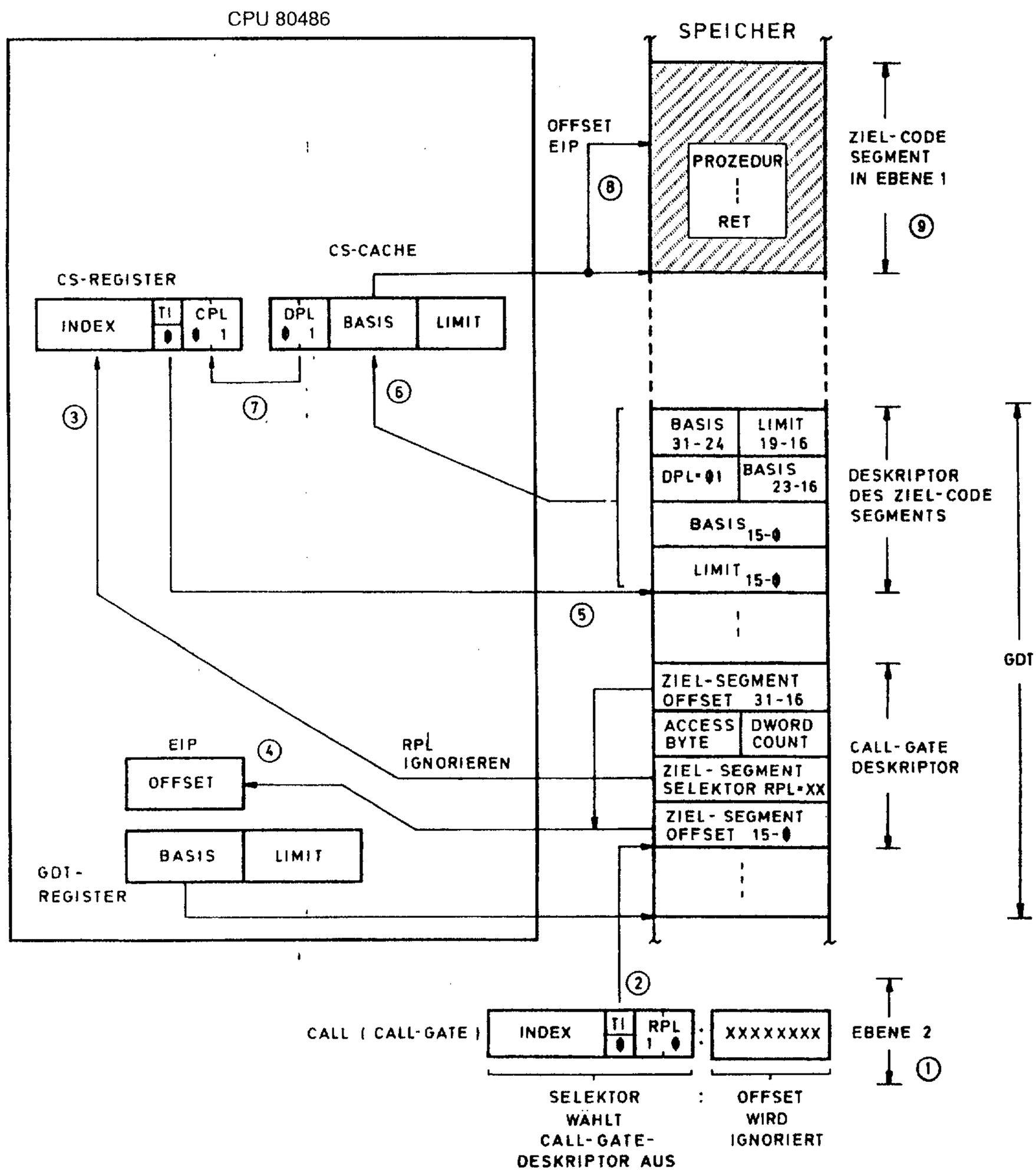
- **CALL Selektor:Offset**
  - Offset → wird ignoriert
  - Selektor → „CALL Gate"-Dekriptor selektieren

einen „Call Gate"-Deskriptor deswegen auf, um eine ***feste*** Einsprungadresse in ein Code-Segment in einer höheren Privilegebene zu bekommen. Diese Einsprungadresse in das Ziel-Segment ist in den Feldern „Destination Selector" und „Destination Offset" im „Call Gate"-Deskriptor enthalten.

Destination-Selector-Feld
Dieses Feld enthält einen Selektor, der entweder in der GDT (TI = 0) oder in der LDT (TI = 1) einen Deskriptor für ein ausführbares Code-Segment auswählt. Das RPL-Feld in diesem Selektor wird ***nicht*** benützt (xx).

Destination-Offset-Feld
Dieses Feld enthält einen 32-Bit-Zeiger, der die Anfangsadresse der auszuführenden Prozedur im ausgewählten Ziel-Code-Segment kennzeichnet.

Das folgende Bild zeigt die Analyse eines Privileg-Wechsels über einen „Call Gate"-Deskriptor.



Es sei vorausgesetzt, daß die Task ein Code-Segment auf der Privilegebene 2 ausführt ①. In diesem Segment soll ein CALL-Befehl einen Selektor liefern (RPL = 2), der in der GDT (TI = 0) einen „Call Gate"-Deskriptor auswählt ②.

Der im „Call Gate"-Deskriptor gespeicherte Ziel-Segment-Selektor überschreibt das augenblickliche CS-Register, wobei RPL dieses Selektors ignoriert wird ③. Parallel dazu wird der Ziel-Segment-Offset in den „Instruction Pointer" EIP geladen ④.

Der im CS-Register gespeicherte Selektor wählt nun in der GDT (TI = 0) den Deskriptor des Ziel-Code-Segments aus ⑤, der anschließend in das CS-Cache-Register geladen wird ⑥.

Die Privilegebene des Ziel-Code-Segments sei 1. Diese Ebene ist im DPL-Feld des zugehörigen Deskriptors spezifiziert. Nach der Initialisierung des CS-Cache-Registers mit

dem Deskriptor des Ziel-Code-Segments wird der DPL-Wert des Deskriptors in das RPL-Feld des CS-Selektors kopiert ⑦. Dadurch wird im CS-Selektor angezeigt, daß sich die Task jetzt auf der Privilegebene 1 befindet (CPL = 1), was dem gewünschten Privileg-Wechsel von Ebene 2 nach Ebene 1 entspricht.
Die Einsprungstelle der Prozedur im Ziel-Code-Segment entnimmt die „Task" dem „Instruction Pointer" EIP ⑧. Die Task führt nun das Ziel-Code-Segment auf der Privilegebene 1 aus ⑨.

3.12.2 Programmierbeispiel:

Privileg-Wechsel über einen „Call Gate"-Deskriptor

Um einen Privileg-Wechsel zu realisieren, ist der Aufruf eines „Call Gate"-Deskriptors erforderlich. Das folgende Programm-Fragment enthält die hierfür notwendige Codierung.

```
LOC       OBJ                 LINE     SOURCE

                               1                NAME      B10
                               2
                               3        EXTRN   CALL_GATE:FAR
                               4
--------                       5        EBENE_2 SEGMENT EO USE32      ;Nur ausfuehrbares Segment mit
                               6                                      ;Basis-Adresse=1000H und DPL=2
                               7                                      ;(Festlegung durch BLD386/486)
                               8                ;
                               9                                      ;Normale Befehlssequenz
                               10               ;
00000000  9A00000000 - - - - R 11               CALL CALL_GATE        ;Aufruf der Prozedur INTER_LEVEL
                               12                                     ;in der Privileg-Ebene 1 ueber einen
                               13                                     ;Call Gate-Deskriptor
                               14                                     ;(Festlegung durch BLD386/486)
                               15               ;
                               16               ;                     ;Normale Befehlssequenz
--------                       17       EBENE_2 ENDS
                               18
                               19
--------                       20       EBENE_1 SEGMENT EO USE32      ;Nur ausfuehrbares Segment mit
                               21                                     ;Basisadresse=200H und DPL=1
                               22                                     ;(Festlegung durch BLD386/486)
                               23               ;
                               24               ;                     ;Normale Befehlssequenz
                               25               ;
00000000                       26       INTER-LEVEL        PROC FAR ;
                               27               ;
                               28               ;                     ;Prozedurkoerper
                               29               ;
00000000  CB                   30               RET
00000001                       31       INTER_LEVEL ENDP
                               32               ;
                               33               ;                     ;Normale Befehlssequenz
--------                       34       EBENE_1 ENDS
                               35
                               36               END
ASSEMBLY COMPLETE, NO WARNINGS, NO ERRORS.
```

Es ist ersichtlich, daß der Modul die beiden Code-Segmente EBENE_1 auf der Privilegebene 1 (DPL = 1) und EBENE_2 auf der Privilegebene 2 (DPL = 2) enthält. Beide Seg-

mente werden von Deskriptoren beschrieben, die der „System Builder“ BLD386/486 spezifiziert hat.

Wenn die Task das EBENE_2-Segment auf der Privilegebene 2 ausführt, liefert der Befehl CALL CALL_GATE einen Selektor, der abhängig vom Wert seines „Table Indicators“ TI einen „Call Gate“-Deskriptor in der GDT (TI = 0) oder in der LDT (TI = 1) auswählt.

Im ASM386/486-Listing ist dieser Selektor durch - - - - - markiert.

Für die Offset-Komponente liefert CALL CALL_GATE den Wert 00000000, da der Offset-Operand zwar präsent sein muß, aber nicht ausgewertet wird.

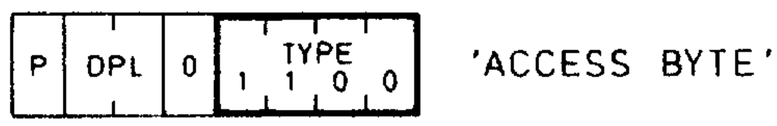
CALL_GATE ist der Name eines „Call Gate“-Deskriptors. Dieser Deskriptor wird vom „System Builder“ BLD386/486 spezifiziert und durch frei wählbare Namen (im Beispiel CALL_GATE) identifiziert. Die von BLD386/486 generierte Systemkomposition ist in einem eigenen Modul, in der sogenannten „Build File“, enthalten. Dies bedeutet, daß in dieser File auch der Deskriptor CALL_GATE existiert.

Daher ist dem Assembler durch EXTRN CALL_GATE:FAR mitzuteilen, daß der Deskriptor CALL_GATE in einer externen Datei definiert ist.

Der im Deskriptor CALL_GATE enthaltene Selektor wählt den Deskriptor des EBENE_1-Segments aus, während die in CALL_GATE enthaltene Offsetkomponente die FAR-Prozedur INTER_LEVEL markiert.

3.12.3 „Call Gate“-Deskriptor Privileg-Level (DPL)

Das folgende Bild zeigt das „Access Byte“ des „Call Gate“-Deskriptors:



Es ist ersichtlich, daß das „Access Byte“ ebenfalls ein DPL-Feld enthält, dessen Wert die Privilegebene angibt, auf der sich der „Call Gate“-Deskriptor befindet.

Das „Type“-Feld

Der 80486 kennt 8 Arten von Gate-Deskriptoren, derern Typ im Type-Feld des Deskriptors spezifiziert wird. Diese Gate-Deskriptoren sind

- 80286-„Call Gate“ ; Type = 0100
- 80286-„Task Gate“ ; Type = 0101
- 80286-„Interrupt Gate“ ; Type = 0110
- 80286-„Trap Gate“ ; Type = 0111
- 80486-„Call Gate“ ; Type = 1100
- 80486-„Task Gate“ ; Type = 0101 (wie 80286-„Task Gate“)
- 80486-„Interrupt Gate“ ; Type = 1110
- 80486-„Trap Gate“ ; Type = 1111

In diesem Kapitel soll zunächst nur das 80486-„Call Gate“ betrachtet werden.

Das „Present"-Bit (P)

Da sich ein Gate-Deskriptor ***nicht*** direkt auf ein normales Segment bezieht, ist es nicht notwendig, daß das Present-Bit im Gate-Deskriptor anzeigt, ob ein Segment gegenwärtig ist. Das P-Bit kann jedoch für andere Zwecke benützt werden.

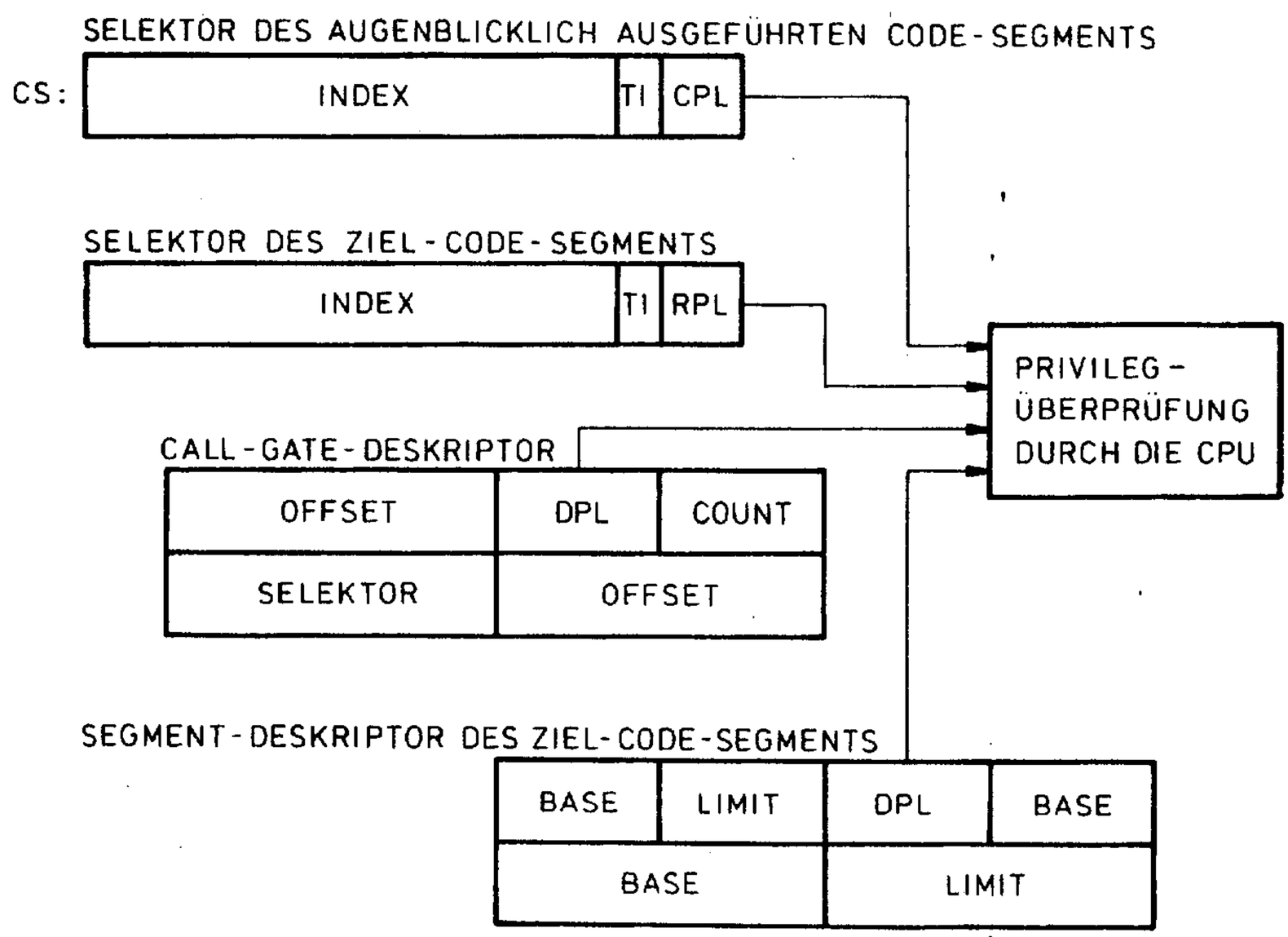
Ist ein Basis-System mit BLD386/486 konfiguriert, müssen Systemdienste wie Ein-/Ausgaben oder Routinen zur Behandlung von Ausnahmesituationen für Anwenderprogramme verfügbar sein.
Daher ist es notwendig, daß System-Codes über Gate-Deskriptoren erreichbar sind. Diese vom Anwender spezifizierten Gate-Deskriptoren können vom „System Builder" BLD386/486 in sogenannte „Export Files" ausgelagert und bei Bedarf mit dem Binder BND386/486 in das System eingebunden werden.

Die „Export Files" kann nun der Anwender benützen, um die Schnittstellen zwischen Anwender-Software und System-Software herzustellen.

Um dem Basis-System mitzuteilen, daß die Gate-Deskriptoren ausgelagert sind, muß BLD386/486 diese Deskriptoren als ***NOT PRESENT*** (P-Bit = 0) markieren. Damit ist der Inhalt der „Call Gate"-Deskriptoren als ungültig gekennzeichnet.

3.12.4 Regeln für „Interlevel"-Programmabläufe

Bei einem Privileg-Wechsel über einen „Call Gate"-Deskriptor benutzt der Prozessor 4 verschiedene Privilegebenen-Felder, um die Gültigkeit des gewünschten Interlevel-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-Befehl gelieferten Selektors.
- Die Privilegebene (DPL) des „Call Gate"-Deskriptors und
- die Privilegebene (DPL) im Segment-Deskriptor des auszuführenden Ziel-Code-Segments.

Ein Segment-Wechsel zwischen verschiedenen Privilegebenen über ein „Call Gate" ist nur dann realisierbar, wenn die folgenden Beziehungen zwischen den Kennziffern der Privilegebenen eingehalten werden:

MAX (Task CPL, Selektor RPL)	**≤ Gate DPL**
und	
Zielsegment DP	**≤ Task CPL**

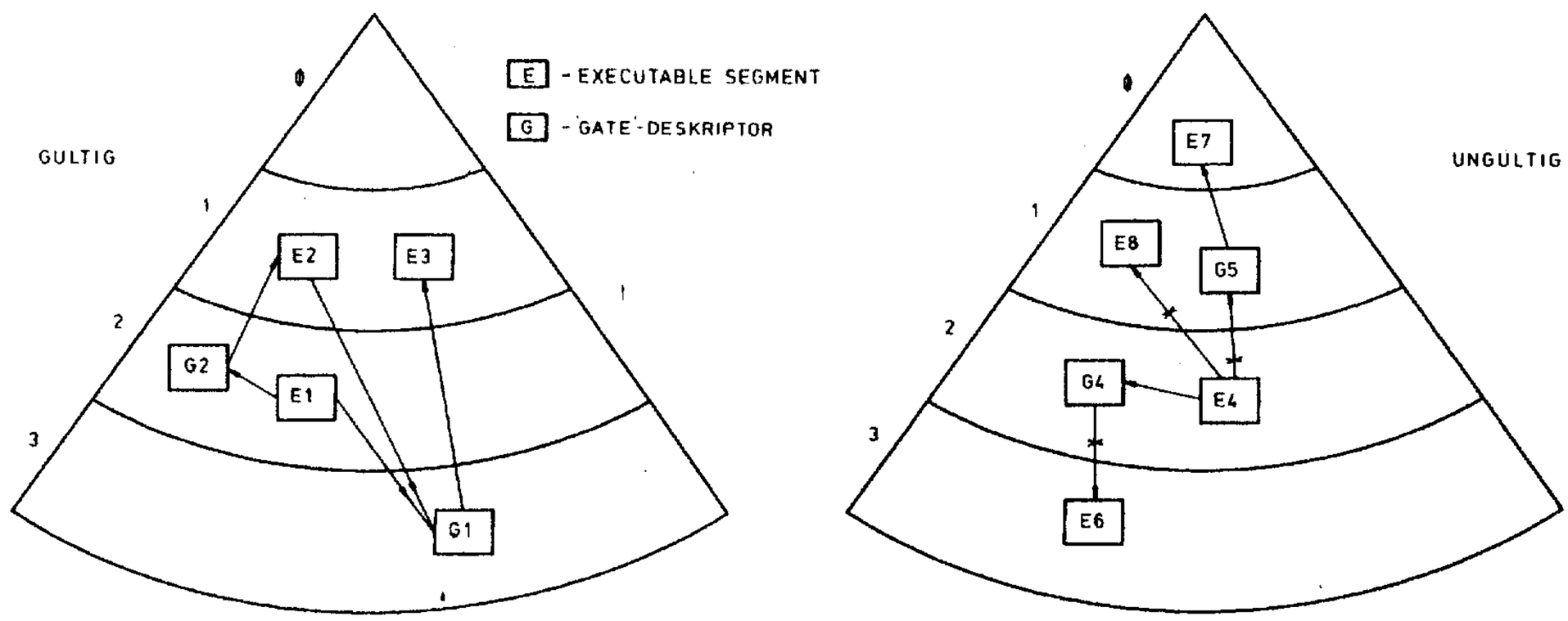
Die angegebenen Regeln zeigen, daß ein CALL-Befehl einen Call-Gate-Deskriptor nur dann erfolgreich benützen kann, wenn

- ein Segmentwechsel in Richtung ***kleinere*** Privileg-Ebenen oder
- ein Segmentwechsel auf der ***gleichen*** Privileg-Ebene

erfolgen soll. Sollte der CALL-Befehl diese Privileg-Regeln nicht einhalten, führt das zwangsläufig zur sogenannten „allgemeinen Schutzverletzung" und damit zur Unterbrechung des Programms.

Für einen Segmentwechsel auf der gleichen Privileg-Ebene liegt es allerdings im Ermessen des System-Programmierers, ob er hierfür ein „Call Gate" erforderlich macht. Wenn ja, verhindert das „Call Gate", daß die aufrufende Prozedur „wild", d. h. unkontrolliert, in das Ziel-Code-Segment springt.

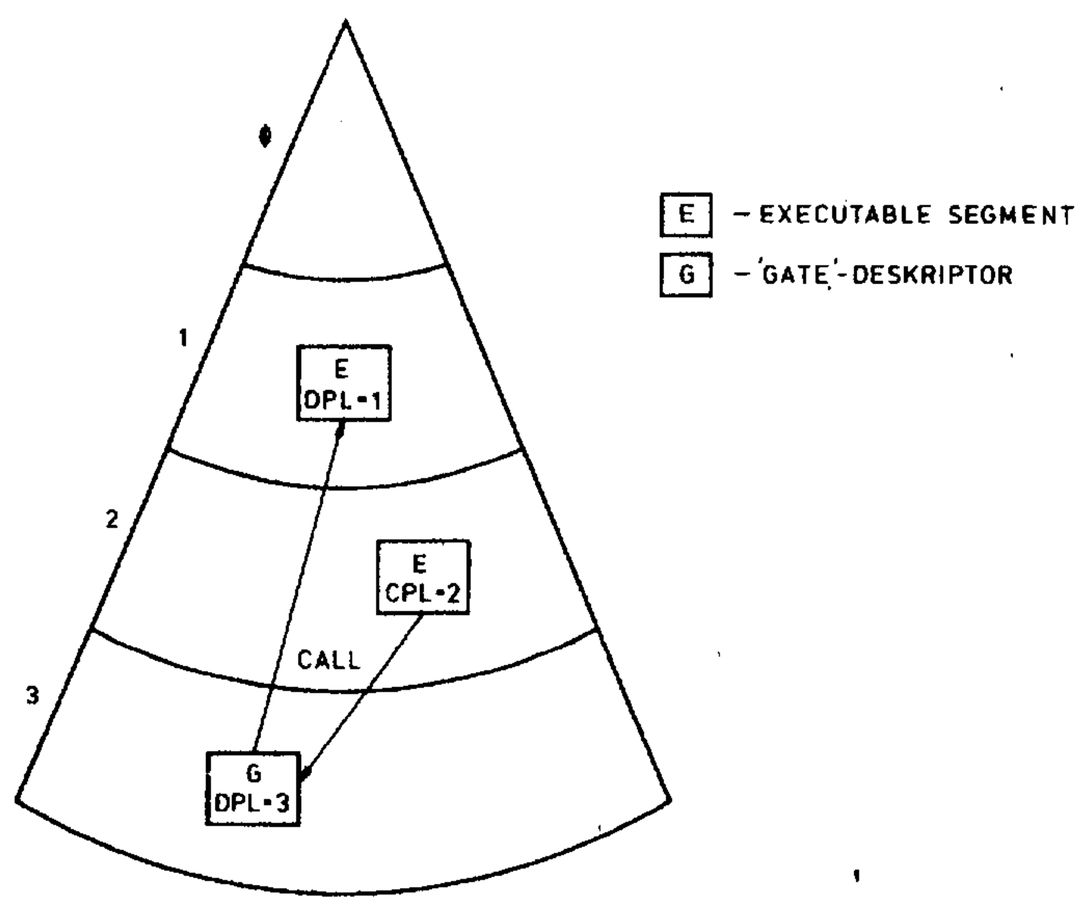
Die folgenden Bilder illustrieren sowohl gültige als auch ungültige Versuche, einen Interlevel-Programmablauf zu realisieren.



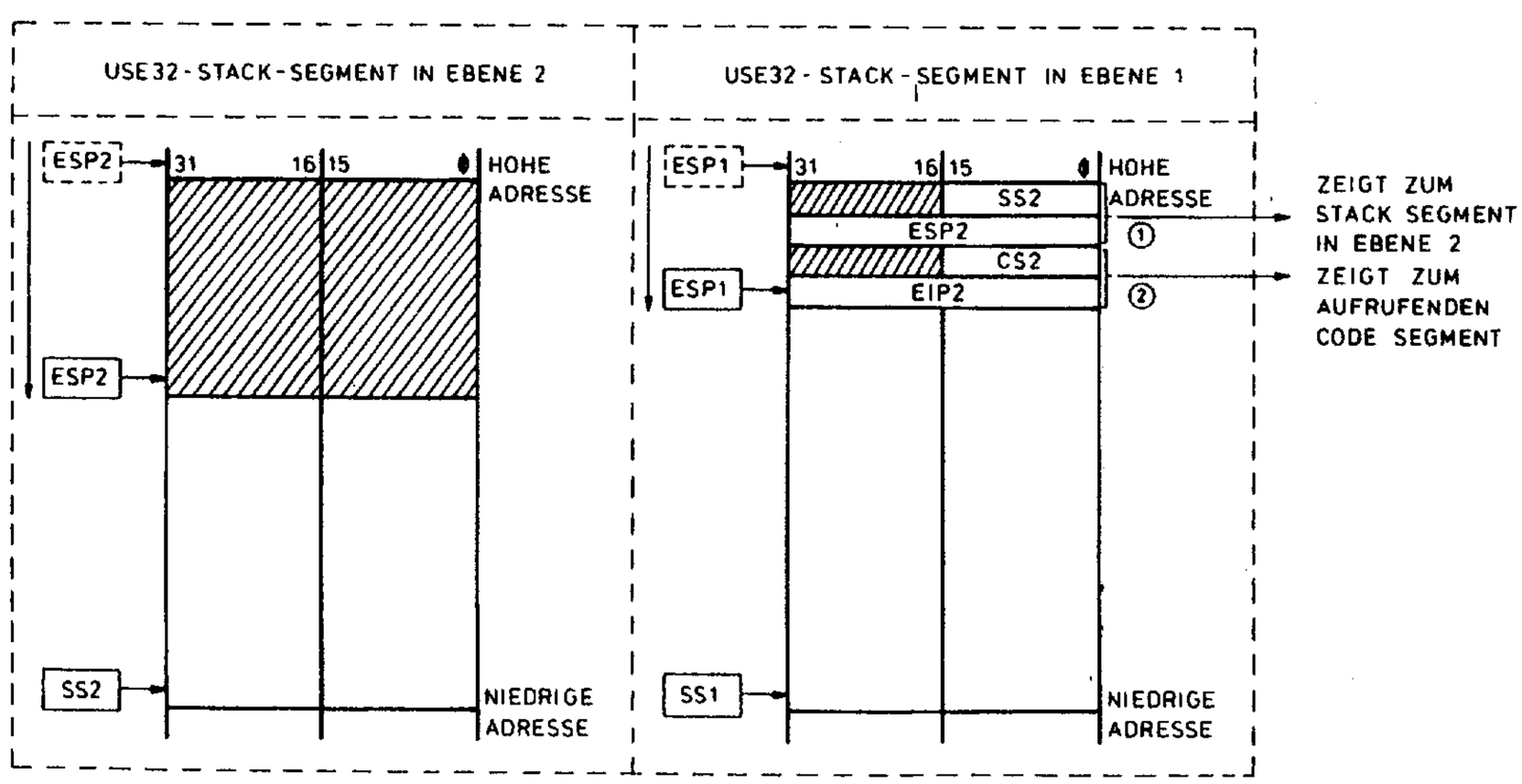
- Der Pfad ***E4-G5-E7*** ist ***ungültig***, da die Privilegebene DPL des Gate-Deskriptors G5 numerisch kleiner ist als CPL des Code-Segments E4.
- Der Pfad ***E4-E8*** ist ebenfalls ***ungültig***, da ein Privileg-Wechsel nur über einen Gate-Deskriptor möglich ist.
- Auch der Pfad ***E4-G4-E6*** ist ***ungültig***, da die Privilegebene des Zielsegments E6 numerisch größer ist als CPL des aktiven Code-Segments E4.
- Nur die Pfade ***E1-G2-E2, E1-G1-E3*** und ***E2-G1-E3*** sind ***gültig***, da hier die oben angegebenen Regeln befolgt werden.

3.13 „Interlevel"-Rückkehr

Immer dann, wenn über einen „Call Gate"-Deskriptor ein Code-Segment in einer höheren Privilegebene aufgerufen wird, bewirkt der Privileg-Wechsel, daß automatisch auch das Stack-Segment gewechselt wird.
Diese Situation sollen die folgenden Bilder an einem Beispiel verdeutlichen.



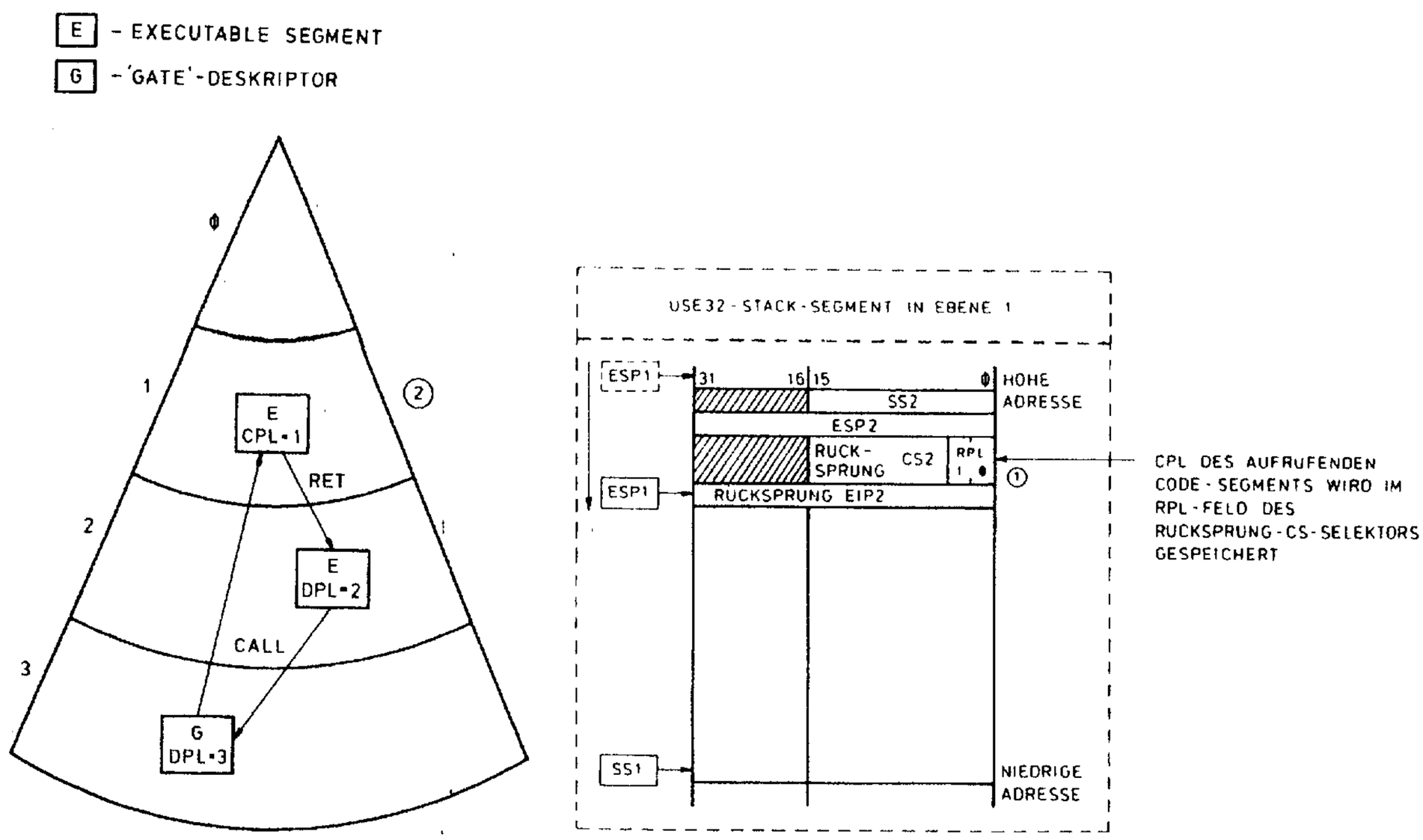
- Es sei vorausgesetzt, daß ein Code-Segment auf der Privilegebene 2 (CPL = 2) über einen „Call Gate"-Deskriptor auf der Privilegebene 3 (DPL = 3) ein Code-Segment auf der Privilegebene 1 (DPL = 1) aufruft.



Die Ausführung des CALL-Befehls auf der Privilegebene 2 (CPL = 2) bewirkt, daß zunächst der momentane 48-Bit-Zeiger SS2:ESP2 zum Stack-Segment in Ebene 2 an das Stack-Segment in Ebene 1 übergeben wird ①. Dabei ist das Ebene-1-Stack-Segment durch einen eigenen 48-Bit-Zeiger SS1:ESP1 gekennzeichnet.

Die Rücksprungadresse CS2:EIP2 zum aufrufenden Code-Segment in Ebene 2 wird anschließend im Ebene-1-Stack-Segment gespeichert ②. Dabei zeigt EIP2 zum nächsten Befehl nach CALL.

Wie bereits früher erwähnt, wird im RPL-Feld des Rücksprung-CS-Selektors die augenblickliche Privilegebene (CPL) des aufrufenden Code-Segments gespeichert. Im Beispiel ist dies die Ebene 2 ①.



Wenn die Task das Ziel-Code-Segment ausführt, befindet sie sich auf der Privilegebene 1 (CPL = 1). Der RET-Befehl in diesem Segment ② liefert den im Ebene-1-Stack gespeicherten Rücksprung-Selektor CS2 und den Rücksprung-„Instruction Pointer" EIP2. Dabei hat der Rücksprung-Selektor CS2 die Privilegebene 2 (RPL = 2).

Immer dann, wenn der RET-Befehl auf einen CS-Wert trifft, dessen RPL > CPL ist, wird ein sogenannter Interlevel-Rücksprung eingeleitet. Dies ist im angegebenen Beispiel mit RPL = 2 und CPL = 1 der Fall.

Da RPL des Rücksprung-Selektors CS2 numerisch größer ist als CPL, wird die effektive Privilegebene der Task EPL = MAX (Task CPL, Selektor RPL) verschlechtert.

Das angegebene Beispiel illustriert, daß ein Privileg-Wechsel durch einen RET-Befehl nur dann möglich ist, wenn die augenblickliche Privilegebene CPL, in der sich der RET-Befehl befindet, numerisch ***kleiner*** ist als DPL des Code-Segments, zu dem der Rücksprung erfolgen soll. Dabei identifiziert RPL im Rücksprung-CS-Selektor die angeforderte Rücksprung-Privilegebene.

Immer dann, wenn ein RET-Befehl einen Privileg-Wechsel einleitet, aktiviert er generell folgende Schritte:

1) Die im augenblicklichen Stack-Segment gespeicherten 48-Bit-Zeiger CS:EIP und SS:ESP werden in die entsprechenden CPU-Register zurückgeladen. Dabei wird der ESP-Wert nicht mit dem Limit-Wert des Stack-Segments in der Ziel-Privileg-Ebene verglichen. Sollte ESP außerhalb des Limit-Wertes liegen, wird dies erst bei der nächsten Stack-Operation erkannt.

2) Die Inhalte der DS-, ES-, FS- und GS-Segment-Register werden überprüft. Sollte sich eines dieser Register auf ein Segment beziehen, dessen DPL-Wert ***kleiner*** ist als der neue CPL-Wert, wird das entsprechende Segment-Register mit dem Null-Selektor (Index = 0, TI = 0) geladen. Auf diese Ausnahme-Situation reagiert der RET-Befehl allerdings nicht. Erst wenn in einer nachfolgenden Speicher-Referenz versucht wird, ein Segment-Register mit einem Null-Selektor zu benutzen, führt dies zur „allgemeinen Schutzverletzung". Dadurch wird verhindert, daß ein weniger privilegierter Code auf höher privilegierte Daten zugreift.

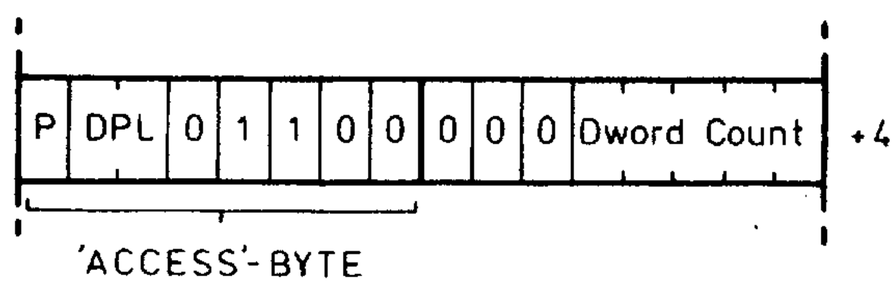
3.14 „Interlevel"-Prozeduraufruf mit Parameterübergabe

3.14.1 Das „DWORD COUNT"-Feld

Wenn ein „Call Gate"-Deskriptor benützt wird, um Privilegebenen zu wechseln, muß das jeweils neue Stack-Segment genügend Platz für die Rücksprungadresse, für alle übergebenen Parameter und die lokalen Variablen enthalten.

Um den Privileg-Wechsel für das aufrufende Programm durchsichtiger zu machen, gibt ein „Call Gate"-Deskriptor an, daß Parameter vom ***alten*** Stack in den ***neuen*** Stack kopiert worden sind.

Wieviele Doppelworte (bis zu 31) vom Stack des aufrufenden Programms in den neuen Stack kopiert wurden, spezifiziert das „DWORD COUNT"-Feld im „Call Gate"-Deskriptor. Enthält das „DWORD COUNT"-Feld den Wert Null, wurden ***keine*** Parameter kopiert. Das folgende Bild zeigt das „DWORD COUNT"-Feld im „Call Gate"-Deskriptor:



3.14.2 Programmierbeispiel: Parameterübergabe

Anhand des folgenden Programm-Fragments soll gezeigt werden, wie in ASM386/486 der „DWORD COUNT"-Parameter zu benützen ist und welche Stack-Layouts sich vor und nach dem Privileg-Wechsel ergeben.

```
LOC       OBJ              LINE  SOURCE

                            1                NAME   B11
                            2
                            3                ASSUME DS:EBENE_2_DATEN
                            4
                            5                EXTRN CALL_GATE:GATE:FAR
                            6
--------                    7   EBENE_2_DATEN       SEGMENT RW USE32  ;Les- und schreibbares Segment mit
                            8                                         ;Basis-Adresse=1000H und DPL=2
                            9                                         ;(Festlegung durch BLD386/486)
00000000  ????????         10   PARAMETER1 DD       ?
00000004  ????????         11   PARAMETER2 DD       ?
00000008  ????????         12   PARAMETER3 DD       ?
                           13
--------                   14   EBENE_2_DATEN       ENDS
                           15
--------                   16   EBENE_2_CODE        SEGMENT EO USE32  ;Nur ausfuehrbares Segment mit
                           17                                         ;Basis-Adresse=2000H und DPL=2
                           18                                         ;(Festlegung durch BLD386/486)
                           19                        ;
                           20                        ;                ;Normale Befehlssequenz
                           21                ;
00000000  FF35000000000  R 22                PUSH   PARAMETER1       ;Uebergabe von 3 Doubleword-
00000006  FF35040000000  R 23                PUSH   PARAMETER2       ;Parametern an das
                                                                     Stack-Segment
0000000C  FF35080000000  R 24                PUSH   PARAMETER3       ;in Ebene 1
00000012  9A00000000---- E 25                CALL CALL_GATE          ;Aufruf der Prozedur INTER_LEVEL
                           26                                         ;in der Privileg-Ebene 1 ueber einen
                           27                                         ;Call Gate-Deskriptor
                           28                                         ;(Festlegung durch BLD386/486)
                           29                ;
                           30                ;                        ;Normale Befehlssequenz
--------                   31   EBENE_2_CODE        ENDS
                           32
                           33
--------                   34   EBENE_1_CODE        SEGMENT EO USE32  ;Nur ausfuehrbares Segment mit
                           35                                         ;Basisadresse=3000H und DPL=1
                           36                                         ;(Festlegung durch BLD386/486)
                           37                ;
                           38                ;                        ;Normale Befehlssequenz
                           39                ;
00000000                   40   INTER_LEVEL PROC FAR WC(3) ;*****************************************
                           41                ;
                           42                ;                        ;Prozedurkoerper
                           43                ;
00000000  CA0C00           44                RET 12                   ;Entferne die Parameter
00000003                   45   INTER_LEVEL ENDP            ;*****************************************
                           46                ;
                           47                ;                        ;Normale Befehlssequenz
--------                   48   EBENE_1_CODE        ENDS
                           49
                           50                END
ASSEMBLY COMPLETE, NO WARNINGS, NO ERRORS.
```

Das Modul enthält das les- und schreibbare Datensegment EBENE_2_DATEN (Typ RW) auf der Privilegebene 2 und die beiden ausführbaren Code-Segmente EBENE_2_CODE auf der Privilegebene 2 und EBENE_1_CODE auf der Privilegebene 1 (Typ EO).

Im EBENE_2_DATEN-Segment sind drei ***„Doubleword“***-Parameter (PARAMETER 1 bis 3) definiert, die zunächst im EBENE_2_CODE-Segment mit PUSH-Befehlen in den Stack der ***Privilegebene 2*** übergeben werden.

Anschließend erfolgt mit CALL CALL_GATE der Aufruf der FAR-Prozedur INTER_LEVEL in der Privilegebene 1 über einen „Call Gate"-Deskriptor.

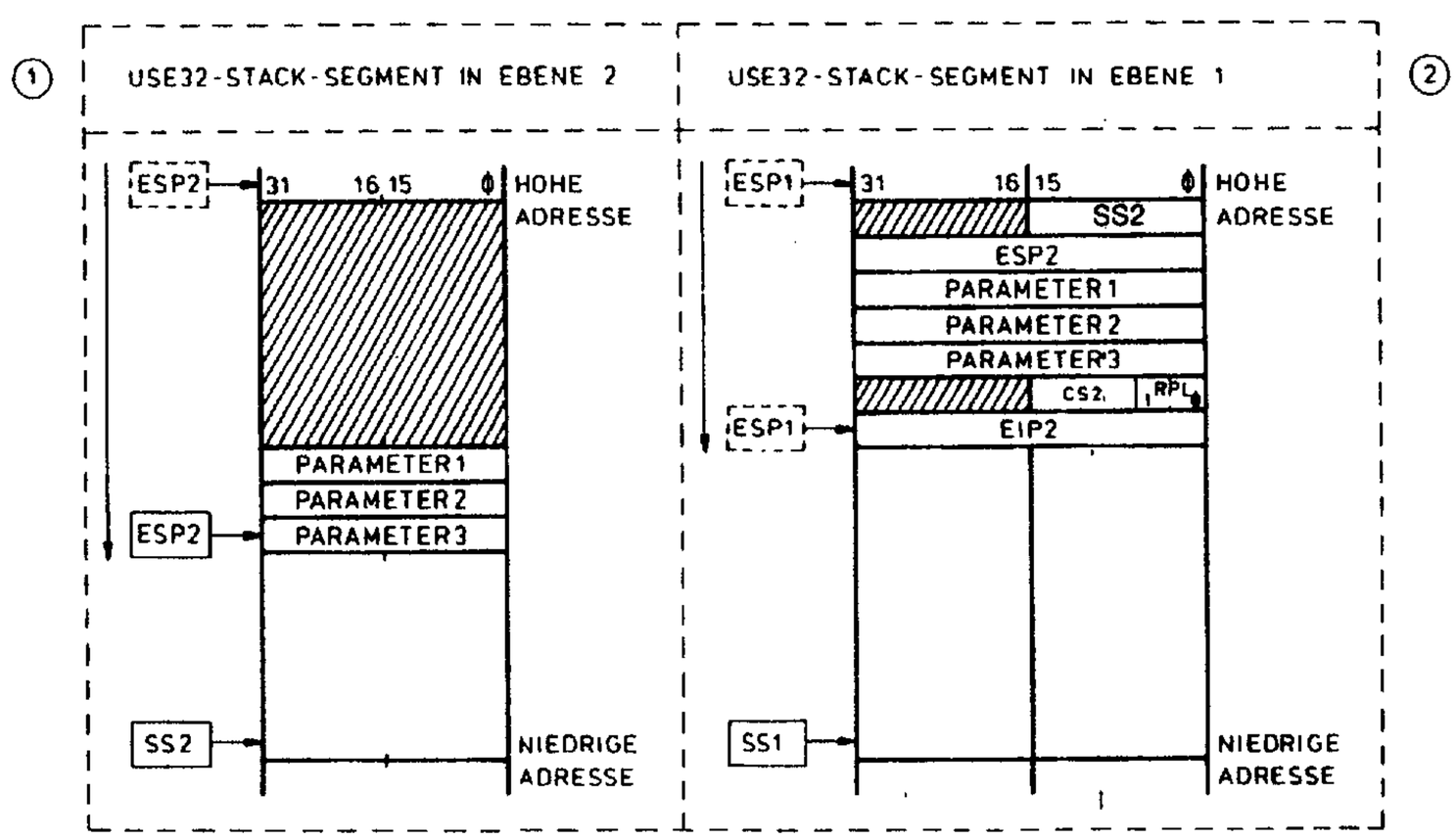
Es ist ersichtlich, daß in der INTER_LEVEL-Prozedurvereinbarung der „*W*ORD *C*OUNT"-Parameter WC(3) angegeben ist, der die allgemeine Form ***WC (Ausdruck)*** hat. Dabei ist ***Ausdruck*** ein Integerwert im Bereich von 0 bis 31. Dieser Wert spezifiziert die Anzahl der ***Doppel***worte, die beim Aufruf der privilegierteren Prozedur als Parameter in den privilegierteren Stack kopiert werden sollen.

Wie das Programm-Fragment zeigt, wird der „DWORD COUNT"-Parameter nicht nur im „Call Gate"-Deskriptor benützt, sondern auch in der PROC-Direktive.
Dies hat den Vorteil, daß damit der Anwender über die Anzahl der kopierten Parameter informiert werden kann, da für ihn der „Call Gate"-Deskriptor „unsichtbar" ist.

Vor dem Aufruf der Prozedur INTER_LEVEL durch CALL CALL_GATE ergibt sich wegen

- PUSH PARAMETER1
- PUSH PARAMETER2
- PUSH PARAMETER3

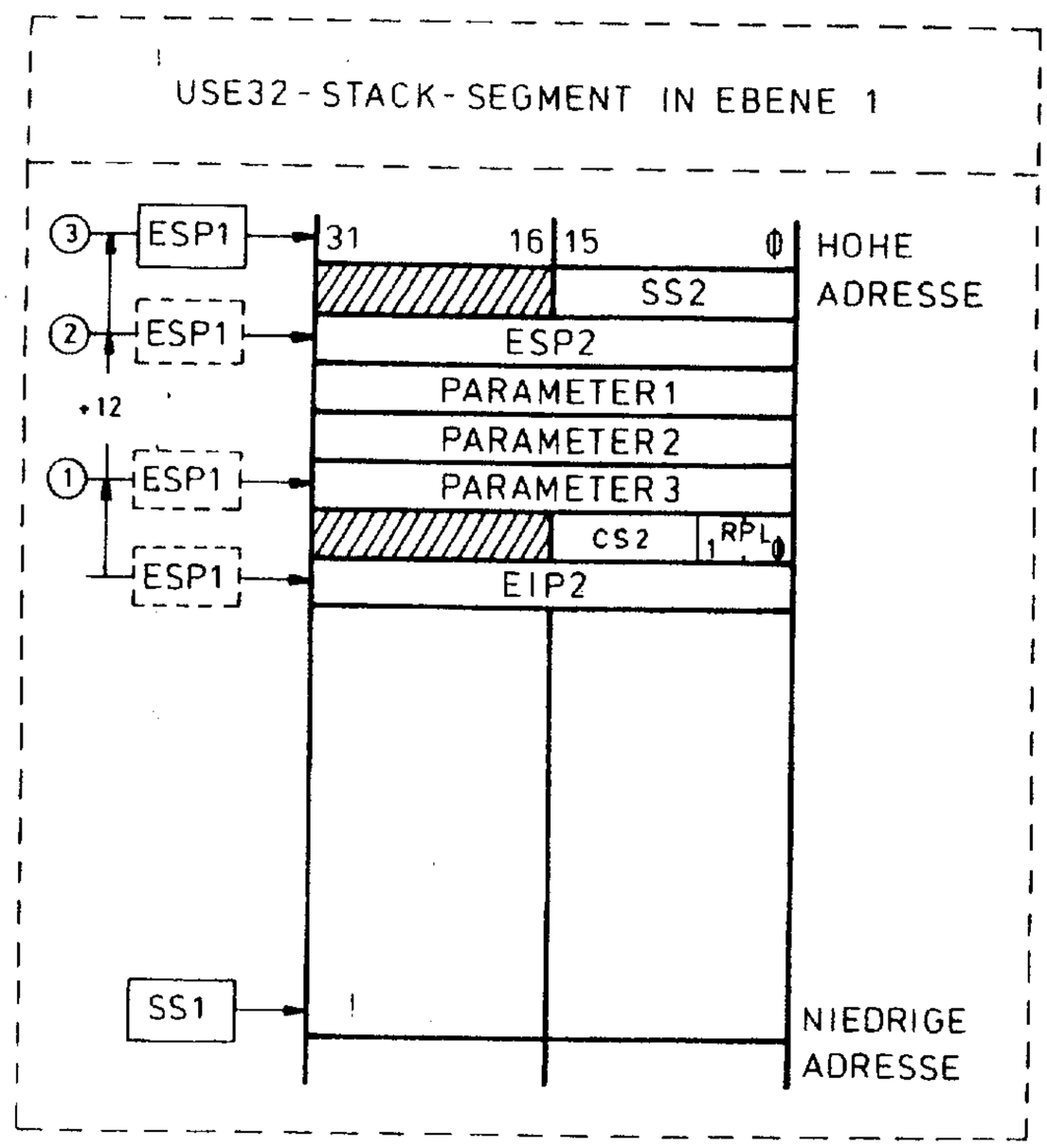
folgendes Stack-Layout in der Privilegebene 2 ①.



Wird nun wegen CALL CALL_GATE der Privileg-Wechsel von Ebene 2 nach Ebene 1 durchgeführt, bestimmt der Wert im „DWORD COUNT"-Feld des „Call Gate"-Deskriptors, wieviele Worte vom Ebene-2-Stack in den Ebene-1-Stack kopiert werden sollen.

Ist der Privileg-Wechsel erfolgt, ergibt sich in der Privilegebene 1 das oben angegebene Stack-Layout ②.
Es ist ersichtlich, daß zunächst der Zeiger SS2:ESP2 zum vorherigen Stack im aktuellen Stack gespeichert wird. Anschließend werden die Parameter, deren Anzahl im „DWORD COUNT"-Feld spezifiziert ist, vom alten in den neuen Stack kopiert.
Schließlich wird danach der Zeiger CS2:EIP2 zum aufrufenden Code-Segment im neuen Stack gespeichert.

Am Ende der FAR-Prozedur INTER_LEVEL erfolgt mit RET 12 der Rücksprung in das aufrufende EBENE_2_CODE-Segment. Die Funktion des Befehls RET 12 soll anhand des Stack-Layouts der Privilegebene 1 erläutert werden.



Der Rücksprung zum EBENE_2_CODE-Segment wird realisiert, indem RET 12 den „Instruction Pointer" EIP mit dem EIP2-Wert und das CS-Register mit dem CS2-Wert initialisiert. Danach zeigt ESP1 zum Parameterfeld ①.

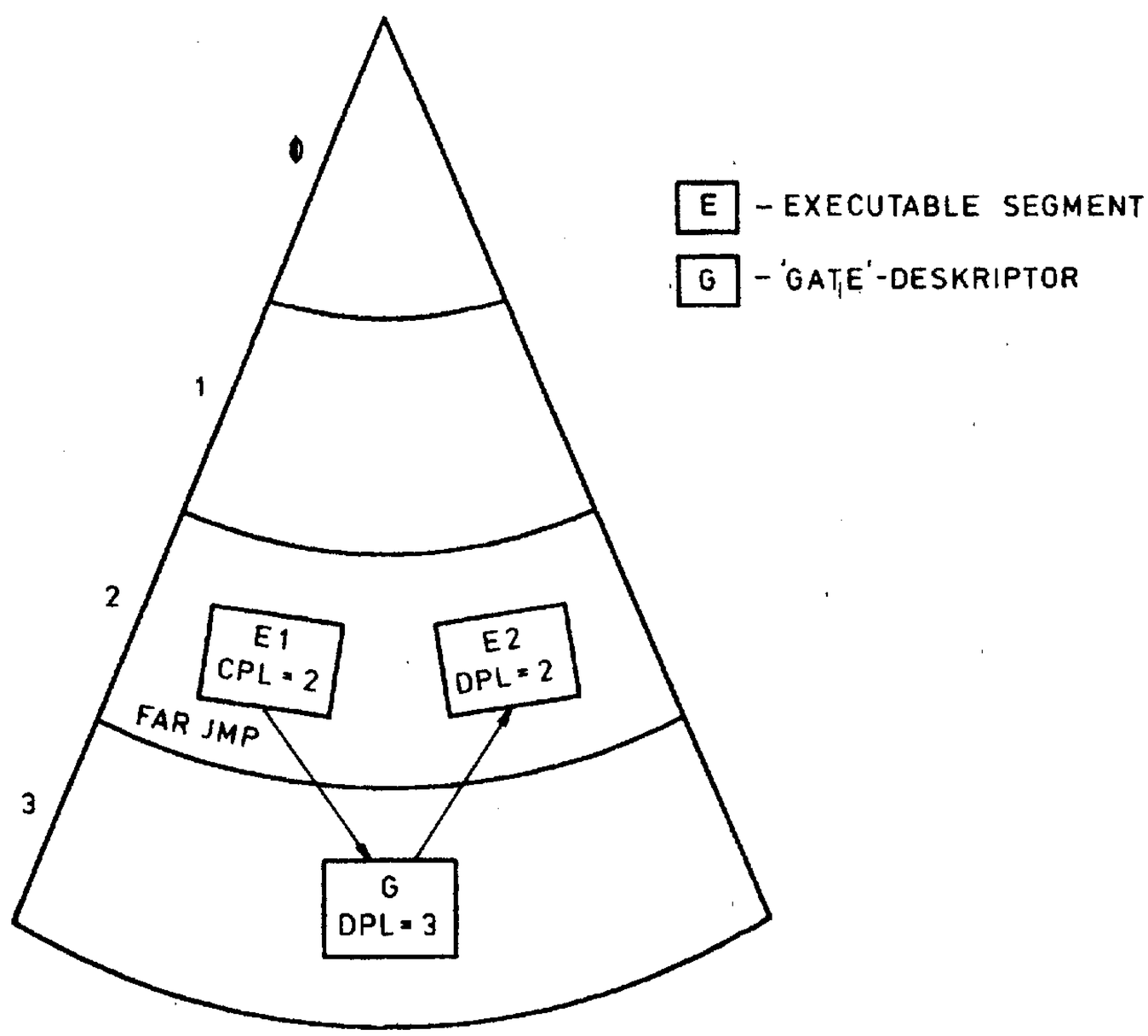
Um nun die Parameter aus dem Ebene-1-Stack zu „entfernen", ist es notwendig, ESP1 auf eine Position jenseits des Parameter-Feldes zu stellen. Im Beispiel wird dies erreicht, wenn der Offsetwert in ESP1 um 12 (12 Bytes;3 Doppelwortparameter) erhöht wird. Die leistet der Operand im Befehl RET 12 automatisch.
Ist RET 12 ausgeführt, zeigt ESP1 zum 48-Bit-Zeiger SS2:ESP2, der das Stack-Segment in der Ebene 2 kennzeichnet ②.
Dieser Zeiger wird nun in das SS-Register und das ESP-Register zurückgeladen, so daß ESP1 wieder an seiner ursprünglichen Position steht ③.

Es sei an dieser Stelle darauf hingewiesen, daß die Stack-Zeiger SS0:ESP0, SS1:ESP1 und SS2:ESP2 für die Privilegebenen 0, 1 und 2 ***nicht*** von CPU-Registern, sondern von einem speziellen System-Segment (Task-Status-Segment) geliefert werden. Dies wird im Kapitel „Tasks und Status-Wechsel" näher erläutert.

3.15 JMP-Befehl und „Call Gate"-Deskriptor

Wie bereits früher erwähnt, kann ein JMP-Befehl für einen Privileg-Wechsel ***nicht*** benützt werden. Er steht lediglich zur Steuerung eines Intra-Segment- oder eines Inter-Segment-Programmablaufs auf der gleichen Privilegebene zur Verfügung.

Für den Fall, daß durch einen JMP-Befehl ein Segmentwechsel erfolgen soll, kann zur Auflage gemacht werden, daß nicht „wild„ in das Ziel-Code-Segment gesprungen wird. Dies wird erreicht, wenn der JMP-Befehl einen „Call Gate"-Deskriptor benützt, da dieser Deskriptor eine feste Einsprungadresse definiert.
Das folgende Bild illustriert eine solche Situation.



Es ist ersichtlich, daß im E1-Code-Segment der Befehl

- **JMP** **Selektor:Offset (FAR JMP)**
  - Offset → wird ignoriert
  - Selektor → „Call Gate"-Deskriptor selektieren

ausgeführt wird, dessen Selektorkomponente einen „Call Gate"-Deskriptor auf der Privilegebene 3 auswählt. Dieser Deskriptor definiert die Einsprungadresse in das E2-Code-Segment, das sich mit dem aufrufenden Code-Segment E1 auf der gleichen Privilegebene befinden muß.

Generell kann ein FAR JMP-Befehl einen „Call Gate"-Deskriptor immer dann benützen, wenn ein Segment-Wechsel

- auf der ***gleichen*** Privileg-Ebene oder
- auf ein Conforming-Code-Segment

erfolgen soll.

Für einen JMP-Befehl auf ein „normales“ (nonconforming) Code-Segment müssen daher folgende Privileg-Regeln eingehalten werden:

MAX (Task CPL, Selektor RPL)	**≤ Gate DPL**
und	
Zielsegment DPL	**= Task CPL**

Beachtet der JMP-Befehl diese Regel nicht, führt dies zur „allgemeinen Schutzverletzung“ und damit zur Programm-Unterbrechung.

3.16 Daten-Zugriffe

Auf Daten darf entweder in ***Daten***-Segmenten oder in ***lesbaren*** Code-Segmenten zugegriffen werden. Immer dann, wenn die Register DS, SS, FS, GS oder ES z. B. durch die Befehle LDS, LES, LGS, LSS, LFS oder MOV geladen werden, erfolgt eine Überprüfung des Access-Bytes im zugehörigen Deskriptor.
Dabei wird festgestellt,

- ob der zu ladende Deskriptor-Typ zum Segment-Register paßt und
- ob ein Segment-Register den Zugriff zu einem bestimmten Segment-Typ zuläßt.

So ist es z. B. nicht möglich, einen „Execute Only“-Segment-Deskriptor in ein Daten-Segment-Register (DS, ES, FS, GS oder SS) zu laden.

Welche Segment-Register Zugriffe auf die unterschiedlichen Segment-Typen zulassen, zeigt die folgende Tabelle:

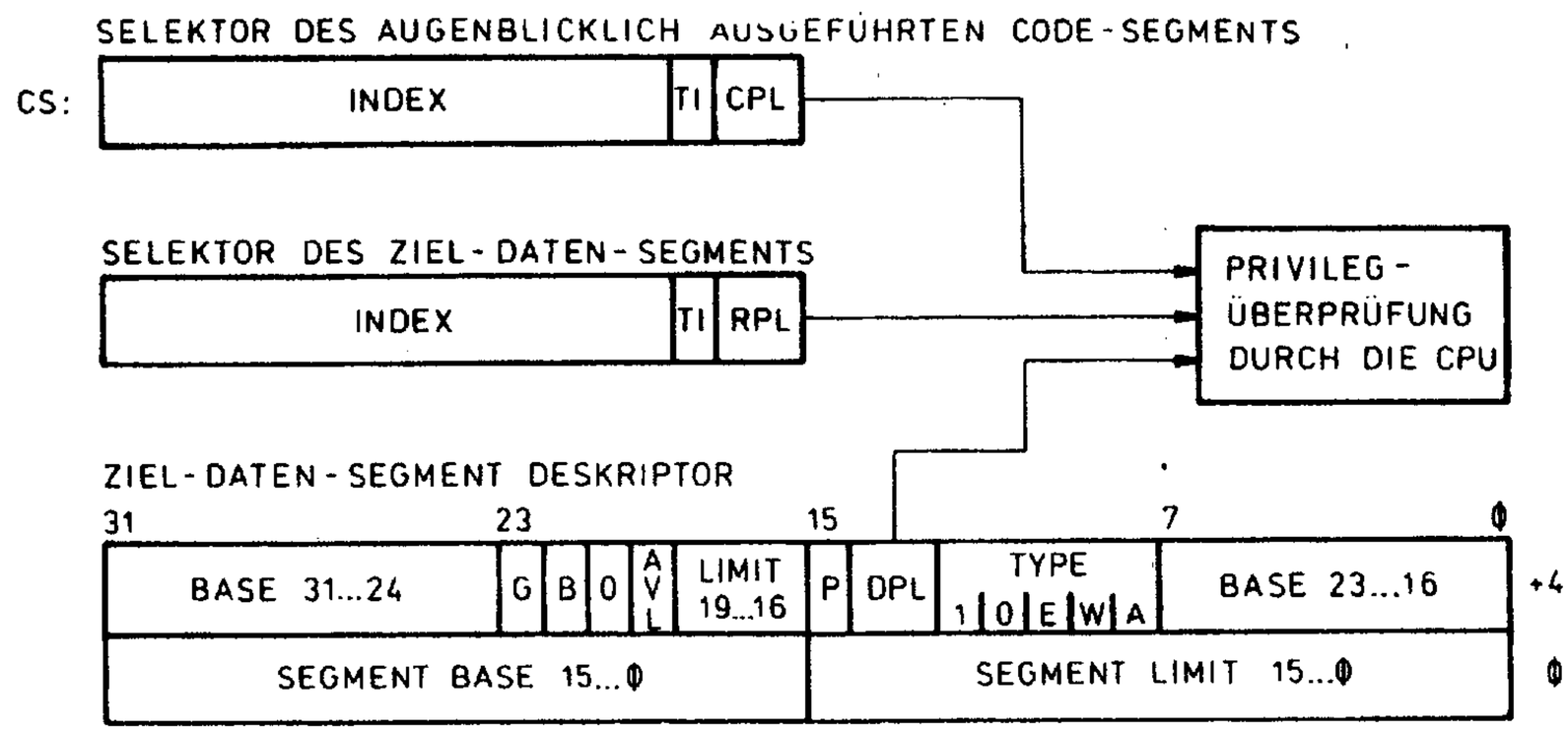
Segment Register	Erlaubte Segment Typen			
	Read-***O***nly (RO) Daten-Segment	***R***ead-***W***rite (RW) Daten-Segment	***E***xecute ***O***nly (EO) Code-Segment	***E***xecute-***R***ead (ER) Code-Segment
DS	Ja	Ja	Nein	Ja
ES	Ja	Ja	Nein	Ja
FS	Ja	Ja	Nein	Ja
GS	Ja	Ja	Nein	Ja
SS	Nein	Ja	Nein	Nein
CS	Nein	Nein	Ja	Ja

Es ist zu sehen, daß z. B. die Segment-Register DS, ES, FS, GS oder SS ***keine*** Zugriffe auf „Execute Only“-Segmente (Typ EO) erlauben.

Neben den oben genannten Überprüfungen wird beim Laden eines Segment-Registers auch eine Privileg-Analyse durchgeführt.
Wie das folgende Bild zeigt, werden dabei ***drei*** verschiedene Privileg-Ebenen berücksichtigt:

- die augenblickliche Privileg-Ebene CPL der Task
- die Deskriptor-Privileg-Ebene DPL des Ziel-Daten-Segments und

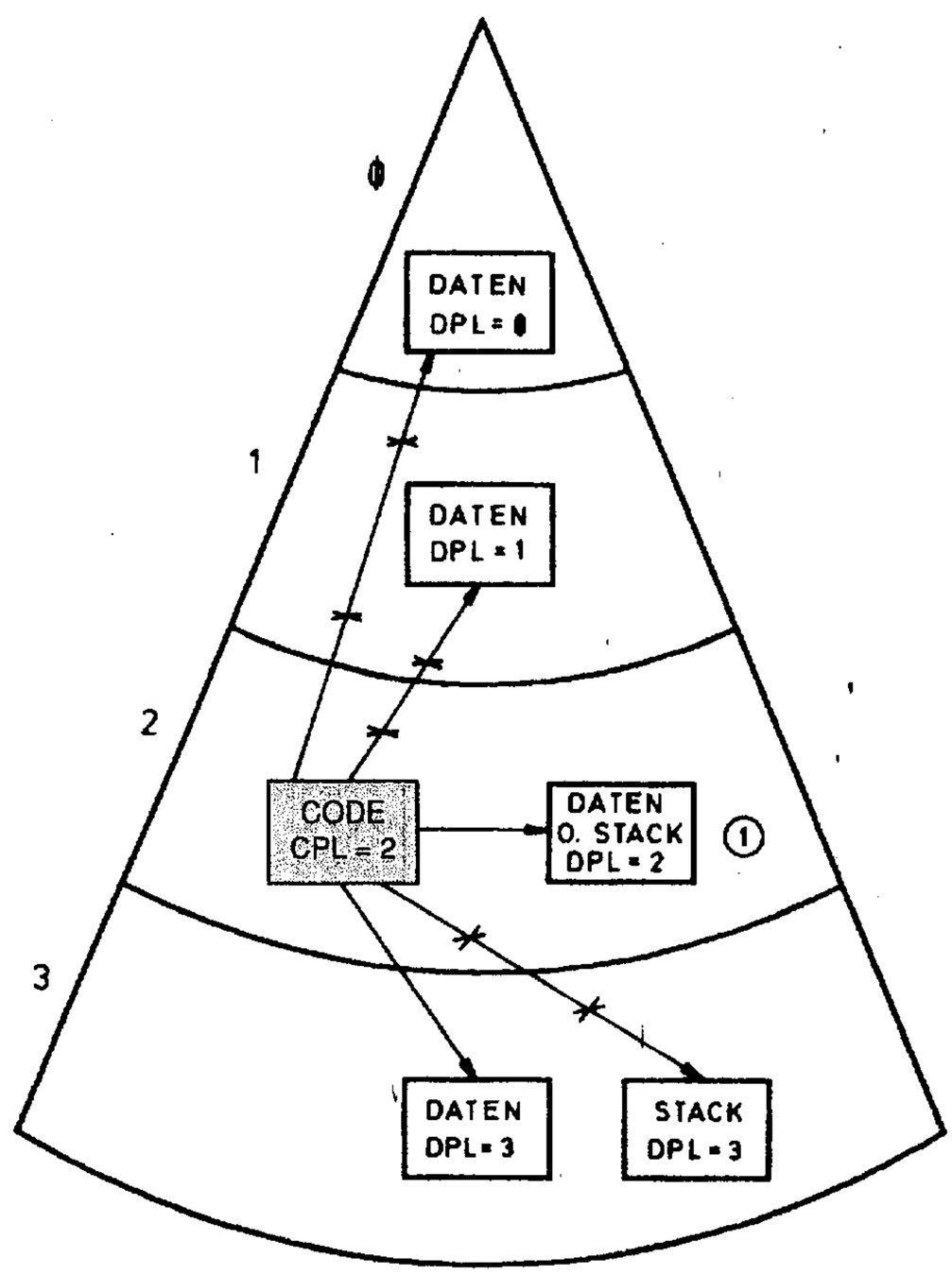
- die Privileg-Ebene RPL des Selektors, den das Programm benützt, um das Ziel-Daten-Segment zu spezifizieren.



Ein Befehl kann ein Daten-Segment-Register nur dann erfolgreich laden, wenn zwischen den Kennziffern der drei Privileg-Ebenen die folgende Beziehung beachtet wird:

Ziel-Daten-Segment DPL ≥MAX (Task CPL, Selektor RPL)

Die Bedeutung dieser Regel ist im folgenden Bild illustriert:



Es zeigt, daß in einem Code-Segment der Zugriff auf ein Daten-Segment nur dann erlaubt ist, wenn DPL des Daten-Segments numerisch ***größer*** oder ***gleich*** CPL der Task ist (DPL ≥ CPL). Lediglich beim Zugriff auf Stack-Daten muß DPL des Stack-Daten-Segments ***gleich*** CPL sein (DPL = CPL) ①.

3.16.1 Zugriff auf Daten in Code-Segmenten

Weniger häufig als Daten-Segmente werden Code-Segmente benutzt, um Daten zu speichern.
Während es nicht möglich ist, in ein Code-Segment zu schreiben, ist jedoch legitim, in einem Code-Segment lesbare Daten (Konstanten) aufzubewahren (Typ ER).

Die folgenden Methoden zeigen nun, wie auf Daten in Code-Segmenten zugegriffen werden kann:

1) Lade ein Daten-Segment-Register (DS, ES, FS oder GS) mit dem Selektor eines non-conforming-, lesbaren und ausführbaren Segments (Typ ER).
2) Lade ein Daten-Segment-Register (DS, ES, FS oder GS) mit dem Selektor eines conforming-, lesbaren und ausführbaren Segments (Typ ER).
3) Benütze die CS-„Override Prefix", um ein les- und ausführbares Segment (Typ ER) zu lesen. Hier muß der Selektor bereits vorher in das CS-Register geladen worden sein.

Im **Fall 1** muß die gleiche Regel beachtet werden, wie sie beim Zugriff auf „normale" Daten-Segmente gilt:

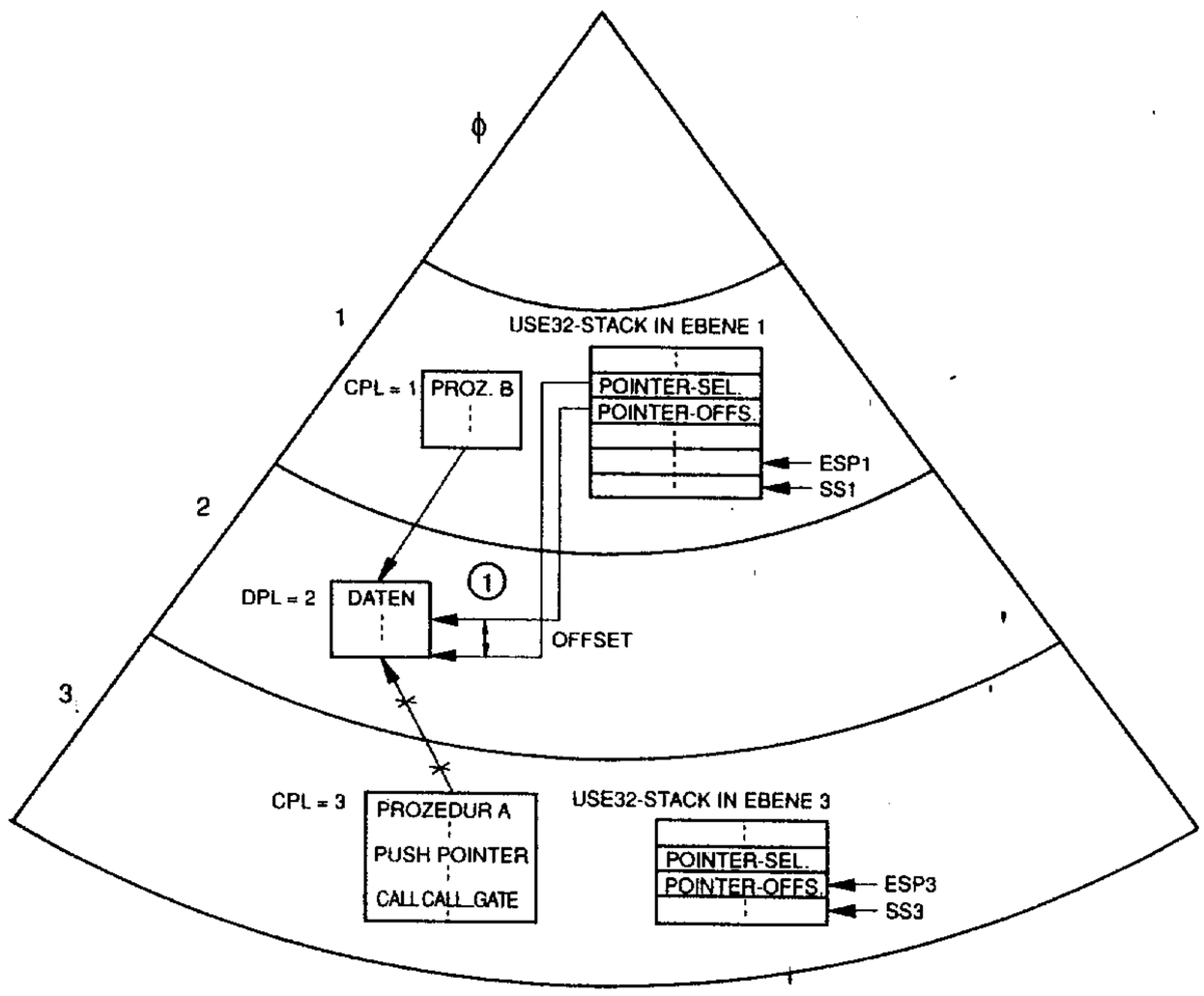
Ziel-Code(Daten)-Segment DPL ≥ MAX (Task CPL, Selektor RPL)

Der **Fall 2** ist immer gültig. Das ist deswegen so, weil ein Conforming-Code-Segment, dessen DPL-Wert ***kleiner*** oder ***gleich*** dem CPL-Wert der Task ist, immer auf der Privileg-Ebene der Task ausgeführt wird. Daher kann auch das von der Task ausgeführte Programm auf die in solchen Conforming-Segmenten gespeicherten Daten zugreifen.
Ist der DPL-Wert eines Conforming-Code-Segments ***größer*** als der CPL-Wert der Task, liegt gemäß Definition ein uneingeschränktes Zugriffs-Recht vor. Das heißt, der Zugriff auf die in diesem Conforming-Code-Segment gespeicherten Daten ist erlaubt.

Der ***Fall 3*** ist ebenfalls immer gültig, da gemäß Definition der DPL-Wert des Ziel-Code(Daten)-Segments mit dem CPL-Wert der Task ***identisch*** ist.

3.17 Parameter-Gültigkeit

Es gibt einen Typ der Privileg-Verletzung, den die 80486-CPU nicht automatisch erkennen kann. Dies soll am folgenden Beispiel erläutert werden.



Es soll die Prozedur A in der Privilegebene 3 existieren, die vor dem Aufruf der Prozedur B (CALL CALL_GATE) in der Privilegebene 1 einen 48-Bit-Pointer-Parameter (PUSH POINTER) an den Stack der Prozedur B übergibt. Dabei soll Prozedur A das RPL-Feld der Parameter-Selektorkomponente so manipuliert haben, daß RPL ***nicht*** mehr die Privilegebene des aufrufenden Code-Segments, sondern eine höhere Ebene spezifiziert. Im Beispiel soll der Parameter-Selektor den RPL-Wert 2 bekommen. Er ist damit mit der Privilegebene eines Datensegments identisch (DPL = 2), auf das die Prozedur B ein uneingeschränktes Zugriffsrecht hat (DPL ≥ CPL).

Während die Parameter-Selektorkomponente einen Deskriptor auswählt, der das Datensegment beschreibt, kann nun die Parameter-Offsetkomponente benützt werden, um z. B. eine Datenstruktur in diesem Segment zu manipulieren ①.

Eine solche Situation, ob absichtlich oder unabsichtlich hervorgerufen, würde normalerweise das Schutzkonzept des 80486 verletzen, da die Privilegebene von Prozedur A (CPL = 3) für einen Zugriff auf eine Datenstruktur in Ebene 2 (DPL = 2) ***nicht*** groß genug ist. Allerdings kann die CPU diesen Typ von Privileg-Verletzung nicht erkennen, da die Privilegebene von Prozedur B (CPL = 1), die eigentlich die Datenstruktur adressiert, ***genügend*** groß ist.

Der beschriebene Trick ist deshalb vergleichbar mit der historischen Geschichte vom „Trojanischen Pferd", mit dessen Hilfe griechische Soldaten in die Stadt Troja eindringen konnten. Zur Lösung dieses sogenannten Trojanischen Problems stellt der 80486 jedoch einige Befehle zur Verfügung, die solche Schutzverletzungen überwachen. Vor der Erläuterung dieser Befehle soll zunächst die beschriebene Situation durch das folgende ASM386/486-Programm realisiert werden.

3.17.1 Programmierbeispiel: Privileg-Verletzung

```
LOC       OBJ                    LINE  SOURCE
                                   1          NAME  B12
                                   2
                                   3  ASSUME DS:EBENE_3_DATEN,ES:EBENE_2_EXTRA_DATEN
                                   4
                                   5  EXTRN CALL_GATE:FAR
                                   6
--------                           7  STACK_FRAME  STRUC
00000000                           8  EBENE_3_EBP          DD   ?                ;Ebene 3-EBP
00000004                           9  EIP3                 DD   ?                ;Ruecksprung-EIP
00000008                          10  CS3                  DD   ?                ;Ruecksprung-CS
                                  11                                             ;(Hoeherwertige 16 Bits sind
                                  12                                             ;undefiniert)
0000000C                          13  POINTER_OFFSET       DD   ?                ;Offset des Pointer-Parameter
00000010                          14  POINTER_SELEKTOR     DD   ?                ;Selektor des Pointer-Parameters
                                  15                                             ;(Hoeherwertige 16 Bits sind
                                  16                                             ;undefiniert)
00000014                          17  ESP3                 DD   ?                ;Ebene 3-ESP
00000018                          18  SS3                  DD   ?                ;Ebene 3-SS
                                  19                                             ;(Hoeherwertige 16 Bits sind
                                  20                                             ;undefiniert
--------                          21  STACK_FRAME  ENDS
                                  22
--------                          23  EBENE_1_CODE SEGMENT EO USE32              ;Nur ausfuehrbares Segment mit
                                  24                                             ;Basisadresse=1000H und DPL=1
                                  25                                             ;(Festlegung durch BLD386/486)
                                  26
00000000                          27  B        PROC   FAR   WC(2)                ;48-Bit Pointer (2 Doppelworte)
                                  28                                             ;an den Ebene 1-Stack ueber
                                                                                 geben
00000000 55                       29           PUSH   EBP                        ;Rette EBP des Ebene 3-Stack
00000001 8BEC                     30           MOV    EBP,ESP                    ;Stelle EBP an die Spitze des
                                  31                                             ;Ebene 1-Stack
                                  32           ;
                                  33           ;
00000003 8B5D0C                   34           MOV EBX,[EBP].POINTER_OFFSET      ;(EBX) = 32 Bit-Pointer-Offset
00000006 8E4510                   35           MOV ES,WORD PTR [EBP].POINTER_SELEKTOR
                                  36                                             ;(ES) = 16 Bit-Pointer-Selektor
00000009 268A03                   37           MOV AL,ES:BYTE PTR[EBX]           ;(AL) = 1.tes Element des
                                                                                 Vektors
                                  38           ;
                                  39           ;
0000000C 5D                       40           POP EBP                           ;Lade EBP mit dem alten
                                                                                 Ebene 3-Wert
0000000D CA0800                   41           RET 8                             ;Entferne 48-Bit-Pointer aus dem
                                  42                                             ;Ebene 1-Stack
00000010                          43  B        ENDP
                                  44
--------                          45  EBENE_1_CODE ENDS
                                  46
--------                          47  EBENE_2_EXTRA_DATEN SEGMENT RW USE32       ;Les- und schreibbares Seg-
                                                                                 ment mit
                                  48                                             ;Basis-Adresse 2000H und
                                                                                 DPL=2
                                  49                                             ;(Festlegung durch BLD386/486)
00000000 00                       50  VEKTOR  DB  0,1,2,3,4,5,6,7,8,9           ;Vektor mit 10 Byte-Elementen
00000001 01
00000002 02
00000003 03
00000004 04
00000005 05
```

```
00000006 06
00000007 07
00000008 08
00000009 09
                                  51
--------                          52  EBENE_2_EXTRA_DATEN ENDS
                                  53
                                  54  EBENE_3_DATEN       SEGMENZ RW USE32   ;Les- und schreibbares Seg-
                                                                             ment mit
                                  55                                         ;Basisadresse 3000H und DPL=3
                                  56                                         ;(Festlegung durch BLD386/486)
00000000 ????????????             57  POINTER        DQ      ?              ;Der an die Prozedur B zu
                                                                             uebergebende
                                  58                                         ;48-Bit Pointer-Parameter ist in
                                  59                                         ;einem Quadword gespeichert
                                  60
--------                          61  EBENE_3_DATEN       ENDS
                                  62
--------                          63  EBENE_3_CODE        SEGMENT EO USE32   ;Nur ausfuehrbares Segment mit
                                  64                                         ;Basisadresse 4000H und
                                                                             DPL=3
                                  65                                         ;(Festlegung druch BLD386/486)
00000000                          66  A       PROC   NEAR
                                  67          ;
                                  68          ;
00000000 C7050000000000000000 R   69          MOV DWORD PTR POINTER,OFFSET VEKTOR     ;32 Bit-POINTER-Offset
                                                                                      = Anfangsadresse
                                  70                                         ;         von VEKTOR
000000A  66C705040000001A00   R   71          MOV WORD PTR POINTER+4,0000000000011010B
                                  72                                         ;16 Bit-POINTER-Selektor mit
                                  73                                         ;         RPL=2
                                  74                                         ;         TI=0 (GDT)
                                  75                                         ;         Index=3 (Offset=18H)
00000013 FF3504000000         R   76          PUSH   DWORD PTR POINTER+4     ;Uebergabe des POINTER-
                                                                             Selektors (die
                                  77                                         ;oberen 16 Bits sind undefi-
                                                                             niert) und
00000019 FF3500000000         R   78          PUSH   DWORD PTR POINTER       ;des POINTER-Offsets an
                                  79                                         ;das Stack-Segment in Ebene 1
0000001F 9A00000000 ----      E   80          CALL CALL_GATE                 ;Aufruf der Prozedur B in der
                                  81                                         ;Privileg-Ebene 1 ueber einen
                                  82                                         ;Call Gate-Deskriptor
                                  83                                         ;(Festlegung durch BLD386/486)
                                  84          ;
                                  85          ;
00000026                          86  A       ENDP
                                  87
--------                          88  EBENE_3_CODE ENDS
                                  89
                                  90          END
ASSEMBLY COMPLETE, NO WARNINGS, NO ERRORS.
```

Das angegebene Programm-Fragment enthält insgesamt ***vier*** Segmente:

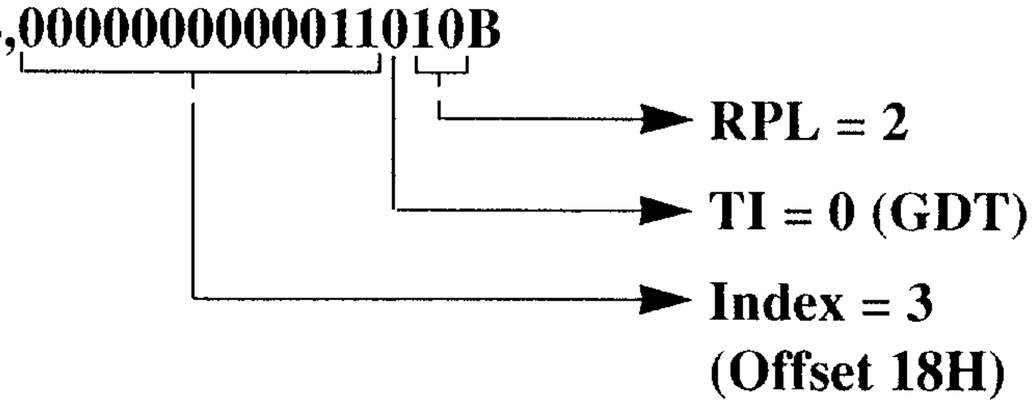
- EBENE_1_CODE in der Privilegebene 1
- EBENE_2_EXTRA_DATEN in der Privilegebene 2
- EBENE_3_DATEN in der Privilegebene 3
- EBENE_3_CODE in der Privilegebene 3

Das EBENE_3_CODE-Segment enthält die Prozedur A, in der zunächst ein 48-Bit-Pointer zur Datenstruktur VEKTOR im EBENE_2_EXTRA_DATEN-Segment generiert wird. Dabei benützt diese Prozedur die „Quadword"-Variable POINTER im EBENE_3_DATEN-Segment, um darin mit

- **MOV DWORD PTR POINTER, OFFSET VEKTOR**

den ***32-Bit-Offset*** von VEKTOR, und mit

- **MOV WORD PTR POINTER+4,0000000000011010B**



den *16-Bit-Selektor* des EBENE_2_EXTRA_DATEN-Segments zu speichern. Das RPL-Feld des Selektors ist mit dem Wert 2 initialisiert. Dies bedeutet, daß der Selektor höher privilegiert ist, als das EBENE_3_CODE-Segment, das diesen Selektor liefert.

Den in der POINTER-Variablen hinterlegten 48-Bit-Pointer speichert die Prozedur A mit

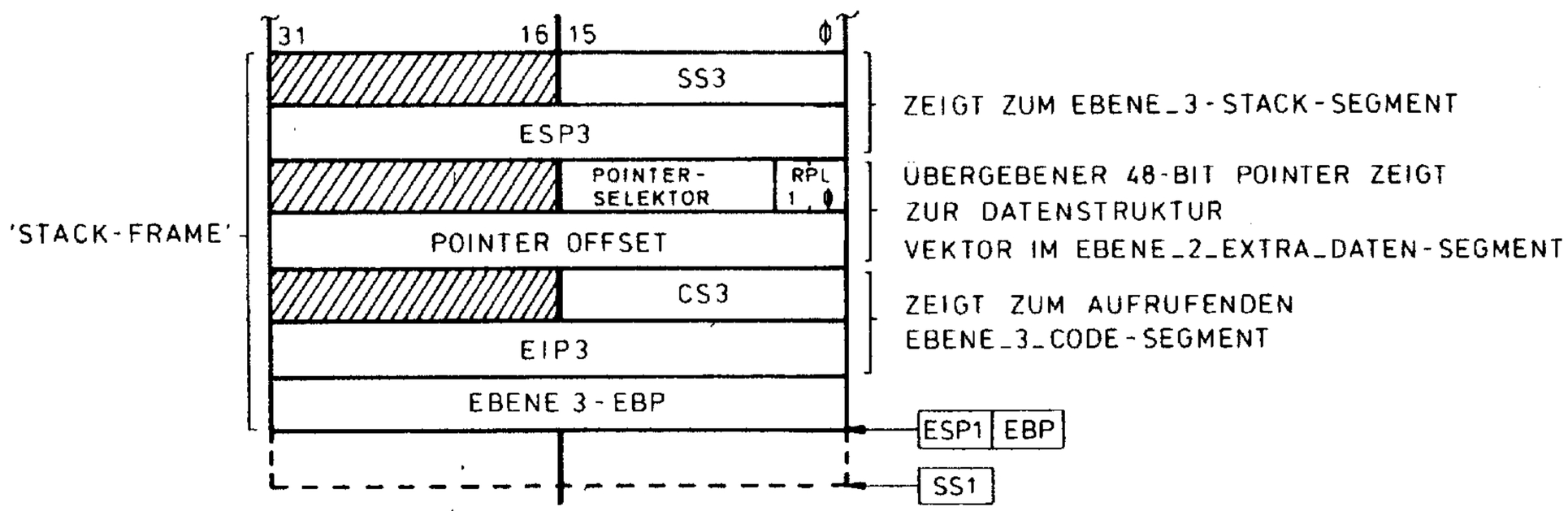
- **PUSH DWORD PTR POINTER+4**
- **PUSH DWORD PTR POINTER**

im Ebene-3-Stack und ruft danach mit

- **CALL CALL_GATE**

über einen „Call Gate"-Deskriptor, die FAR-Prozedur B im EBENE_1_CODE-Segment in der Privilegebene 1 auf.

Das Ebene 1-Stack-Layout nach Aufruf der Prozedur B zeigt das folgende Bild:



Damit der Zugriff auf den 48-Bit-Pointer in der Prozedur B im EBENE_1_CODE-Segment bequem programmiert werden kann, ist das Ebene-1-Stack-Layout als ASM386/486-Struktur mit dem Namen STACK_FRAME vereinbart.

```
                STACK_FRAME          STRUC
[EBP].--------> EBENE_3_EBP          DD ?     ;Ebene 3-EBP
        |       EIP3                 DD ?     ;Rücksprung-EIP
        |       CS3                  DD ?     ;Rücksprung-CS
        |                                     ;(Höherwertige 16 Bits sind
        |                                     ;undefiniert)
        |-------> POINTER_OFFSET     DD ?     ;Offset des Pointer-Parameters
        |-------> POINTER_SELEKTOR   DD ?     ;Selektor des Pointer-Parameters
                                              ;(Höherwertige 16 Bits sind
                                              ;undefiniert)
                ESP3                 DD ?     ;Ebene 3-ESP
                SS3                  DD ?     ;Ebene 3-SS
                                              ;(Höherwertige 16 Bits sind
                                              ;undefiniert)
                STACK_FRAME          ENDS
```

Die Übernahme des Pointers aus dem Ebene-1-Stack geschieht dann mit

- **MOV EBX,[EBP].POINTER_OFFSET**
- **MOV ES, WORD PTR [EBP]. POINTER_SELEKTOR**

Dadurch wird EBX mit dem 32-Bit-Offset von VEKTOR und ES mit dem 16-Bit-Selektor des EBENE_2_EXTRA_DATEN-Segments geladen.

Dem Pointer-Selektor mit dem manipulierten RPL-Wert 2 ist es erlaubt, den Deskriptor des EBENE_2_EXTRA_DATEN-Segments mit DPL = 2 in das ES-Cache-Register zu laden (DPL ⩾ RPL). Da sich das EBENE_1_CODE-Segment in der Privilegebene 1 befindet (CPL = 1), kann nun die darin enthaltene Prozedur B uneingeschränkt auf die Datenstruktur VEKTOR im EBENE_2_EXTRA_DATEN-Segment zugreifen (DPL ⩾ CPL). Ein solcher Zugriff ist im Programm-Fragment durch

- **MOV AL,ES:BYTE PTR [EBX]**

repräsentiert.

3.17.2 Anwendung des ARPL-Befehls

Um das Problem des unberechtigten Segment-Zugriffs durch beabsichtigte oder unbeabsichtigte Privileg-Verletzung zu vermeiden, müssen höher privilegierte Code-Segmente die Möglichkeit haben, die Privilegebene des übergebenen Selektors auf das niedrigere Niveau des aufrufenden Code-Segments zu reduzieren.

Wie das obige Beispiel zeigte, sind Zugriffe auf das EBENE_2_EXTRA_DATEN-Segment nur deswegen möglich, weil die Bedingung Pointer-Selektor RPL ⩽ Deskriptor DPL

(2 = 2) erfüllt ist. Wenn das RPL-Feld des Pointer-Selektor mit der Privilegebenen-Kennziffer des aufrufenden Code-Segments initialisiert werden könnte (im Beispiel mit dem Wert 3), wäre die oben genannte Bedingung ***nicht*** mehr erfüllt. Benützt das aufrufende Code-Segment diesen ***angepaßten*** Selektor, stellt dies eine Ausnahmesituation dar und führt zur Unterbrechung des Programms.

Um eine solche Privileg-Anpassung des übergebenen Pointer-Selektors zu realisieren, stellt die 80486-CPU den Befehl

- **ARPL SELEKTOR_A,SELEKTOR_B; (Adjust RPL Field of Selector)**

zur Verfügung.

Dieser Befehl hat zwei Operanden:

- Der erste Operand (SELEKTOR_A) ist eine 16-Bit-Speichervariable oder ein Wort-Register und enthält einen Selektor, dessen RPL-Feld angepaßt werden soll.
- Der zweite Operand (SELEKTOR_B) ist ein Wort-Register und enthält ebenfalls einen Selektor, dessen RPL-Feld den Anpassungswert liefert.

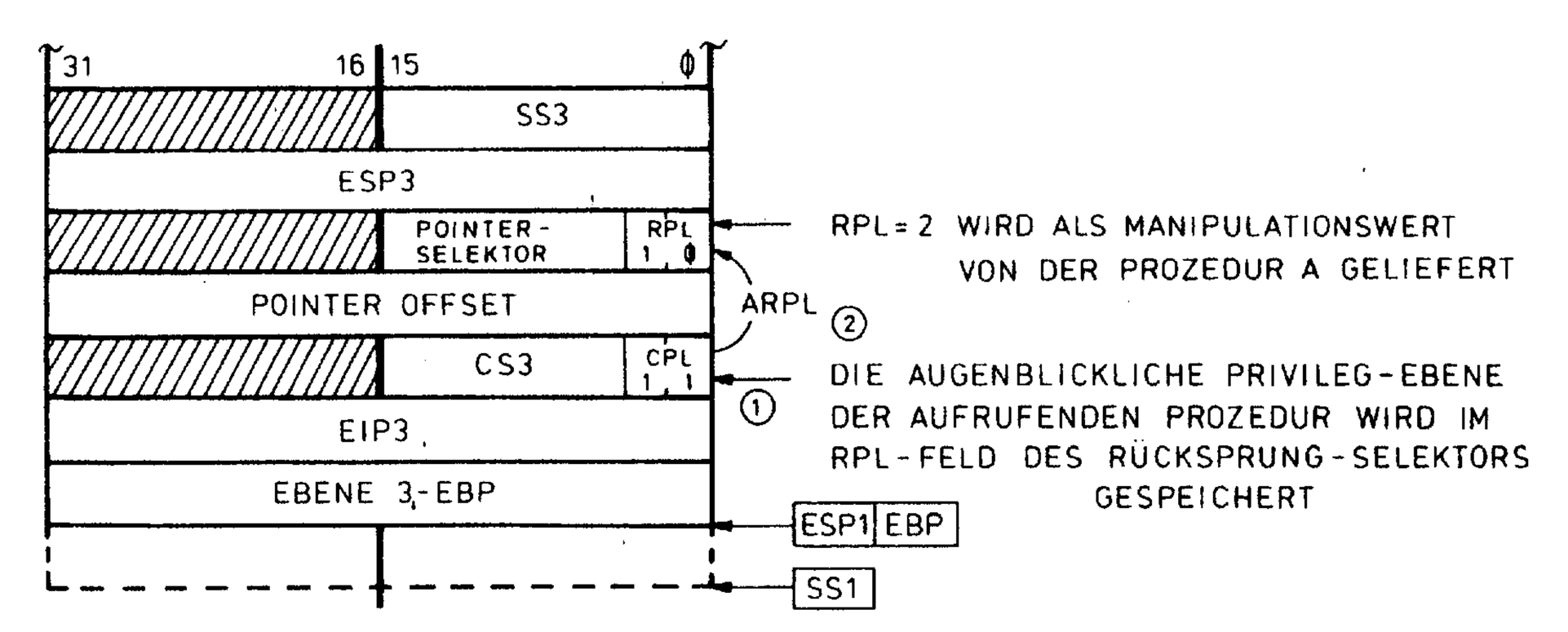
Wenn das RPL-Feld des ersten Operanden ***kleiner*** ist als das RPL-Feld des zweiten Operanden, wird das

- „Zero Flag" auf 1 gesetzt, und
- das RPL-Feld des ersten Operanden bekommt den RPL-Wert des zweiten Operanden.

Wenn das RPL-Feld des ersten Operanden ***größer*** ist als das RPL-Feld des zweiten Operanden, wird das

- „Zero Flag" auf 0 gesetzt, und
- das RPL-Feld des ersten Operanden behält seinen Wert.

Wie der ARPL-Befehl zu benützen ist, soll anhand des Programmierbeispiels einer Privileg-Verletzung erläutert werden. Wie bereits gezeigt worden ist, ergibt sich nach dem Aufruf der Prozedur B das folgende Ebene-1-Stack-Layout:



Es ist ersichtlich, daß die augenblickliche Privilegebene (CPL = 3) der aufrufenden Prozedur A (im EBENE_3_CODE-Segment) im RPL-Feld des Rücksprung-Selektors CS3 gespeichert ist ①.
Diesen Wert kann ARPL benützen, um damit das RPL-Feld des Pointer-Selektors zu überschreiben ②.

Um den Rücksprung-Selektor CS3 und den Pointer-Selektor im Ebene-1-Stack zu erreichen, kann die bereits definierte Struktur STACK_FRAME benützt werden:

```
                STACK_FRAME          STRUC
[EBP].--------> EBENE_3_EBP          DD  ?      ;Ebene 3-EBP
                EIP3                 DD  ?      ;Rücksprung-EIP
        -------> CS3                 DD  ?      ;Rücksprung-CS
                                                ;(Höherwertige 16 Bits sind
                                                ;undefiniert)
                POINTER_OFFSET       DD  ?      ;Offset des Pointer-Parameters
        -------> POINTER_SELEKTOR    DS  ?      ;Selektor des Pointer-Parameters
                                                ;(Höherwertige 16 Bits sind
                                                ;undefiniert)
                ESP3                 DD  ?      ;Ebene 3-ESP
                SS3                  DD  ?      ;Ebene 3-SS
                                                ;(Höherwertige 16 Bits sind
                                                ;undefiniert)
                STACK_FRAME          ENDS
```

Dabei ist mit WORD PTR [EBP].CS3 der Rücksprung-Selektor CS3 mit CPL = 3 und mit WORD PTR [EBP].POINTER_SELEKTOR der Pointer-Selektor mit RPL = 2 zu erreichen.

Um nun sicher zu gehen, daß RPL des Pointer-Selektors numerisch ***nicht*** kleiner ist als CPL der aufrufenden Prozedur, kann die folgende Form von Anweisungen benützt werden:

- **MOV AX,WORD PTR [EBP].CS3** ;(AX) = Rücksprung-Selektor
- **ARPL WORD PTR [EBP].POINTER_SELEKTOR,AX**

;RPL-Feld des Rücksprung-
;Selektors ***liefert*** den
;Anpassungswert 3
;RPL-Feld des Pointer-
;Selektors ***bekommt*** den
;Anpassungswert 3

Es sei darauf hingewiesen, daß ARPL in der Betriebssystem-Software erscheint und ***nicht*** in Anwenderprogrammen. Außerdem wird ARPL im „Real Address Mode" ***nicht*** erkannt.

Er sollte von jeder Betriebssystem-Prozedur verwendet werden, zumal er einen preiswerten Schutz gegen eine versehentliche Benützung ungültiger Parameter bietet.

3.17.3 **Programmierbeispiel:** Privileg-Anpassung

Das folgende Programm-Fragment entspricht dem Programmierbeispiel „Privileg-Verletzung" und soll zeigen, wie eine privilegierte Datenstruktur durch den ARPL-Befehl gegen den Zugriff eines ungültigen Pointer-Parameters geschützt werden kann.

```
LOC        OBJ                LINE  SOURCE

                               1          NAME    B13
                               2
                               3   ASSUME DS:EBENE_3_DATEN,ES:EBENE_2_EXTRA_DATEN
                               4
                               5   EXTRN CALL_GATE:FAR
                               6
--------                       7   STACK_FRAME   STRUC
00000000                       8   EBENE_3_EBP        DD   ?          ;Ebene 3-EBP
00000004                       9   EIP3               DD   ?          ;Ruecksprung-EIP
00000008                      10   CS3                DD   ?          ;Ruecksprung-CS
                              11                                      ;(Hoeherwertige 16 Bits sind
                              12                                      ;undefiniert)
0000000C                      13   POINTER_OFFSET     DD   ?          ;Offset des Pointer-Parameters
00000010                      14   POINTER_SELEKTOR   DD   ?          ;Selektor des Pointer-Parameters
                              15                                      ;(Hoeherwertige 16 Bits sind
                              16                                      ;undefiniert)
00000014                      17   ESP3               DD   ?          ;Ebene 3-ESP
00000018                      18   SS3                DD   ?          ;Ebene 3-SS
                              19                                      ;(Hoeherwertige 16 Bits sind
                              20                                      ;undefiniert)
--------                      21   STACK_FRAME   ENDS
                              22
--------                      23   EBENE_1_CODE   SEGMENT E0 USE32    ;Nur ausfuehrbares Segment mit
                              24                                      ;Basisadresse=1000H und DPL=1
                              25                                      ;(Festlegung durch BLD386/486)
                              26
00000000                      27   B       PROC     FAR  WC(2)        ;48-Bit Pointer (2 Doppelworte)
                              28                                      ;an den Ebene 1-Stack uebergeben
00000000 55                   29          PUSH     EBP               ;Rette EBP des Ebene 3-Stack
00000001 8BEC                 30          MOV      EBP,ESP           ;Stelle EBP an die Spitze des
                              31                                      ;Ebene 1-Stack
00000003 668B4508             32          MOV AX,WORD PTR [EBP].CS3  ;(AX)=16 Bit-Ruecksprung-Selektor
                              33                                      ;CPL-Feld liefert Anpassungswert
00000007 634510               34          ARPL     WORD PTR [EBP].POINTER_SELEKTOR,AX
                              35                                      ;RPL-Feld des Pointer-Selektors
                              36                                      ;bekommt den Anpassungswert
0000000A 7409                 37          JZ RPL_ANPASSUNG           ;Sprung, wenn RPL des Pointer-
                                                                      Selektors
                              38                                      ;angepasst worden ist (Zero Flag = 1)
                              39                                      ;Erfolgte keine Anpassung, kann
                              40                                      ;Prozedur B auf den Vektor zugreifen
0000000C 8B5D0C               41          MOV EBX,[EBP].POINTER_OFFSET ;(EBX)=32 Bit-Pointer-Offset
0000000F 8E4510               42          MOV ES,WORD PTR [EBP].POINTER_SELEKTOR
                              43                                      ;(ES)=16 Bit-Pointer-Selektor
00000012 268A03               44          MOV AL,ES:BYTE PTR[EBX]    ;(AL)=1.tes Element von Vektor
                              45          ;
                              46          ;
00000015                      47   RPL_ANPASSUNG:                    ;Prozedur A kann Prozedur B nicht
```

```
                                  48                                  ;fuer einen Zugriff auf den VEKTOR
                                  49                                  ;benuetzen, daher Prozedur B
                                                                      verlassen
00000015 5D                       50          POP EBP                 ;Lade EBP mit dem alten Ebene 3-
                                                                      Wert
00000016 CA0800                   51          RET 8                   ;Entferne 48-Bit Pointer aus dem
                                  52                                  ;Ebene 1-Stack
00000019                          53  B       ENDP
                                  54
--------                          55  EBENE_1_CODE   ENDS
                                  56
--------                          57  EBENE_2_EXTRA_DATEN SEGMENT     ;Les- und schreibbares Segment mit
                                      RW USE32
                                  58                                  ;Basis-Adresse 2000H und DPL=2
                                  59                                  ;(Festlegung durch BLD386/486)
00000000 (10                      60  VEKTOR          DB    10 DUP(?) ;Vektor mit 10 Byte-Elementen
         ??
         )
                                  61
--------                          62  EBENE_2_EXTRA_DATEN ENDS
                                  63
--------                          64  EBENE_3_DATEN SEGMENT RW USE32   ;Les- und schreibbares Segment mit
                                  65                                  ;Basisadresse 3000H und DPL=3
                                  66                                  ;(Festlegung durch BLD386/486)
00000000 ????????????????         67  POINTER         DQ    ?         ;Der an die Prozedur B zu ueber-
                                                                      gebende
                                  68                                  ;48-Bit Pointer-Parameter ist in
                                  69                                  ;einem Quadword gespeichert
00000008 (10                      70  EBENE_2_DATEN DB    10 DUP(?)   ;Von Vektor uebernommene Elemente
         ??
         )
                                  71
--------                          72  EBENE_3_DATEN ENDS
                                  73
--------                          74  EBENE_3_CODE   SEGMENT EO USE32  ;Nur ausfuehrbares Segment mit
                                  75                                  ;Basisadresse 4000H und DPL=3
                                  76                                  ;(Festlegung durch BLD386/486)
00000000                          77  A       PROC NEAR
                                  78          ;
                                  79          ;
00000000 C70500000000000000000R   80          MOV DWORD PTR POINTER,OFFSET VEKTOR ;32 Bit-POINTER-Offset =
                                                                      Anfangsadresse
                                  81                                  ;           von VEKTOR
0000000A 66C705040000001A00   R   82          MOV WORD PTR POINTER+4,0000000000011010B
                                  83                                  ;16 Bit-PONTER-Selektor mit
                                  84                                  ;           RPL=2
                                  85                                  ;           TI=0 (GDT)
                                  86                                  ;           Index=3 (Offset=18H)
00000013 FF3504000000         R   87          PUSH    DWORD PTR POINTER+4 ;Uebergabe des POINTER-Selektors (die
                                  88                                  ;oberen 16 Bits sind undefiniert) und
00000019 FF3500000000         R   89          PUSH    DWORD PTR POINTER   ;des POINTER_Offsets an
                                  90                                  ;das Stack-Segment in Ebene 1
0000001F 9A00000000- - -      E   91          CALL    CALL_Gate           ;Aufruf der Prozedur B in der
                                  92                                  ;Privileg-Ebene 1 ueber einen
                                  93                                  ;Call Gate_Deskriptor
                                  94                                  ;(Festlegung durch BLD386/486)
00000026 A208000000  R            95          MOV     EBENE_2_DATEN,AL    ;(AL)=1.tes Element von VEKTOR an
                                  96                                  ;           EBENE_2_DATEN ueber-
                                                                              geben
                                  97          ;
                                  98          ;
0000002B                          99  A       ENDP
                                 100
```

```
--------                    101   EBENE_3_CODE ENDS
                            102
                            103         END

ASSEMBLY COMPLETE, NO WARNINGS, NO ERRORS.
```

Aus dem Programm-Listing ist ersichtlich, daß in der Prozedur B vor der Übernahme des POINTER-Parameters aus dem Ebene-1-Stack mit

- **MOV AX, WORD PTR [EBP].CS3**
- **ARPL WORD PTR [EBP].POINTER_SELEKTOR,AX**

zunächst überprüft wird, ob RPL des POINTER_SELEKTOR's an CPL des Rücksprung-Selektors CS3 angepaßt werden muß. Da RPL(2) < CPL(3) ist, führt ARPL die Anpassung durch und setzt das „Zero Flag" = 1.

Da jetzt ein Zugriff auf den VEKTOR im EBENE_2_EXTRA_DATEN-Segment ***nicht*** erlaubt ist, wird der Befehl

- **JZ RPL_ANPASSUNG**

ausgeführt. Er bewirkt, daß die Prozedur B durch

- **POP EBP**
- **RET 8**

verlassen wird.

Wäre die Anpassung ***nicht*** vorgenommen worden, hätte Prozedur B den POINTER-Parameter geladen und anschließend mit

- **MOV AL, ES:BYTE PTR [EBX]**

das erste Element von VEKTOR im AL-Register zwischengespeichert.

Nach dem Verlassen der Prozedur B würde nun die Prozedur A das AL-Register benützen, um mit

- **MOV EBENE_2_DATEN, AL**

das vom EBENE_2_EXTRA_DATEN-Segment (Privilegebene 2) geholte Element in das EBENE_3_DATEN-Segment (Privilegebene 3) zu übernehmen (Trojanisches Problem).

3.18 Genaue Überprüfung von Pointer-Parametern

Während der ARPL-Befehl alleine ausreicht, um ***ungültige*** Parameter zu erkennen, kann trotzdem eine Ausnahmesituation entstehen, wenn eine Prozedur einen Parameter benützt. So könnte es z. B. vorkommen, daß die Prozedur ein Segment ***lesen*** will, obwohl der Parameter-Selektor einen Deskriptor geladen hat, der dieses Segment als ***nur ausführbar*** spezifiziert.

Da normalerweise nach dem Aufruf der Prozedur mehrere Befehle ausgeführt werden,

kann eine solche Ausnahmesituation erst dann auftreten, wenn die aufgerufene Prozedur den Parameter bekommt.

Um Parameterfehler ***frühzeitig*** zu erkennen, sollte das Betriebssystem die Pointer-Parameter mit den Befehlen

- **VERR** (Verify a Segment for Reading)
- **VERW** (Verify a Segment for Writing)
- **LAR** (Load Access Rights Byte)
- **LSL** (Load Segment Limit)

überprüfen.

Als Strategie zur genauen Überprüfung der Gültigkeit von Pointer-Parametern können folgende Aktionen vorgeschlagen werden:

- Benütze ARPL, um sicher zu sein, daß das RPL-Feld des Pointer-Parameters die Privilegebene der aufrufenden Prozedur enthält.
- Benütze VERR oder VERW, um sicher zu sein, daß das gekennzeichnete Segment in der Privilegebene der aufrufenden Prozedur erreichbar ist. VERR bestimmt auch, ob das gekennzeichnete Segment ***lesbar*** ist, da z. B. ein ***nur ausführbares*** Segment nicht gelesen werden kann. Daneben bestimmt VERW, ob das Segment ***beschreibbar*** ist.
- Benütze LAR und LSL, um sicher zu sein, daß die Offset-Komponente auf eine Position innerhalb der Segmentgrenzen zeigt. Dabei liefert LAR das Access-Byte des aktuellen Deskriptors, so daß bestimmt werden kann, ob das Segment z. B. ein „Expand Down"-Datensegment ist.

  LSL liefert das Segment-Limit-Feld des aktuellen Deskriptors. Wenn das Segment ein „Expand Down"-Datensegment ist, muß die Offset-Komponente ***größer*** als die Segment-Untergrenze sein. Bei „Expand UP"-Datensegmenten muß der Offset ***kleiner*** oder ***gleich*** der Segment-Obergrenze sein.

Die angegebene Strategie zur Überprüfung der Gültigkeit von Pointer-Parametern ist „kostspieliger" als die alleinige Benützung des ARPL-Befehls. Dabei ist es vernünftig, die Anwendung dieser Strategie auf solche Betriebssystem-Prozeduren zu beschränken, die von ***weniger*** privilegierten Anwender-Prozeduren aufgerufen werden können.

3.18.1 Anwendung der Befehle VERR und VERW

- **VERR ew** ; Verify a Segment for Reading
- **VERW ew** ; Verify a Segment for Writing

Der in beiden Befehlen angegebene Operand ew ist ein Wort-Register oder ein Word-Speicheroperand, der den Wert eines Selektors enthält.
Die Befehle bestimmen, ob das durch den Selektor gekennzeichnete Segment von der augenblicklichen Privilegebene erreichbar ist und ob das Segment gelesen oder beschrieben werden kann.

- Wenn auf das Segment ***zugegriffen*** werden kann, wird das „Zero Flag" auf 1 gesetzt.
- Wenn auf das Segment ***nicht zugegriffen*** werden kann, wird das „Zero Flag" auf 0 gesetzt.

Damit das „Zero Flag" ***gesetzt*** werden kann, müssen noch folgende Bedingungen erfüllt sein:

- Der Selektor muß den Deskriptor eines ***Code-*** oder ***Daten***-Segments kennzeichnen, wobei sich der Deskriptor in der ***GDT*** oder ***LDT*** aufhalten muß.
- Wenn der Befehl VERR benützt wird, muß das Segment ***lesbar*** sein. Dies bedeutet, daß die Zugriffsrechte des Segments durch die Attribute RW (Read-Write) oder ER (Execute-Read) oder RO (Read-Only) gekennzeichnet sein müssen.
- Wenn der Befehl VERW benützt wird, muß das Zugriffsrecht des Segments durch das Attribut RW (Read-Write) gekennzeichnet sein.
- Wenn auf ein Code-Segment zugegriffen wird, das ***lesbar*** und ***nicht „conforming"*** („non-conforming") ist, muß die Deskriptor-Privilegebene DPL dieses Segments numerisch größer oder gleich der augenblicklichen Privilegebene CPL der aufrufenden Prozedur und RPL des übergebenen Selektors sein (DPL ⩾ CPL, RPL).
- Wenn auf ein Code-Segment zugegriffen wird, das ***lesbar*** und ***„conforming"*** ist, kann die Deskriptor-Privilegebene DPL dieses Segments einen beliebigen Wert haben.

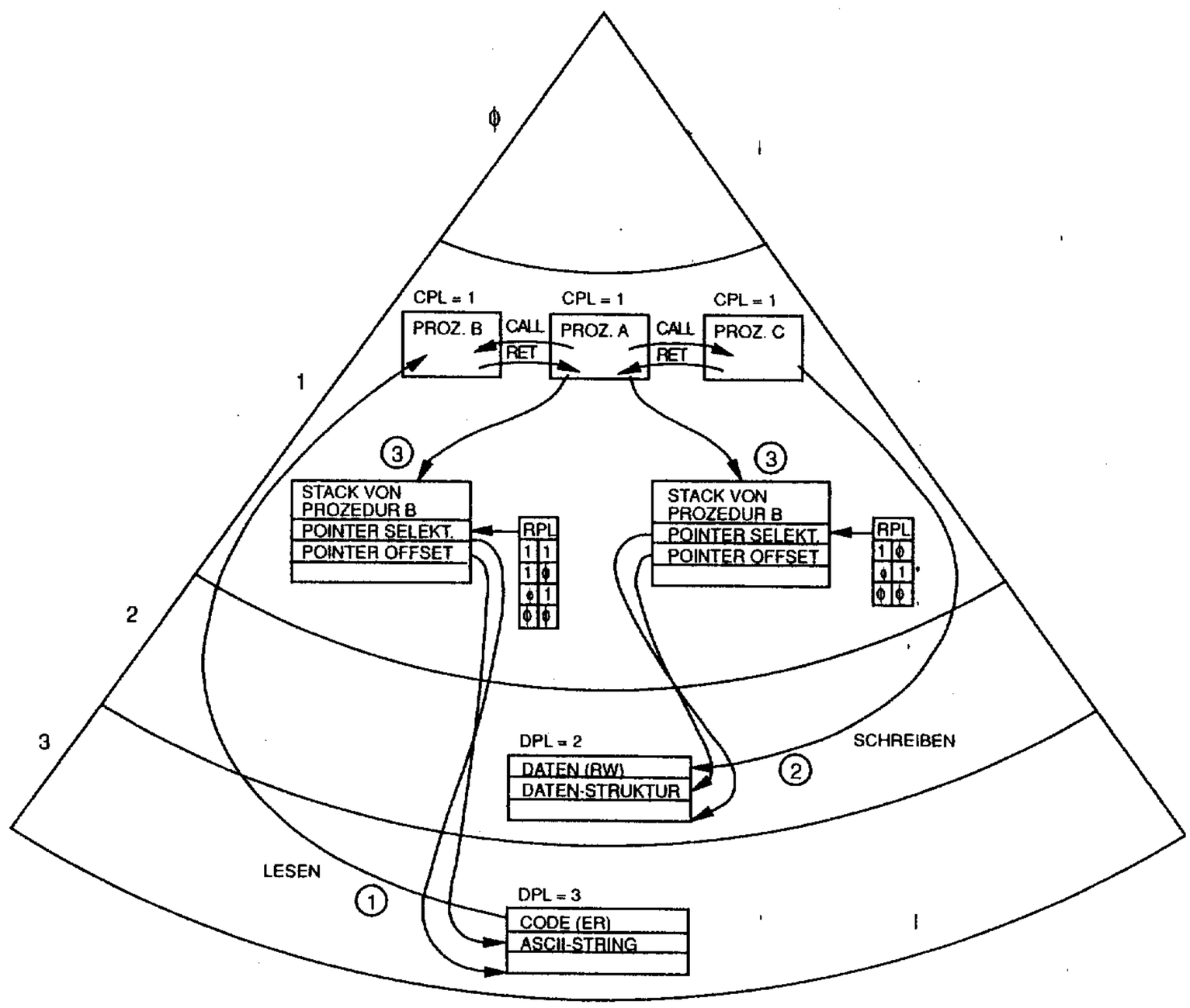
Die von VERR oder VERW geleisteten Gültigkeits-Überprüfungen sind die gleichen, die beim Laden von DS, ES, FS oder GS durchgeführt werden.
Hier einige Beispiele:

- **MOV AX,SELEKTOR_1**
  MOV DS,AX
  oder
- **MOV AX,SELEKTOR_2**
  MOV ES,AX

Der einzige Unterschied ist der, daß das Ergebnis der Gültigkeits-Überprüfung im „Zero Flag" angezeigt wird, also zu keiner Ausnahmesituation führt. Dies gibt der Software die Möglichkeit, eventuelle Segment-Zugriffsprobleme zu verhindern.

Es sei darauf hingewiesen, daß eine Programm-Unterbrechung erfolgt, wenn die Befehle VERR und VERW im Real-Address-Modus ausgeführt werden.

Um zu zeigen, wie die Befehle VERR und VERW benützt werden können, soll eine Situation vorliegen, wie sie das folgende Bild zeigt:



In der Privilegebene 1 existiert die Prozedur B, die einen konstanten ASCII-String in einem non-conforming-Code-Segment (Typ ER) in der Privilegebene 3 (DPL = 3) ***liest*** ①.

Daneben existiert in der Privilegebene 1 die Prozedur C, die das Daten-Segment in der Privilegebene 2 (DPL = 2) ***beschreibt*** ②.

Prozedur A benützt die Prozeduren B und C, um entweder auf das Code- oder das Daten-Segment zuzugreifen. Zu diesem Zweck übergibt A an B oder C passende Pointer ③.

Da z.B. in dynamischen Systemen die Prozedur A den Typ des Code- bzw. Daten-Segments nicht kennt, wird sie vor dem Aufruf der Prozeduren B oder C zunächst die zu übergebenden Pointer-Selektoren benützen, um die Zugriffsrechte festzustellen.

HINWEIS:
Dynamische Systeme sind solche, bei denen die Deskriptoren durch das Programm manipuliert werden.

Die Prozedur A wird also überprüfen, ob das Code-Segment lesbar und ob das Daten-Segment beschreibbar ist.

Setzt dabei der Befehl VERR das „Zero Flag" auf 1, kann Prozedur A die Prozedur B aufrufen, um den ASCII-String im Code-Segment zu ***lesen***.

Setzt der Befehl VERW das „Zero Flag" auf 1, kann Prozedur A die Prozedur C aufrufen, um z. B. eine Datenstruktur im Daten-Segment zu ***beschreiben***.

Aus dem obigen Bild ist ersichtlich, daß die Prozeduren A und C den CPL-Wert 1 haben. Damit das „Zero Flag" gesetzt werden kann, muß die Grundbedingung DPL $\geqslant$ (RPL, CPL) erfüllt sein. Daher kann für den Zugriff auf das Code-Segment die Prozedur A Poin-

ter-Selektoren mit RPL = 3, 2, 1 oder 0 übergeben. Für den Zugriff auf das Daten-Segment kann die Prozedur A Pointer-Selektoren mit RPL = 2, 1 oder 0 an die Prozedur C übergeben.

Das folgende Programmierbeispiel – Zugriffs-Überprüfung – enthält den ASM386/486-Code für die dargestellte Situation.
Es zeigt, daß zunächst in der Prozedur A mit VERR überprüft wird, ob das EBENE_3_CODE-Segment, das einen ASCII-String enthält, ***gelesen*** werden kann.

Ist dies der Fall, übergibt Prozedur A mit

- **PUSH DS** ;Selektor des EBENE_3_CO-
  und ;DE-Segments
- **PUSH OFFSET ASCII_STRING**

einen passenden 48-Bit-Pointer an die Prozedur B, die dann ihrerseits mit

- **MOV AL, DS:BYTE PTR [EBX]**

den ASCII-String liest. Das EBENE_3_CODE-Segment kann deswegen DS-relativ (auch ES-, FS- oder GS-relativ) gelesen werden, weil es vom Typ ER (Execute-Read) ist.

Anschließend überprüft die Prozedur A mit VERW, ob das EBENE_2_DATEN-Segment ***beschreibbar*** ist.
Ist dies der Fall, übergibt Prozedur A mit

- **PUSH DS** ;Selektor des EBENE_2_DA-
  und
- **PUSH OFFSET VEKTOR** ;TEN-Segments

einen passenden 48-Bit-Pointer an die Prozedur C, die dann ihrerseits mit

- **MOV BYTE PTR [EBX], AL**

in die Datenstruktur VEKTOR schreibt.

Nur dann, wenn Prozedur B auf das EBENE_3_CODE-Segment ***lesend*** und Prozedur C auf das EBENE_2_DATEN-Segment ***schreibend*** zugreifen kann, wird ein Programm-Pfad durchlaufen, der ein Zeichen von ASCII_STRING nach VEKTOR kopiert.

3.18.2 **Programmierbeispiel:** Zugriffs-Überprüfung

```
LOC       OBJ                  LINE    SOURCE

                                1            NAME    B14
                                2
                                3    ASSUME DS:EBENE_2_DATEN
                                4
--------                        5    STACK_FRAME    STRUC
                                6
00000000                        7    PROZEDUR_A_EBP    DD    ?          ;Prozedur A-EBP
00000004                        8    EIP1              DD    ?          ;Ruecksprung-EIP
00000008                        9    CS1               DD    ?          ;Ruecksprung-CS
                               10                                       ;(Hoeherwertige 16 Bits sind
                               11                                       ;undefiniert)
```

```
0000000C                    12  POINTER          DP    ?            ;48-Bit Pointer-Parameter
00000012                    13                   DW    ?            ;Die hoeherwertigen 16 Bits des
                            14                                       ;Pointer-Selektors sind undefiniert
--------                    15  STACK_FRAME      ENDS
                            16
--------                    17  EBENE_1_CODE     SEGMENT EO USE32 ;Nur ausfuehrbares Segment mit
                            18                                       ;Basisadresse=1000H und DPL=1
                            19                                       ;(Festlegung durch BLD386/486)
00000000                    20  A      PROC      NEAR
                            21
00000000 66B8000090 ---- R  22        MOV       AX,EBENE_3_CODE      ;(AX)=Selektor von EBENE_3_CODE
00000007 8ED8               23        MOV       DS,AX                ;Selektor laden
00000009 0F00E0             24        VERR      AX                   ;Ist EBENE_3_CODE-Segment lesbar
0000000C 7400               25        JZ        LESBAR               ;Sprung, wenn lesbar (Zero Flag=1)
                            26        ;
                            27        ;
0000000E                    28  LESBAR:
0000000E 1E                 29        PUSH      DS                   ;16 Bit-Pointer-Selektor als
                            30                                       ;Doubleword und
0000000F 6800000000      R  31        PUSH      OFFSET ASCII_STRING  ;32 Bit-Pointer-Offset in
                            32                                       ;Ebene 1-Stack laden
00000014 9A00000000 ---- R  33        CALL      FAR PTR B            ;Aufruf von Prozedur B inter-segment
                            34                                       ;direkt
0000001B 66B900009090 ---- R 35       MOV       CX,EBENE_2_DATEN     ;(CX)=Selektor von EBENE_2_DATEN
00000023 8ED9               36        MOV       DS,CX                ;Selektor laden
00000025 0F00E9             37        VERW      CX                   ;Ist EBENE_2_DATEN-Segment
                                                                     beschreibbar
00000028 7400               38        JZ        BESCHREIBBAR         ;Sprung, wenn beschreibbar
                                                                     (Zero Flag=1)
                            39        ;
                            40        ;
0000002A                    41  BESCHREIBBAR:
0000002A 1E                 42        PUSH      DS                   ;16 Bit-Pointer-Selektor als
                            43                                       ;Dounleword und
0000002B 6800000000      R  44        PUSH      OFFSET VEKTOR        ;32 Bit-Pointer-Offset in
                            45                                       ;Ebene 1-Stack laden
00000030 9A0C000000- ---- R 46        CALL      FAR PTR C            ;Aufruf von Prozedur C inter-segment
                            47                                       ;direkt
00000037                    48  A      ENDP
                            49
--------                    50  EBENE_1_CODE     ENDS
                            51
--------                    52  EBENE_1_CODE_1 Segment EO USE32      ;Nur ausfuehrbares Segment mit
                            53                                       ;Basisadresse=2000H und DPL=1
                            54                                       ;(Festlegung durch BLD386/486)
00000000                    55  B      PROC      FAR
                            56
00000000 55                 57        PUSH      EBP                  ;Rette EBP der Prozedur A
00000001 8BEC               58        MOV       EBP,ESP              ;Stelle EBP an die Spitze des
                            59                                       ;Ebene 1-Stack
00000003 C55D0C             60        LDS       EBP, [EBX].POINTER   ;(EBX)=Pointer-Offset (32 Bits)
                            61                                       ;(DS)=Pointer-Selektor (16 Bits)
00000006 8A03               62        MOV       AL,DS:BYTE PTR [EBX] ;(AL)=1.tes Zeichen von ASCII_STRING
00000008 5D                 63        POP       EBP                  ;Lade EBP mit dem alten Prozedur
                                                                     A-Wert
00000009 CA0800             64        RET       8                    ;Entferne 48-Bit Pointer aus dem
                            65                                       ;Ebene 1-Stack
0000000C                    66  B      ENDP
                            67
0000000C                    68  C      PROC      FAR
                            69
0000000C 55                 70        PUSH      EBP                  ;Rette EBP der Prozedur A
0000000D 8BEC               71        MOV       EBP,ESP              ;Stelle EBP an die Spitze des
```

```
                                  72                                        ;Ebene 1-Stack
0000000F C55D0C                   73          LDS     EBX, [EBP].POINTER    ;(EBX)=Pointer-Offset (32 Bits)
                                  74                                        ;(DS)=Pointer-Selektor (16 Bits)
00000012 8803                     75          MOV     BYTE PTR[EBX],AL      ;(AL) in 1.tes Element von VEKTOR
00000014 5D                       76          POP     EBP                   ;Lade EBP mit dem alten Prozedur
                                                                            A-Wert
00000015 CA0800                   77          RET     8                     ;Entferne 48-Bit Pointer aus dem
                                  78                                        ;Ebene 1-Stack
                                  79
00000018                          80    C     ENDP
                                  81
--------                          82    EBENE_1_CODE_1 ENDS
                                  83
--------                          84    EBENE_2_DATEN  SEGMENT RW USE32     ;Schreib- und lesbares Segment mit
                                  85                                        ;Basisadresse=3000H und DPL=2
                                  86                                        ;(Festlegung durch BLD386/486)
00000000 (10                      87    VEKTOR         DB  10 DUP(?)        ;Vektor mit 10 Byte-Elementen
         ??
         )
                                  88
--------                          89    EBENE_2_DATEN  ENDS
                                  90
--------                          91    EBENE_3_CODE   SEGMENT ER USE32     ;Ausfuehrbares und lesbares Segment
                                  92                                        ;mit Basisadresse=4000H und DPL=3
                                  93                                        ;(Festlegung durch BLD386/486)
                                  94          ;
                                  95          ;                             ;Befehle
                                  96          ;
00000000 4C6573656E               97    ASCII_STRING   DB 'Lesen'           ;Zu lesender ASCII-String
                                  98
--------                          99    EBENE_3_CODE   ENDS
                                  100
                                  101         END
ASSEMBLY COMPLETE, NO WARNINGS, NO ERRORS.
```

3.18.3 Anwendung der Befehle LAR und LSL

1. LAR – Load Access Rights Byte

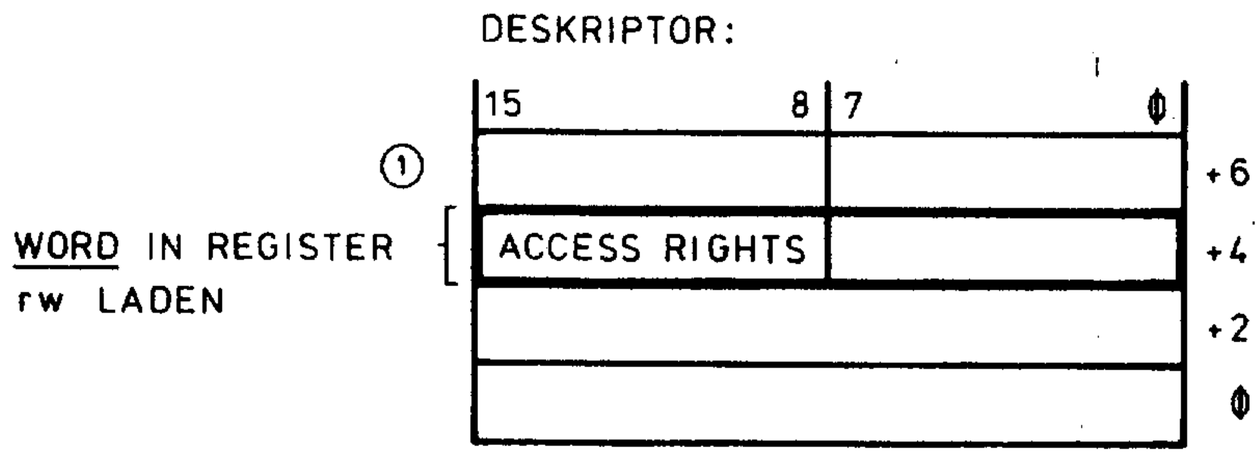
Der 80486 unterscheidet beim LAR-Befehl zwischen der 16-Bit-Form und der 32-Bit-Form.

16-Bit-Form:

- **LAR rw,ew** ;Load Access Rights Byte

Der im 16-Bit-LAR-Befehl angegebene zweite Operand ***ew*** ist ein ***Word-Register*** oder ***Word-Speicher-Operand,*** der den Wert eines Selektors enthält. Der erste Operand ***rw*** ist ein ***Word-Register***.

Wenn der in ***ew*** gespeicherte Selektor einen Deskriptor erreicht, lädt die 16-Bit-Form dieses Befehls das „Word" bei Offset + 4 im selektierten Deskriptor in das Register ***rw*** ①. Wie das Format eines Deskriptors zeigt, ist in den höherwertigen 16 Bits dieses „Words" das „Access Rights"-Byte enthalten.



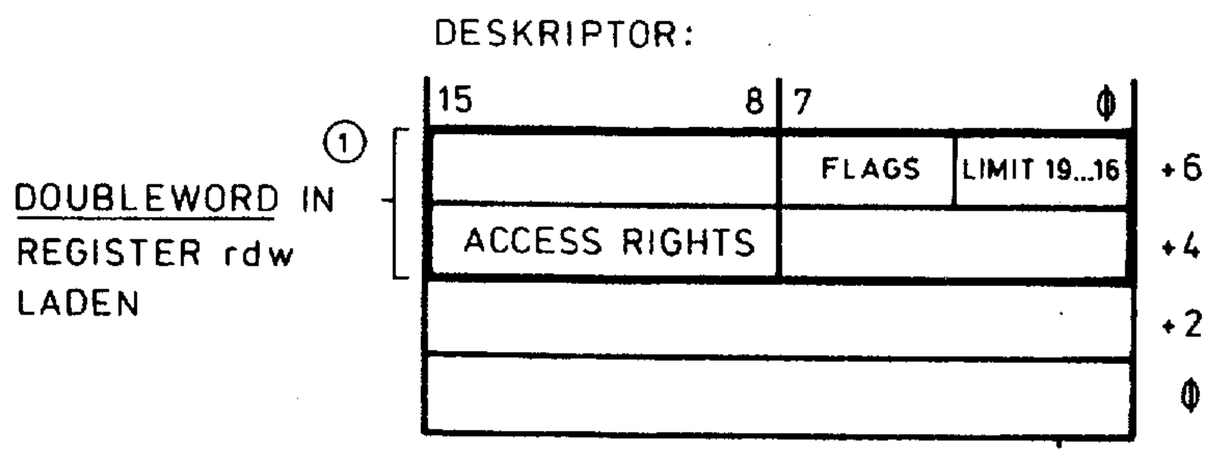
Nach dem Ladevorgang maskiert der 16-Bit-LAR-Befehl das Register ***rw*** mit FF00H. Dies hat zur Folge, daß das XL-Feld (z. B. AL, BL) von ***rw gelöscht*** wird, während im XH-Feld (z. B. AH, BH) von ***rw*** die ***Zugriffsrechte*** des Deskriptors zur Verfügung stehen.

32-Bit-Form:

- **LAR rdw,edw** ;Load Access Rights Byte

Der im 32-Bit-LAR-Befehl angegebene zweite Operand ***edw*** ist ein ***Doubleword-Register*** oder ein ***Doubleword-Speicher-Operand***, der den Wert eines Selektors enthält. Der erste Operand ist ein ***Doubleword-Register***.

Wenn der in ***edw*** gespeicherte Selektor einen Deskriptor erreicht, lädt die 32-Bit-Form dieses Befehls das „Doubleword" bei Offset + 4 im selektierten Deskriptor in das Register ***rdw*** ①.



Nach dem Ladevorgang maskiert der 32-Bit-LAR-Befehl das Register ***rdw*** mit 00FXFF00H. Dies hat zur Folge, daß in ***rdw*** die Deskriptor-***Flags*** und die ***Zugriffsrechte*** des Deskriptors zur Verfügung stehen. Dabei sind die Zugriffsrechte im XH-Feld (z. B. AH, BH) von ***rdw*** enthalten. Das X in der Maske korrespondiert mit LIMIT 19 ... 16 und gibt an, daß dieses Feld nach dem Laden einen undefinierten Wert enthält.

Bei beiden Formen von LAR wird das „Zero Flag" auf 1 gesetzt, wenn

- der Selektor ***nicht*** Null ist,
- der Selektor-Index ***innerhalb*** der Tabellen-Grenze liegt,
- die Privileg-Ebene DPL des ausgewählten Deskriptors ***größer*** oder ***gleich*** der augenblicklichen Privileg-Ebene CPL ist (DPL ≥ CPL; wie Datenzugriff) und die Privileg-Ebene des ausgewählten Deskriptors ***größer*** oder ***gleich*** dem RPL-Wert des Selektors ist (DPL ≥ Selektor RPL; wie Datenzugriff).

Sind diese Bedingungen nicht erfüllt, wird das „Zero Flag" gelöscht (ZF = 0).
Wenn diese Befehle im „Real Address Mode" ausgeführt werden, erfolgt eine Programm-Unterbrechung.

HINWEIS:
Alle Code- und Daten-Segment-Deskriptoren sind für die LAR-Befehle ***gültig***.
Welche speziellen Segment- und Gate-Deskriptoren für LAR ebenfalls gültig sind, sind in der folgenden Tabelle zusammengefaßt:

Typ	Name	gültig/ungültig
0	Ungültig	Ungültig
1	Verfügbares 80286-TSS	Gültig
2	LDT	Gültig
3	Busy 80286-TSS	Gültig
4	80286 call gate	Gültig
5	80286/80486 task gate	Gültig
6	80286 trap gate	Gültig
7	80286 interrupt gate	Gültig
8	Ungültig	Ungültig
9	Verfügbares 80486-TSS	Gültig
A	Ungültig	Ungültig
B	Busy 80486 TSS	Gültig
C	80486 call gate	Gültig
D	Ungültig	Ungültig
E	80486 trap gate	Gültig
F	80486 interrupt gate	Gültig

2. LSL-Load Segment Limit

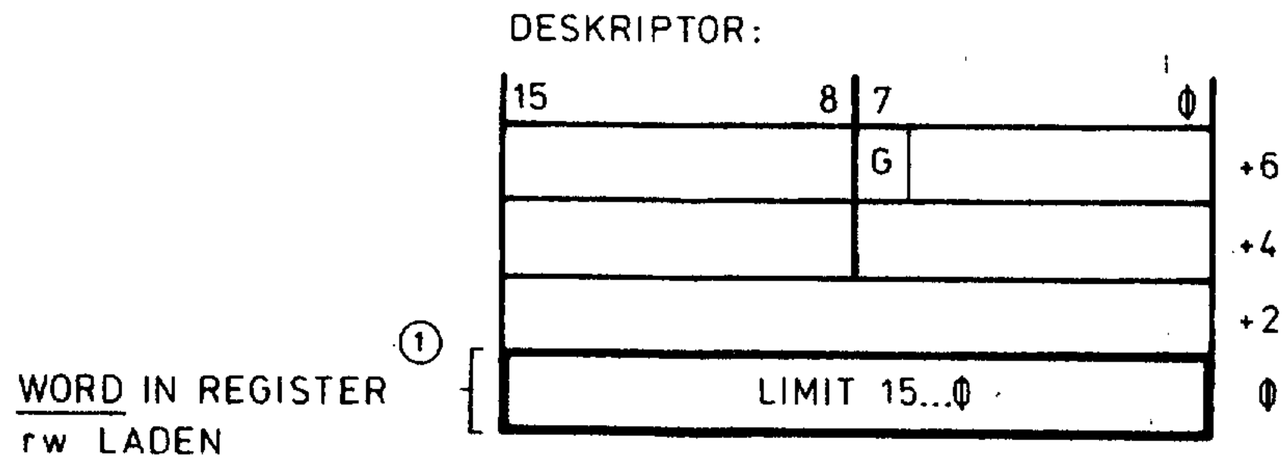
Der 80486 unterscheidet beim LSL-Befehl zwischen der 16-Bit-Form und der 32-Bit-Form, wobei er zusätzlich die Körnung (Byte oder Page) des Segments bei beiden Formen berücksichtigt.

16-Bit-Form:

- **LSL rw,ew** ; Load Segment Limit (***byte*** granular)
- **LSL rw,ew** ; Load Segment Limit (***page*** granular)

Der im 16-Bit-LSL-Befehl angegebene zweite Operand ***ew*** ist ein ***Word-Register*** oder ***Word-Speicher-Operand***, der den Wert eines Selektors enthält. Der erste Operand ***rw*** ist ein ***Word-Register***.

Wenn der in ***ew*** gespeicherte Selektor einen Deskriptor erreicht, lädt die 16-Bit-Form dieses Befehls das LIMIT-Feld 15 ... 0 des selektierten Deskriptors in das Register ***rw*** ①.



Generell gilt, daß unabhängig von der Befehlsform jeder Limit-Wert als sogenannter ***„byte-granulierter"*** Wert in das Ziel-Register geladen wird.

- ***Ist im Deskriptor das G-Bit = 0*** (byte granular), dann ist der geladene Wert identisch mit dem 16-Bit-„Roh"-Wert im LIMIT-Feld.
- ***Ist im Deskriptor das G-Bit = 1*** (page granular), übersetzt LSL vor dem Laden den „page-granulierten" Limit-Wert in einen „byte-granulierten" Wert. Dies geschieht dadurch, daß er den 16-Bit-Roh-Limitwert um 12 Bit-Stellen nach links schiebt und anschließend eine ODER-Verknüpfung mit 0FFFH durchführt. Der resultierende Wert wird dann anschließend im 16-Bit-Ziel-Register ***rw*** als „byte-granulierter" Limitwert gespeichert.

  BEISPIEL: In einer 16-Bit-Umgebung ist die maximale Segment-Länge 64K. Es soll ein Segment existieren, das in Pages eingeteilt ist (G-Bit = 1) und dessen Roh-Limitwert bei 0005H liegt. Wenn LSL die Konvertierung durchführt, ergeben sich folgende Schritte:

Roh-Limitwert	=	0005H
Roh-Limitwert um 12 Bit-Stellen nach links schieben	=	5000H
	ODER	0FFFH
„byte-granulierter" Limitwert	=	**5FFFH**

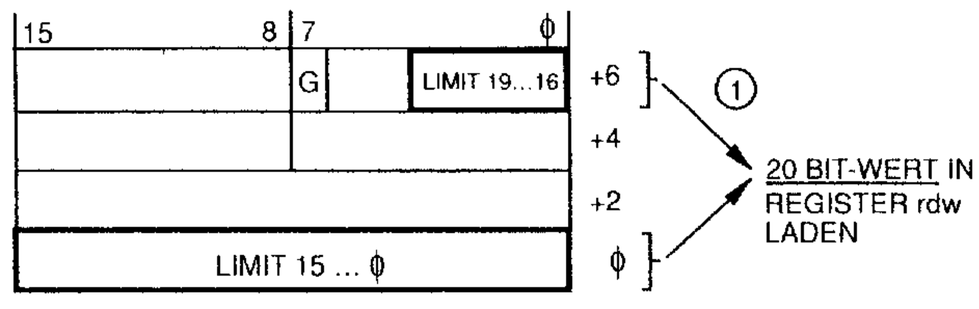
  Das Ergebnis zeigt, daß das Segment in sechs 4K-Pages (Page 0 ... Page 5) eingeteilt ist.

32-Bit-Form:

- **LSL rdw,edw** ; Load Segment Limit (***byte*** granular)
- **LSL rdw,edw** ; Load Segment Limit (***page*** granular)

Der im 32-Bit-LSL-Befehl angegebene zweite Operand ***edw*** ist ein ***Doubleword-Register*** oder ein ***Doubleword-Speicher-Operand,*** der den Wert eines Selektors enthält. Der erste Operand ***rdw*** ist ein ***Doubleword-Register.***

Wenn der in ***edw*** gespeicherte Selektor einen Deskriptor erreicht, lädt die 32-Bit-Form dieses Befehls die Felder LIMIT 15 ... 0 und LIMIT 19 ... 16 des selektierten Deskriptors in das Register ***rdw*** ①.



- ***Ist im Deskriptor das G-Bit = 0*** (byte granular), dann ist der geladene Wert identisch mit dem 20-Bit-Roh-Wert in den beiden Limit-Feldern.
- ***Ist im Deskriptor das G-Bit = 1*** (page granular), übersetzt LSL vor dem Laden den „page-granulierten" Limitwert in einen „byte-granulierten" Wert. Dies geschieht dadurch, daß er den 20-Bit-Roh-Limitwert um 12 Stellen nach links schiebt und anschließend eine ODER-Verknüpfung mit 00000FFFH durchführt. Der resultierende Wert wird dann anschließend im 32-Bit-Zielregister ***rdw*** als „byte-granulierter" Limitwert gespeichert.

Bei allen Formen von LSL wird das „Zero Flag" auf 1 gesetzt, wenn

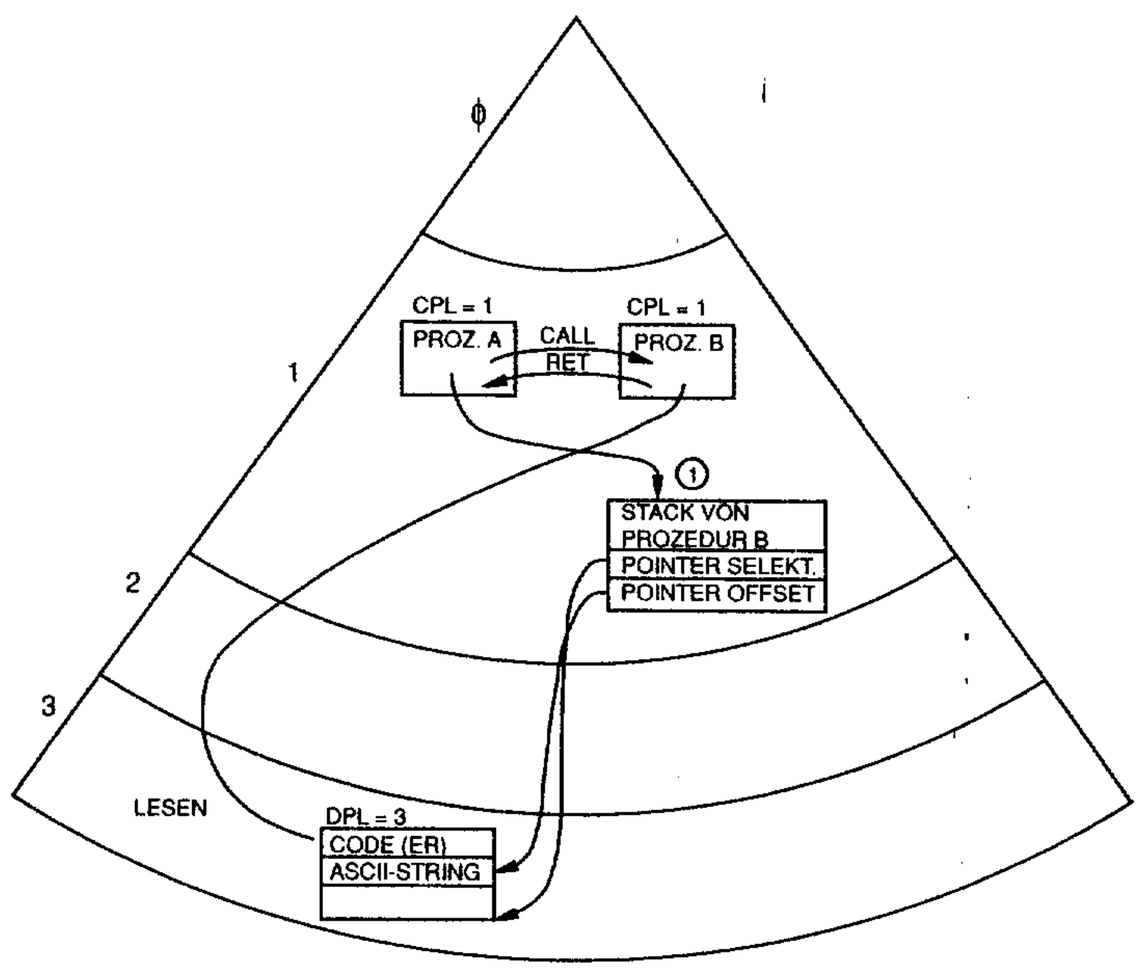
- der Selektor ***nicht*** Null ist,
- der Selektor-Index ***innerhalb*** der Tabellen-Grenze liegt,
- die Privileg-Ebene DPL des ausgewählten Deskriptors ***größer*** oder ***gleich*** der augenblicklichen Privileg-Ebene CPL ist (DPL ≥ CPL; wie Datenzugriff) und
- die Privileg-Ebene DPL des ausgewählten Deskriptors ***größer*** oder ***gleich*** dem RPL-Wert des Selektors ist (DPL ≥ Selektor RPL; wie Datenzugriff).

Sind diese Bedingungen nicht erfüllt, wird das „Zero Flag" gelöscht (ZF = 0). Wenn diese Befehle im „Real Address Mode" ausgeführt werden, erfolgt eine Programm-Unterbrechung.

HINWEIS: Alle Code- und Daten-Segment-Deskriptoren sind für die LSL-Befehle gültig.
Welche speziellen Segment- und Gate-Deskriptoren für LSL ebenfalls gültig sind, sind in der folgenden Tabelle zusammengefaßt:

Typ	Name	gültig/ungültig
0	Ungültig	Ungültig
1	Verfügbares 80286-TSS	Gültig
2	LDT	Gültig
3	Busy 80286-TSS	Gültig
4	80286 call gate	Ungültig
5	80286/80486 task gate	Ungültig
6	80286 trap gate	Ungültig
7	80286 interrupt gate	Ungültig
8	Ungültig	Ungültig
9	Verfügbares 80486-TSS	Gültig
A	Ungültig	Ungültig
B	Busy 80486 TSS	Gültig
C	80486 call gate	Ungültig
D	Ungültig	Ungültig
E	80486 trap gate	Ungültig
F	80486 interrupt gate	Ungültig

Um zu zeigen, wie die Befehle LAR und LSL benützt werden können, soll eine Situation vorliegen, wie sie das folgende Bild zeigt:



In der Privilegebene 1 existieren die Prozeduren A und B. Dabei benützt A die Prozedur B, um auf ein non-conforming-Code-Segment (Typ ER) in der Privilegebene 3 (DPL = 3) zuzugreifen.

Zu diesem Zweck übergibt Prozedur A an den Stack der Prozedur B einen passenden 48-Bit-Pointer ①, den Prozedur B benützt, um einen ASCII-String im Privileg-3-Code-Segment zu lesen.

Das folgende Programmierbeispiel – Zugriffs- und Limitüberprüfung – enthält den ASM386/486-Code für die angegebene Situation. Dabei wird der Befehl

- **LAR DX, AX** ; (DX) = Access-Byte des
  ; EBENE_3_CODE-Deskriptors

benützt, um später mit

- **TEST DH, 00000010B**

zu überprüfen, ob das „READABLE"-Bit im Access-Byte des EBENE_3_CODE-Segment-Deskriptors gesetzt ist.

Ob das Access-Byte überhaupt geladen worden ist, überprüft der Befehl

- **JNZ SELEKTOR_UNGUELTIG**

Für den Fall, daß der Selektor in AX den Deskriptor ***nicht*** erreicht, weil z. B. die Bedingung DPL ⩾ RPL, CPL ***nicht*** erfüllt ist, wird das „Zero Flag" 0 gesetzt. Der JNZ-Befehl benützt diesen Wert und bewirkt, daß die Prozedur A verlassen wird.

Konnte das Access-Byte geladen werden, wertet der Befehl

- **JZ NICHT_LESBAR**

das Ergebnis des TEST-Befehls aus.

Für den Fall, daß das „READABLE"-Bit 0 ist, kann das EBENE_3_CODE-Segment ***nicht*** gelesen werden. Der TEST-Befehl setzt das „Zero Flag" 1. Diesen Wert benützt der JZ-Befehl und bewirkt, daß die Prozedur A verlassen wird.

Ist das EBENE_3_CODE-Segment lesbar, wird anschließend mit

- **LSL EDI, EAX**

das Limit-Feld des zugehörigen Deskriptors in das EDI-Register geladen. Ob das Laden erfolgreich war, ist durch eine Überprüfung des „Zero Flags" an dieser Stelle ***nicht*** mehr notwendig. Daß der Selektor in AX bzw. in EAX den EBENE_3_CODE-Deskriptor erreichen kann, hat bereits der Befehl JNZ SELEKTOR_UNGUELTIG überprüft.

Der Befehl

- **CMP EDI, (OFFSET ASCII_STRING + SIZE ASCII_STRING-1)**

stellt nun fest, ob der Wert des Limit-Felds mindestens der Größe des EBENE_3_CODE-Segments entspricht. Wenn nein, bewirkt

- **JB LIMIT_ZU_KLEIN**

daß die Prozedur A verlassen wird.

3.18.4 **Programmierbeispiel:** Zugriffs- und Limitüberprüfung

```
LOC      OBJ              LINE  SOURCE

                           1          NAME      B15
                           2
--------                   3  STACK_FRAME       STRUC
                           4
00000000                   5  PROZEDUR_A_EBP          DD  ?      ;Prozedur A-EBP
00000004                   6  EIP1                    DD  ?      ;Ruecksprung-EIP
00000008                   7  CS1                     DD  ?      ;Ruecksprung-CS
                           8                                     ;(Hoeherwertige 16 Bits sind
                           9                                     ;undefiniert
0000000C                  10  POINTER                 DP  ?      ;48-Bit Pointer-Parameter
00000012                  11                          DW  ?      ;Die hoeherwertigen 16 Bits des
                          12                                     ;Pointer-Selektors sind undefiniert
--------                  13  STACK_FRAME       ENDS
                          14
--------                  15  EBENE_1_CODE       SEGMENT EO USE32  ;Nur ausfuehrbares Segment mit
                          16                                     ;Basisadresse=1000H und DPL=1
                          17                                     ;(Festlegung durch BLD386/486)
00000000                  18  A        PROC      NEAR
                          19
00000000 66B8000090- - - R 20         MOV       AX,EBENE_3_CODE  ;(AX)=Selektor von EBENE_3_CODE
00000007 8ED8             21          MOV       DS,AX                ;Selektor laden
00000009 660F02D0         22          LAR       DX,AX                ;(DX)=Access-Byte des EBENE_3_CODE-
                          23                                         ;Deskriptors
0000000D 751D             24          JNZ       SELEKTOR_UNGUELTIG   ;Sprung, wenn ungueltiger Selektor
                          25                                         ;(Zero Flag=0)
0000000F F6C602           26          TEST      DH,00000010B         ;Ist READABLE-Bit gesetzt?
00000012 7418             27          JZ        NICHT_LESBAR         ;Sprung, wenn READABLE-Bit=0
                          28                                         ;(Zero Flag=1)
00000014 0F03F8           29          LSL       EDI,EAX              ;(EDI)=Limit des EBENE_3_CODE-Seg-
                                                                     ments
```

```
00000017 81FF04000000   R 30        CMP     EDI,(OFFSET ASCII_STRING+SIZE ASCII_STRING-1)
                          31                                    ;Limit-Wert mit Segment-Groesse
                          32                                    ;vergleichen
0000001D 720D             33        JB  LIMIT_ZU_KLEIN          ;Sprung, wenn der Limit-Wert zu
                          34                                    ;klein ist
0000001F 1E               35        PUSH    DS                  ;16 Bit-Pointer-Selektor als
                          36                                    ;Doubleword und
00000020 68000000000    R 37        PUSH    OFFSET ASCII_STRING ;32 Bit-Pointer-Offset in
                          38                                    ;Ebene 1-Stack laden
00000025 9A00000000- - - R 39       CALL    FAR PTR B           ;Aufruf von Prozedur B inter-segment
                          40                                    ;direkt
                          41        ;
                          42        ;
0000002C                  43        SELEKTOR_UNGUELTIG:
0000002C                  44        NICHT_LESBAR:
0000002C                  45        LIMIT_ZU_KLEIN:
0000002C C3               46                RET                 ;Prozedur A verlassen
                          47
0000002D                  48        A       ENDP
                          49
--------                  50        EBENE_1_CODE        ENDS
                          51
--------                  52        EBENE_1_CODE_1      Segment EO USE32    ;Nur ausfuehrbares Segment mit
                          53                                    ;Basisadresse=2000H und DPL=1
                          54                                    ;(Festlegung durch BLD386/486)
00000000                  55        B       PROC    FAR
                          56
00000000 55               57                PUSH    EBP         ;Rette EBP der Prozedur A
0000001  8BEC             58                MOV     EBP,ESP     ;Stelle EBP an die Spitze des
                          59                                    ;Ebene 1-Stack
00000003 C55D0C           60                LDS     EBX,[EBP].POINTER   ;(EBX)=Pointer-Offset (32 Bits)
                          61                                    ;(DS)=Pointer-Selektor (16 Bits)
00000006 8A03             62                MOV         AL,DS:BYTE PTR[EBX] ;(AL)=1.tes Zeichen von ASCII_STRING
00000008 5D               63                POP     EBP         ;Lade EBP mit dem alten Prozedur
                                                                A-Wert
00000009 CA0800           64                RET     8           ;Entferne 48-Bit Pointer aus dem
                          65                                    ;Ebene 1-Stack
0000000C                  66        B       ENDP
                          67
--------                  68        EBENE_1-CODE_1  ENDS
                          69
--------                  70        EBENE_3_CODE    SEGMENT ER USE32    ;Ausfuehrbares und lesbares Segment
                          71                                    ;mit Basisadresse=4000H und DPL=3
                          72                                    ;(Festlegung durch BLD386/486)
                          73                ;
                          74                ;                   ;Befehle
                          75                ;
00000000 4C6573656E       76        ASCII_STRING DB 'LESEN'     ;Zu lesender ASCII-String
                          77
--------                  78        EBENE_3_CODE    ENDS
                          79
                          80                END
ASSEMBLY COMPLETE, NO WARNINGS, NO ERRORS.
```

3.19 Kombination von Segment- und Page-Schutz

Jedes 80486-Betriebssystem benutzt eines der Segmentierungsmodelle. Im Gegensatz dazu kann Paging optional unterhalb der Segmentierung implementiert werden.

Ein individuelles Segment enthält entweder Code oder Daten und ist daher durch den Typ

seines Inhalts charakterisiert. Eine solche Charakterisierung gibt es für Pages nicht. ***Alle*** Pages sind grundsätzlich

- ***ausführbar*** (executable), wenn sie Code enthalten und
- ***nur-lesbar*** (read only), wenn sie Daten enthalten.

Sie können allerdings mit sogenannten Page-Schutz Attributen versehen werden, die sie entweder

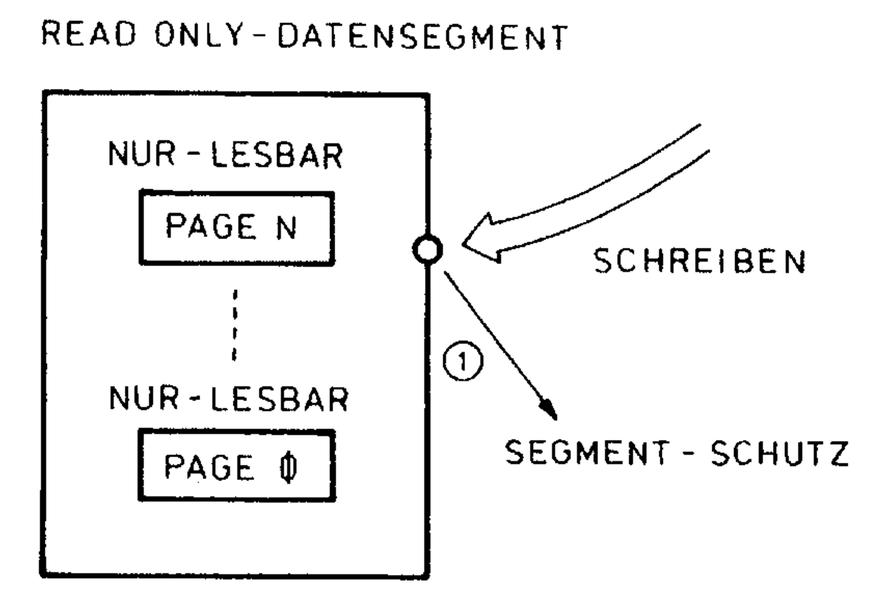
- vor unautorisierten Zugriffen bewahren oder
- beschreibbar machen.

Wenn Pages innerhalb eines Segments definiert sind, stehen die Page-Schutz-Attribute in enger Beziehung mit den Segment-Schutz-Attributen.
Ist der Paging-Mechanismus freigegeben, führt die ***M***emory ***M***anagement ***U***nit (MMU) des 80486 die Segment- und Page-Übersetzung ***parallel*** aus.

Bevor der Zugriff auf eine Page innerhalb des Segments erfolgt, prüft der Prozessor

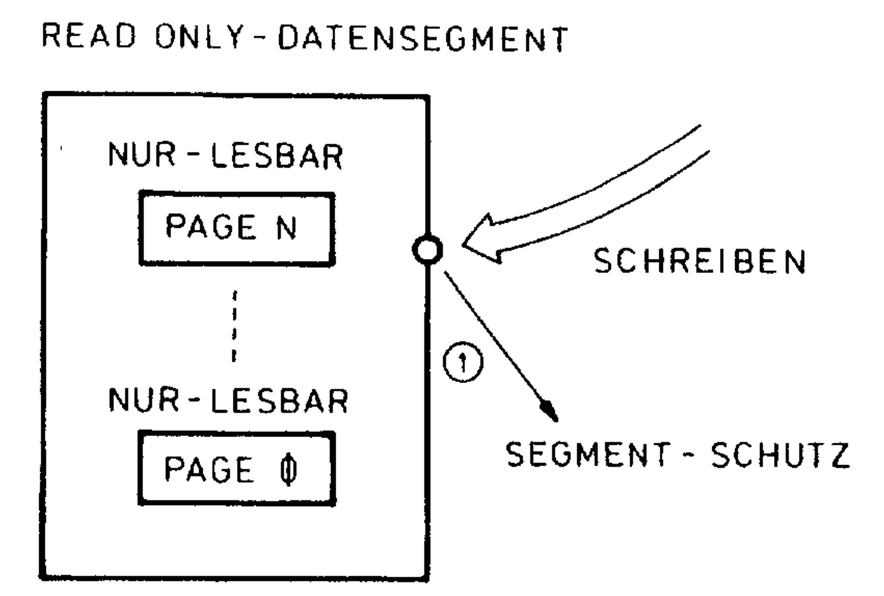
- ***zuerst*** die Segment-Schutz-Attribute
- ***dann*** die Page-Schutz-Attribute.

BEISPIEL 1: Es soll ein Read Only-Datensegment existieren, in dem ***nur-lesbare*** Pages definiert sind. Ein solches Segment hat das ***effektive*** Schutz-Attribut „Read Only".



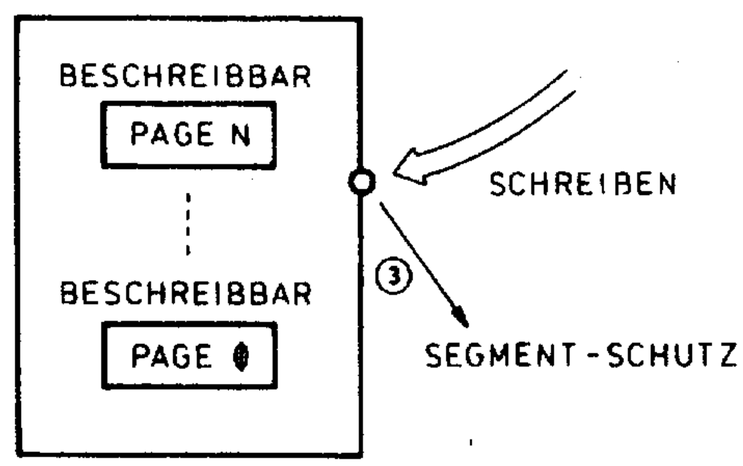
Wenn nun versucht wird, in ein solches Segment zu schreiben, verhindert bereits der Segment-Schutz den Zugriff auf eine Page ①. Eine anschließende Überprüfung der Page-Schutz-Attribute wird dann nicht mehr durchgeführt.

BEISPIEL 2: Es soll ein Read/Write-Datensegment existieren, in dem ***nur-lesbare*** Pages definiert sind. Ein solches Segment hat das effektive Schutz-Attribut „Read/Write".



Wenn hier versucht wird, in das Segment zu schreiben, reagiert der Segment-Schutz ***nicht***. Statt dessen verhindert der nachgeschaltete Page-Schutz den Zugriff auf eine Page ②.

BEISPIEL 3: Es soll ein Read Only-Datensegment existieren, in dem ***beschreibbare*** Pages definiert sind.



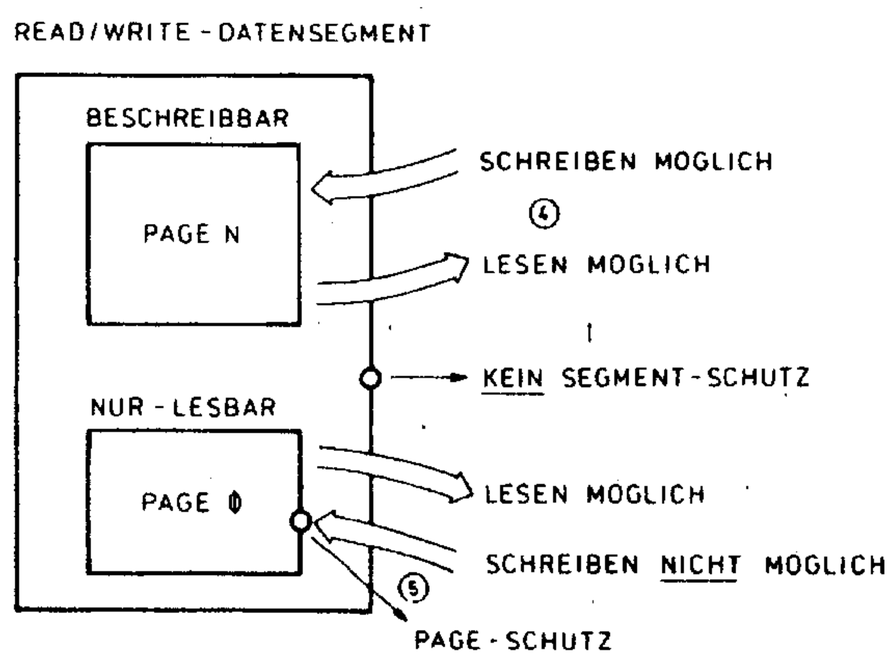
Da das ***effektive*** Schutz-Attribut dieses Segments „Read Only" ist, verhindert bereits der Segment-Schutz, daß in irgendeine Page geschrieben wird ③.

Um den Zugriff auf individuelle Pages flexibel gestalten zu können, muß ein page-orientiertes Betriebssystem garantieren, daß die Segment-Deskriptoren die Page-Schutz-Überprüfung erlauben. Mit anderen Worten, die Segment-Schutz-Überprüfung muß „ausgeschaltet" sein.

Wie die obigen Beispiele zeigen, ist dies nur möglich, wenn das effektive Schutz-Attribut eines Datensegments „Read/Write" ist.

Bei dieser Technik ist es nun möglich, daß individuell einige Untereinheiten des Datensegments ***nur-lesbar*** und andere Untereinheiten auch ***beschreibbar*** sind. So kann zum Beispiel in einem UNIX-ähnlichen System ein großes Datensegment mit einigen nur-lesbaren konstanten ROM-Daten definiert sein.

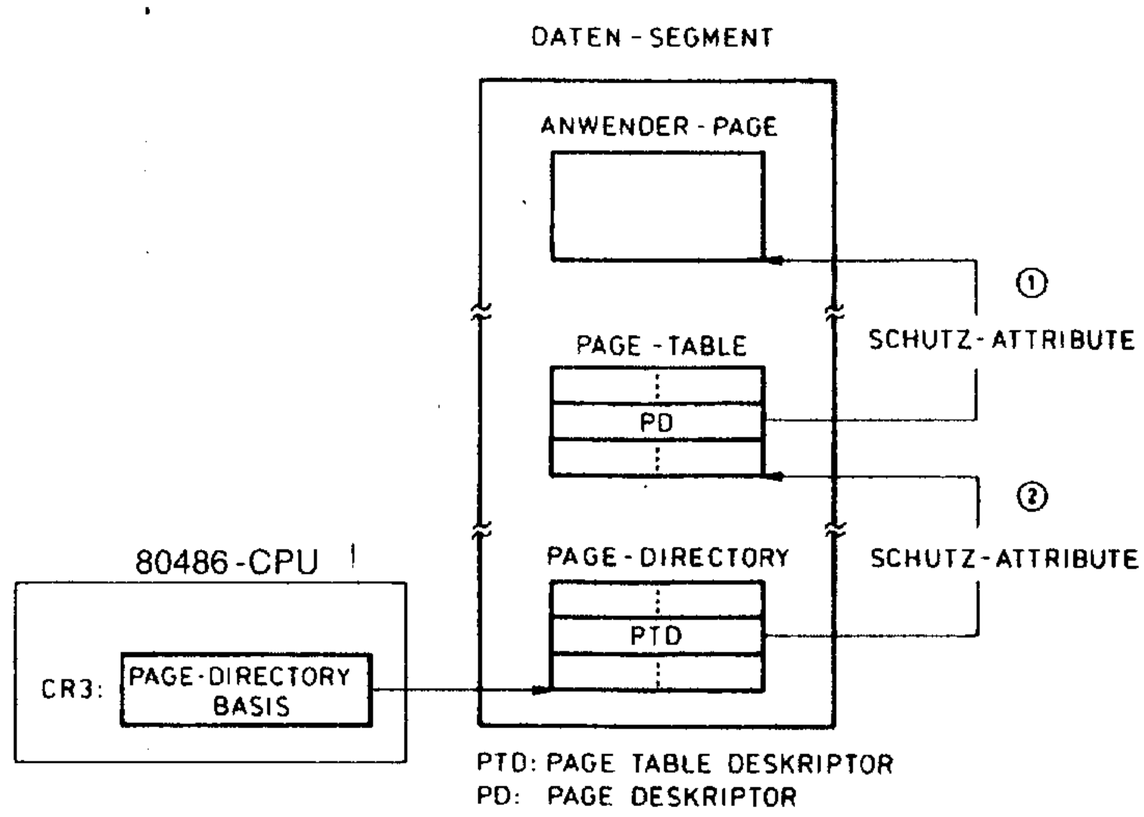
Das folgende Bild illustriert ein Read/Write-Datensegment, das Pages mit unterschiedlichen Schutz-Attributen enthält.



Während die beschreibbare Page N problemlos gelesen (naturgemäß) und beschrieben werden kann ④, verhindert der Page-Schutz Schreibzugriffe auf die nur-lesbare Page 0 ⑤.

Wenn umgekehrt in einer einzelnen Page mehrere Segmente enthalten sind, sollte die Page beschreibbar sein, so daß die Segment-Zugriffsrechte von den Segment-Attributen alleine geregelt werden können.

Wie die meisten Anwender-Pages halten sich auch die „Page Tables" und die Page-Directory-Tabelle als optionale Datenstrukturen in einem Datensegment auf. Dies verdeutlicht das folgende Bild:



Dabei werden die Schutz-Attribute, die eine individuelle Anwender-Page vor unberechtigten Zugriffen schützt, von ihrem Page-Deskriptor PD geliefert, der in der zugehörigen „Page Table" enthalten ist ①.

Damit eine individuelle „Page Table" in der gleichen Weise wie eine Anwender-Page geschützt werden kann, liefert der Page-Table-Deskriptor PTD in der „Page Directory" die gleichen Schutz-Attribute wie ein Page-Deskriptor ②.

Die Layouts der beiden Deskriptoren sind in den folgenden Bildern gezeigt:

'PAGE TABLE'-DESKRIPTOR:

31 … 12	11 10 9	8	7	6	5	4	3	2	1	0
PAGE TABLE-BASISADRESSE 31...12	AVAIL	0	0	D	A	PCD	PWT	U-S	R-W	P

'PAGE'-DESKRIPTOR:

31 … 12	11 10 9	8	7	6	5	4	3	2	1	0
ANWENDER-PAGE-BASISADRESSE 31...12	AVAIL	0	0	D	A	PCD	PWT	U-S	R-W	P

An dieser Stelle soll zunächst nur das Schutz-Attribut R/W (***R***ead/***W***rite) von Interesse sein.

- ***R/W = 0:*** eine Anwender-Page oder eine „Page Table" kann ***nur gelesen*** werden.

- ***R/W = 1:*** eine Anwender-Page oder eine „Page Table" kann sowohl ***gelesen*** als auch ***beschrieben*** werden.

Der flexibelste Schutz für „Page Tables" ist gegeben, wenn die Segment-Schutz-Attribute ausgeschaltet sind. Dies ist der Fall, wenn sie genauso wie die Anwender-Pages in einem Read/Write-Datensegment aufbewahrt sind.

Die Page-Directory-Tabelle muß als Wurzelverzeichnis jederzeit aktualisiert, d. h. beschrieben werden können. Daher ist für sie ein R/W-Attribut überflüssig. Dies zeigt auch das Root-Register CR3, das lediglich die physikalische Basis-Adresse der „Page Directory" enthält.

Damit aber der wahlfreie Zugriff auf die „Page Directory" gewährleistet ist, muß sie in einem Read/Write-Datensegment enthalten sein. Wenn die Page-Directory-Tabelle vor Zugriffen von Anwender-Prozeduren in der Privileg-Ebene 3 (CPL = 3) geschützt werden soll, dann muß sich dieses Datensegment in den Betriebssystem-Privileg-Ebenen 0, 1 oder 2 befinden.

3.19.1 Page-orientierter virtueller Speicher

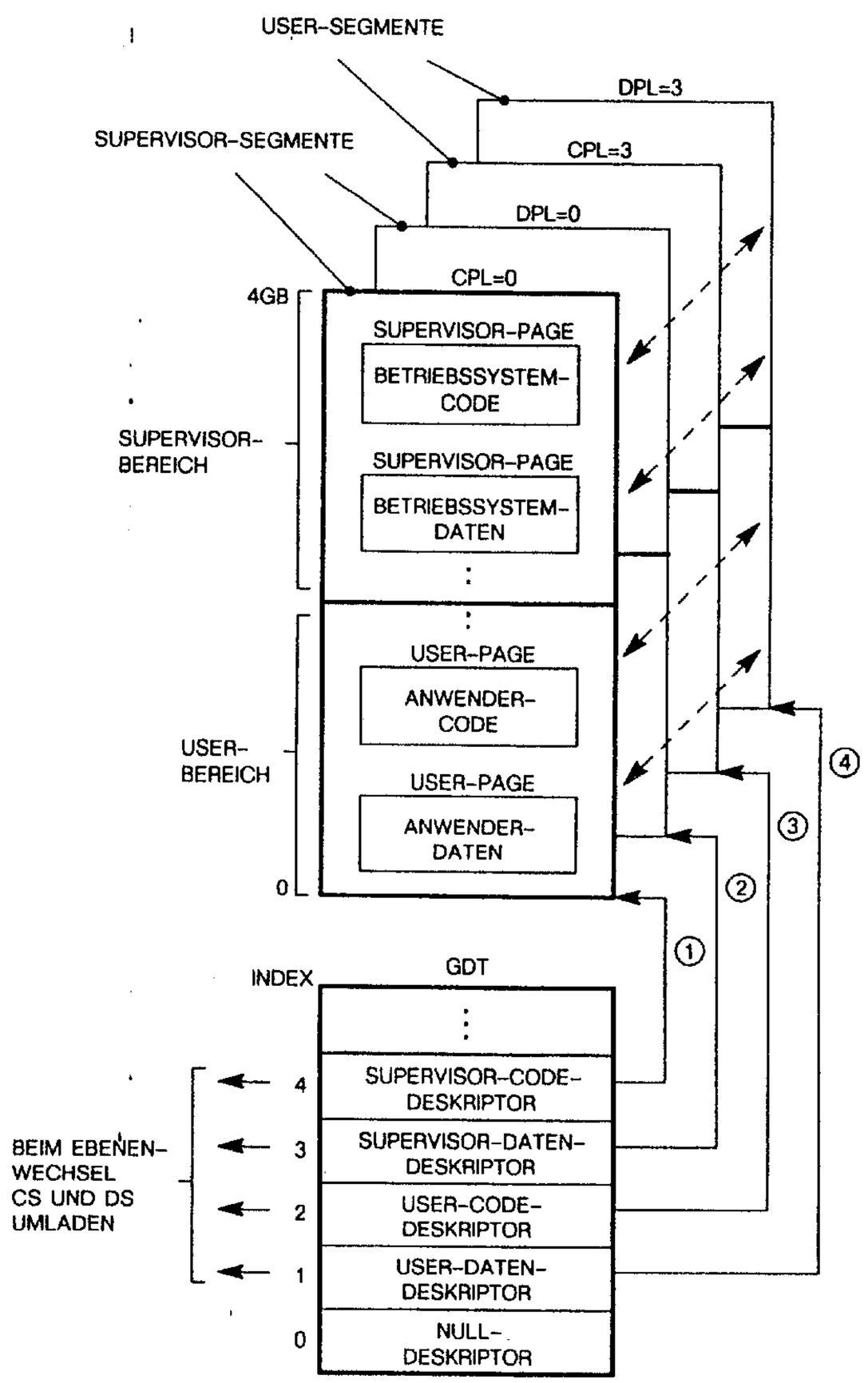
Ein 80486-Betriebssystem kann einen virtuellen Speicher implementieren, wenn es zwischen dem sekundären Massenspeicher und dem physikalischen Arbeitsspeicher entweder Segmente oder Pages transferiert. Beide Methoden haben ihre Berechtigung. Das Ein- bzw. Auslagern von Segmenten oder Pages erledigt der sogenannte „Swapper", der in der Privileg-Ebene 0 (CPL = 0) laufen sollte. Der Swapper führt folgende reziproken Funktionen aus:

- ***swap-in***
  Segment- oder Page-Transfer vom sekundären Massenspeicher zum physikalischen Arbeitsspeicher
- ***swap-out***
  Segment- oder Page-Transfer vom physikalischen Arbeitsspeicher zum sekundären Massenspeicher

In einem segment-orientierten System kann die swap-in-Zeit beträchtlich wachsen, wenn der Swapper für sehr große Segmente (im Extremfall 4 Gigabytes) freie lineare Speicherblöcke finden muß. Für diesen Fall wird man ein page-orientiertes System wählen, da hier jeder freie Page-Raum tauglich ist, um darin in kürzester Zeit eine Page einzulagern.

Wenn in einem page-orientierten virtuellen Speicherraum der „flache" Adreßraum benützt wird, sind in diesem maximal 4 Gigabyte großen Segment sowohl Betriebssystem-Code und -Daten als auch Anwender-Code und -Daten untergebracht. Es liegt auf der Hand, daß bei einer solchen Konfiguration neue Schutz-Attribute erforderlich sind, um die Betriebssystem-Region vor der Anwender-Region zu schützen.

Das folgende Bild illustriert ein hypothetisches Betriebssystem, das Paging sowohl für den Schutz als auch für den virtuellen Speicher benützt.



Das Bild zeigt, daß das Code-Segment der Anwender-Task in der Privileg-Ebene 3 den gleichen page-granulierten Adreßraum umfaßt wie das Code-Segment des Betriebssystems in der Privilegebene 0. Analoges gilt für die beiden Daten-Segmente. Auch hier umfaßt das Daten-Segment der Anwender-Task in der Privileg-Ebene 3 den gleichen linearen Adreßraum wie das Daten-Segment des Betriebssystems in der Privileg-Ebene 0.

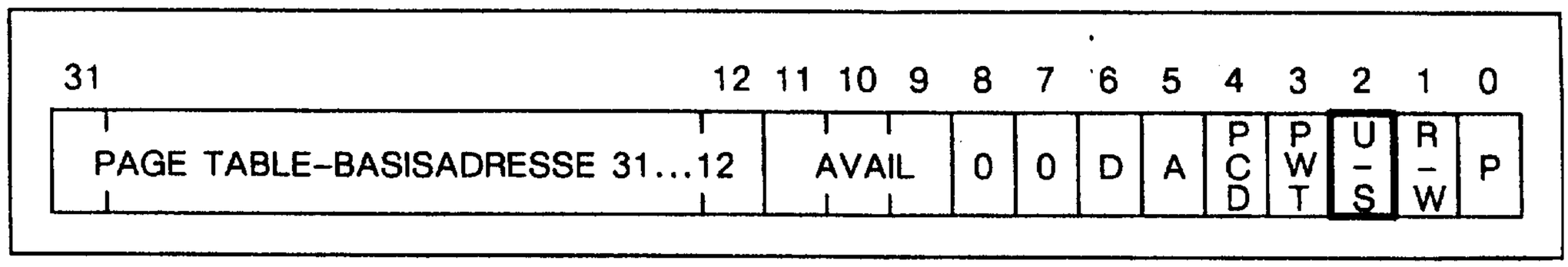
Die in der GDT enthaltenen Deskriptoren legen für diese Segmente zwar unterschiedliche Zugriffsrechte fest; sie haben aber alle die Basisadresse 0 und ihre Obergrenzen liegen bei 4 Gigabytes (Base = 0, Limit = 0FFFFFH, Granularity = 1).

- Der **Deskriptor** bei **Index 4** beschreibt das Segment als Code-Segment in der Privileg-Ebene 0 (CPL = 0) ①.
- Der **Deskriptor** bei **Index 3** beschreibt das Segment als Daten-Segment in der Privileg-Ebene 0 (DPL = 0) ②.
- Der **Deskriptor** bei **Index 2** beschreibt das Segment als Code-Segment in der Privileg-Ebene 3 (CPL = 3) ③.
- Der **Deskriptor** bei **Index 1** beschreibt das Segment als Daten-Segment ④.

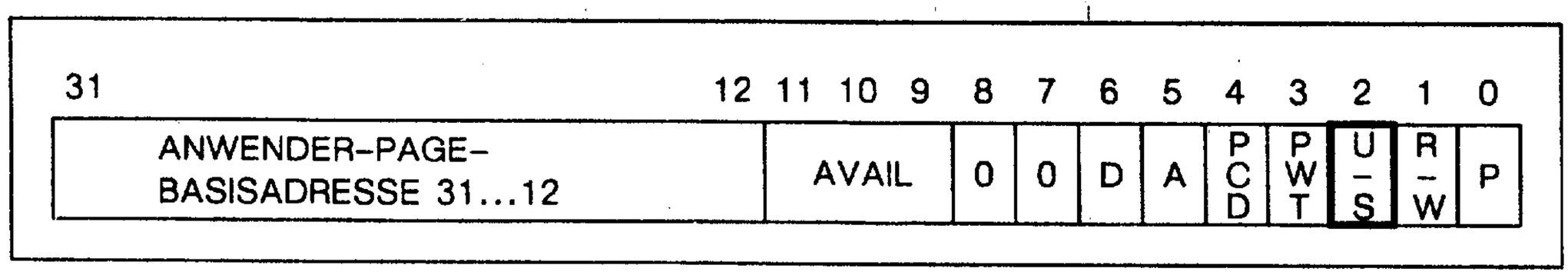
Da alle Segmente auf der linearen Basisadresse 0 basieren, können alle Eintragungen (Code oder Daten) durch einfache 32-Bit-Offsetzeiger erreicht werden.

Immer dann, wenn eine Task ein Code-Segment in der Privileg-Ebene 0, 1 oder 2 ausführt (CPL = 0, 1 oder 2), befindet sich der Prozessor in der sogenannten **Supervisor-Ebene**. Führt dagegen eine Task ein Code-Segment in der Privileg-Ebene 3 aus (CPL = 3), dann befindet sich der Prozessor in der sogenannten **User-Ebene**. Diese Ebenen können als Attribute auch für individuelle Pages und Page-Tables vergeben werden. Zu diesem Zweck steht sowohl im Page-Deskriptor als auch im Page-Table-Deskriptor das U/S-Bit (User/Supervisor) zur Verfügung. Ihre Positionen sind in den folgenden Deskriptor-Layouts zu sehen:

'PAGE TABLE'-DESKRIPTOR:



'PAGE'-DESKRIPTOR:



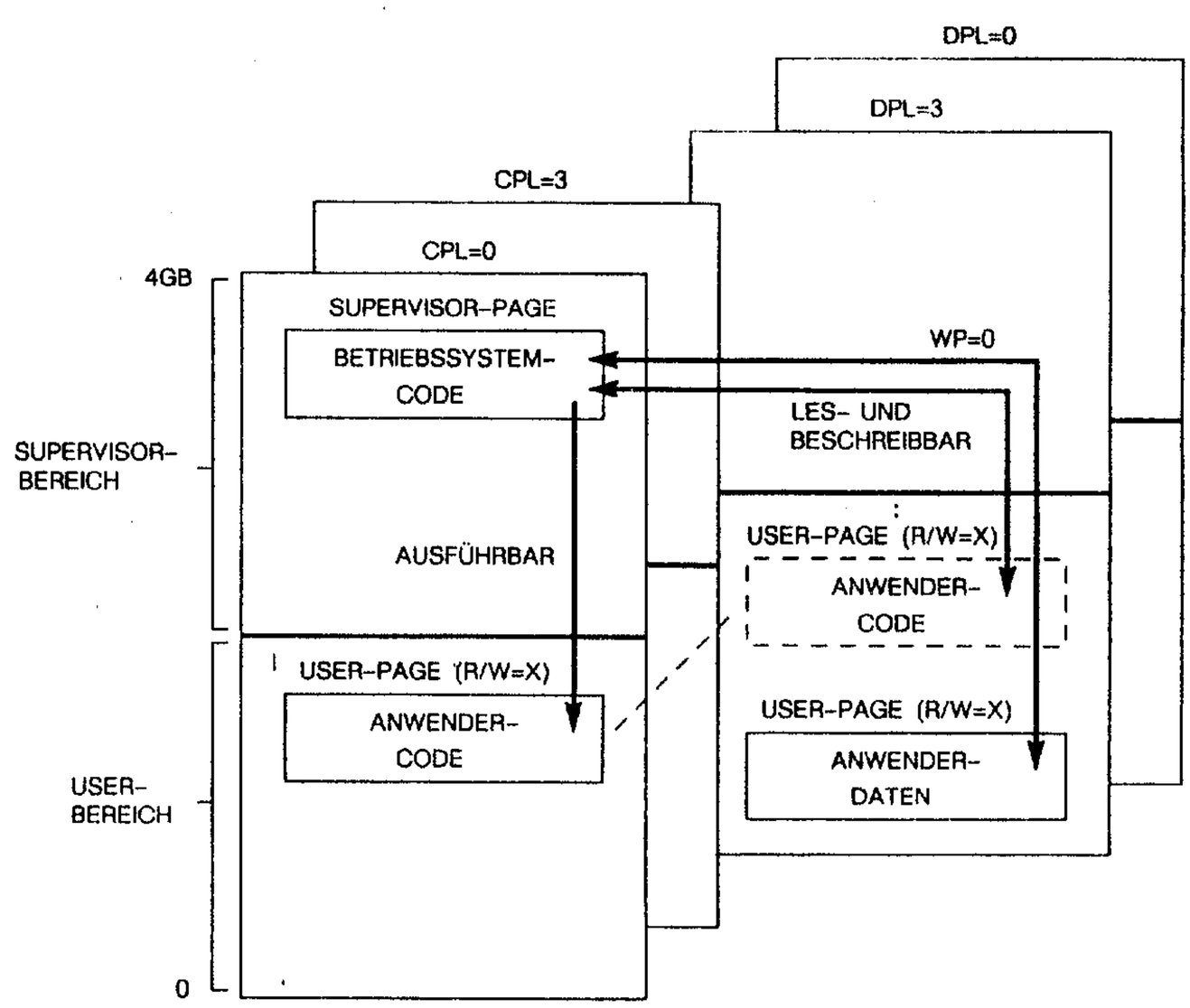
- **Supervisor-Ebene**
  Mit **U/S = 0** befindet sich die Page oder die Page-Table auf der Supervisor-Ebene. Diese Ebene wird vom Betriebssystem benutzt.
- **User-Ebene**
  Mit U/S = 1 befindet sich die Page oder Page-Table auf der User-Ebene. Diese Ebene wird von der Anwender-Software benutzt.

3.19.2 Supervisor-Zugriffsrechte

Wenn eine Betriebssystem-Task in den Privileg-Ebenen 0, 1 oder 2 ausgeführt wird (CPL = 0, 1, 2) dann benutzt sie den gleichen linearen Adreßraum wie die Anwender-Task. Dies bedeutet, daß sie sich die User-Pages „einverleibt" und damit das Zugriffsrecht auf die eigenen und **auch** auf die User-Pages hat.

Wie im folgenden Bild zu sehen ist, kann die Betriebssystem-Task unabhängig vom Page-Schutz-Attribut R/W User-Code **ausführen** und User-Daten sowohl **lesen** als auch **beschreiben**. Selbst „Read-Only-Pages" können von der Supervisor-Ebene aus **überschrieben** werden.

Die Betriebssystem-Task hat aber noch eine weitere Möglichkeit. Betrachten wir das User-Code-Segment. Ein solches Segment ist per Definition **nicht** beschreibbar. Nun sind aber diesem Segment zwei beschreibbare Daten-Segmente in der Privileg-Ebene 0 und 3 überlagert. Weil die Betriebssystem-Task in der Privileg-Ebene 0 alle Zugriffsrechte auf diese Daten-Segmente hat, kann sie außerdem noch dern User Code **überschreiben**. Auch in diesem Fall spielt das Page-Schutz-Attribut R/W keine Rolle.



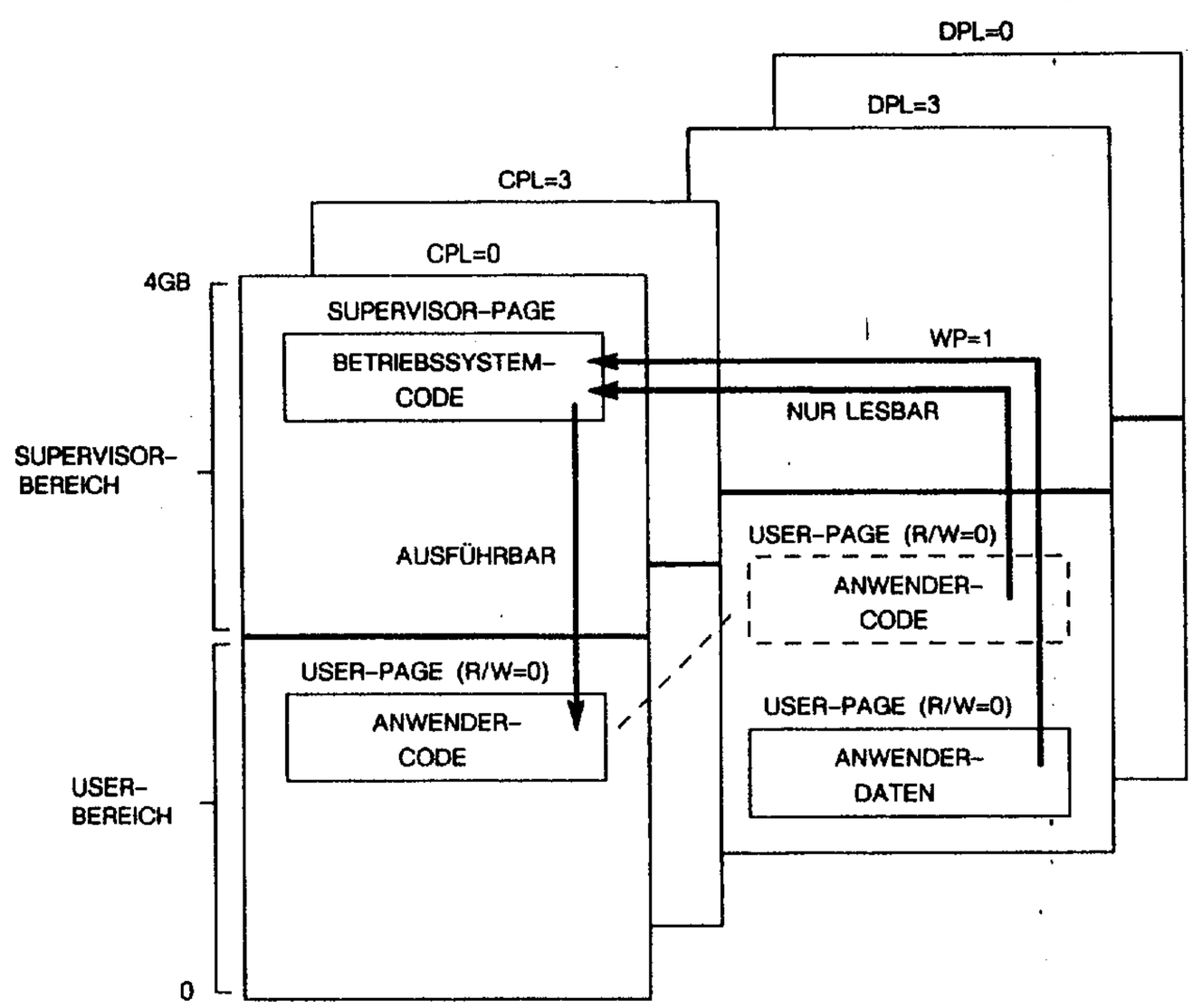
Die R/W- und U/S-Bits stehen in enger Verbindung mit dem WP-Bit (Write Protect) im CR0-Register. Die Position dieses Bits ist im folgenden Layout zu sehen:

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

MSW

- Wenn **WP = 0** ist, dann kann der Supervisor (CPL = 0) **alle** Pages **beschreiben** und zwar unabhängig vom R/W-Status. Die User- und Supervisor-Codes sind dabei uneingeschränkt ausführbar.
  Diese Situation ist im obigen Bild am Beispiel von User-Pages illustriert:
- Wenn **WP = 1** ist und **R/W = 0** (Read-Only-Page), dann sind alle Pages in der User- und Supervisor-Ebene **schreibgeschützt**. Mit anderen Worten, Read-Only-Pages können **nicht** mehr überschrieben werden. Die User- und Supervisor-Codes sind aber noch uneingeschränkt ausführbar.

Das folgende Bild zeigt die Wirkung von WP = 1 am Beispiel von User-Pages:



Was ist der Sinn von WP? Wenn **WP = 0** ist und der 80486 befindet sich in den höheren Stufen der Vertrauenshierarchie, dann stehen ihm alle möglichen Zugriffe auf die Pages offen. So kann sich die Software in diesen Privileg-Stufen absichtlich oder unabsichtlich modifizieren. Es gibt keinen Schreibschutz für Pages in diesen Ebenen.

Bei **WP = 1** sind alle Pages einer Task als schreibgeschützt markiert. Nehmen wir an, eine Task ist im Besitz von einigen Pages und es findet ein Task-Wechsel statt.

Um sicherzugehen, daß die Nachfolge-Task („Sohn") in ihrem eigenen Adreßraum arbeitet, werden **alle** Pages der Vorgänger-Task („Vater") in den linearen Adreßraum des „Sohnes" kopiert. Dabei kann der Kopieraufwand beträchtlich sein. Dem UNIX-Systemprogrammierer ist dieser Vorgang als **„forking"** bekannt.

Allerdings ist nicht einzusehen, warum alle Pages des „Vaters" in den Adreßraum des „Sohnes" kopiert werden müssen. Greift nämlich der „Sohn" auf einige Pages des „Vaters" nur **lesend** zu, dann ändert sich deren Inhalt nicht. Daher besteht für solche Pages kein zwingender Grund, sie in den Adreßraum des „Sohnes" zu kopieren. Eine andere Situation ergibt sich, wenn der „Sohn" in eine Page des „Vaters" schreiben möchte. Nur in diesem Fall muß sich die betreffende Page im Adreßraum des „Sohnes" aufhalten. Also lautet die Strategie: Kopiere nur diejenigen Pages des „Vaters" in den Adreßraum des „Sohnes", auf die der „Sohn" schreibend zugreift.

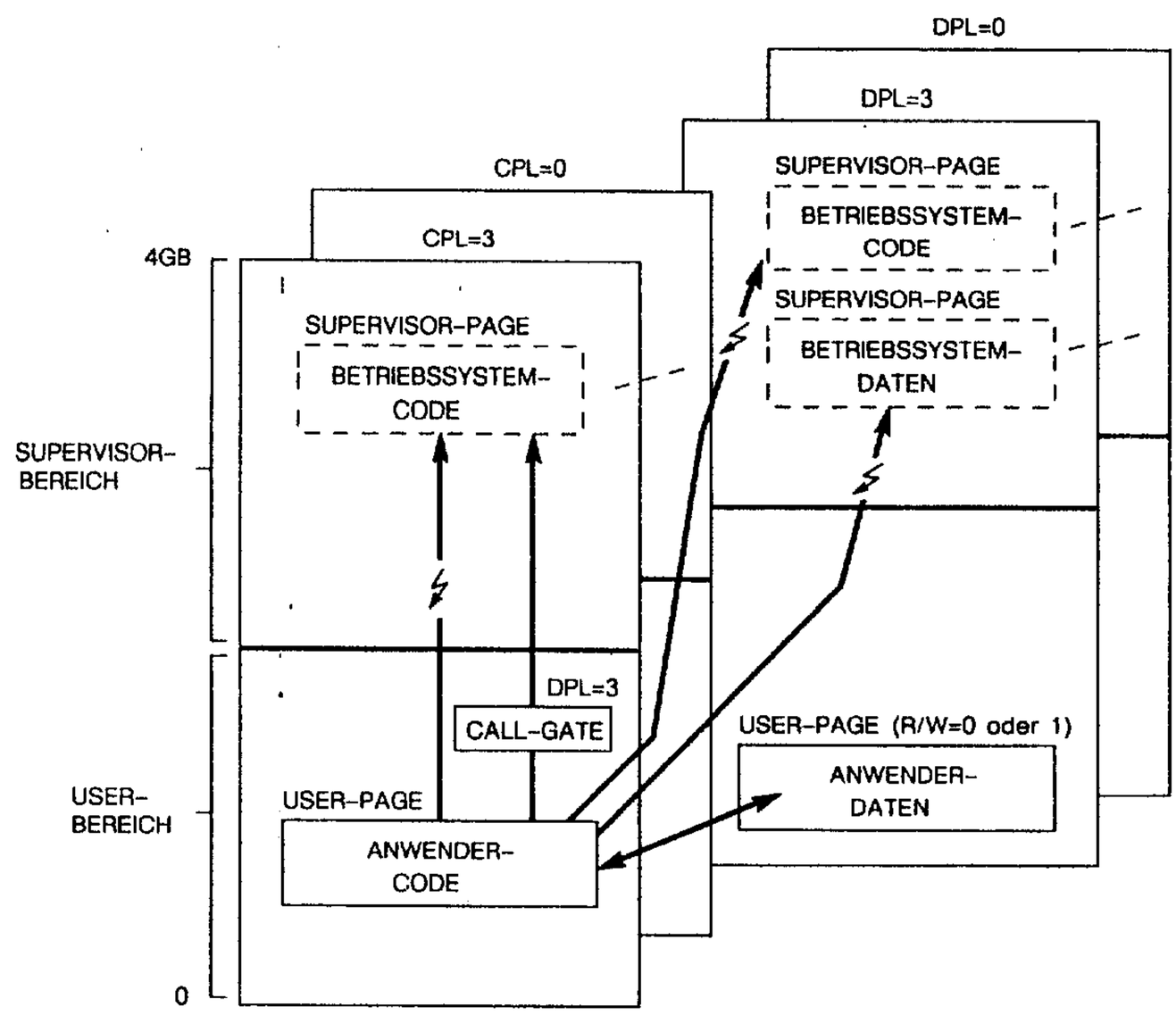
Was ist also zu tun? Wenn der „Sohn" versucht, in eine schreibgeschützte Page (WP = 1) zu schreiben, dann erzeugt die 80486-Hardware eine Ausnahmebedingung (Interrupt 14). Das Betriebssystem wird dadurch von diesem Schreibversuch unterrichtet und kopiert daraufhin die betreffende Page in den Adreßraum des „Sohnes".

Nutzt man diese Möglichkeit, dann werden nach einem Taskwechsel nicht alle Pages des „Vaters" kopiert, sondern nur diejenigen, auf die der „Sohn" **schreibend** zugreift. Aus die-

sem Grund wird diese Methode auch als **„Copy-On-Write“** oder **„Lazy-Copying“** bezeichnet. Der Vorteil dabei ist, daß sich der Kopieraufwand wesentlich verringert.

3.19.3 User-Zugriffsrechte

Wenn eine Anwender-Task in der Privileg-Ebene 3 (CPL = 3) ausgeführt wird, dann benutzt sie den gleichen linearen Adreßraum wie die Betriebssystem-Task. Obwohl sie sich damit die Supervisor-Pages „einverleibt“, hat sie trotzdem **nur** Zugriffsrechte auf diejenigen Pages, die sich in der User-Ebene aufhalten. Dabei werden in dieser Ebene die Zugriffsrechte ausschließlich vom R/W-Bit geregelt. Das WP-Bit hat hier **keine** Wirkung. Im folgenden Bild ist dies illustriert:



Wie zu sehen ist, kann der Anwender-Code die Anwender-Daten abhängig vom R/W-Status sowohl lesen als auch beschreiben. Wenn aber die Anwender-Task versucht, die Pages in der Supervisor-Ebene zu adressieren, stellt dies eine Page-Schutz-Verletzung dar, die zur Unterbrechung des Programms führt (Interrupt 14). Mit anderen Worten, die Anwender-Task kann **keinen** Betriebssystem-Code ausführen. Dies ist nur **indirekt** über einen Call-Gate-Deskriptor möglich. Hier initialisiert dann der Prozessor das CS-Register mit dem Betriebssystem-Code-Deskriptor und setzt anschließend das Programm in der Privileg-Ebene 0, 1 oder 2 fort.

Obwohl dem Anwender-Code-Segment (CPL = 3) ein beschreibbares Anwender-Daten-Segment (DPL = 3) überlagert ist, sind die darin enthaltenen Supervisor-Pages mit Betriebssystem-Code und -Daten ebenfalls weder lesbar noch beschreibbar. Auch hier führt der Versuch eines Zugriffs auf diese Pages zur Unterbrechung des Programms (Interrupt 14).

Zusammenfassung: Die folgende Tabelle zeigt alle Kombinationen der U/S-, R/W- und WP-Bits und die daraus resultierenden User- und Supervisor-Zugriffsrechte.

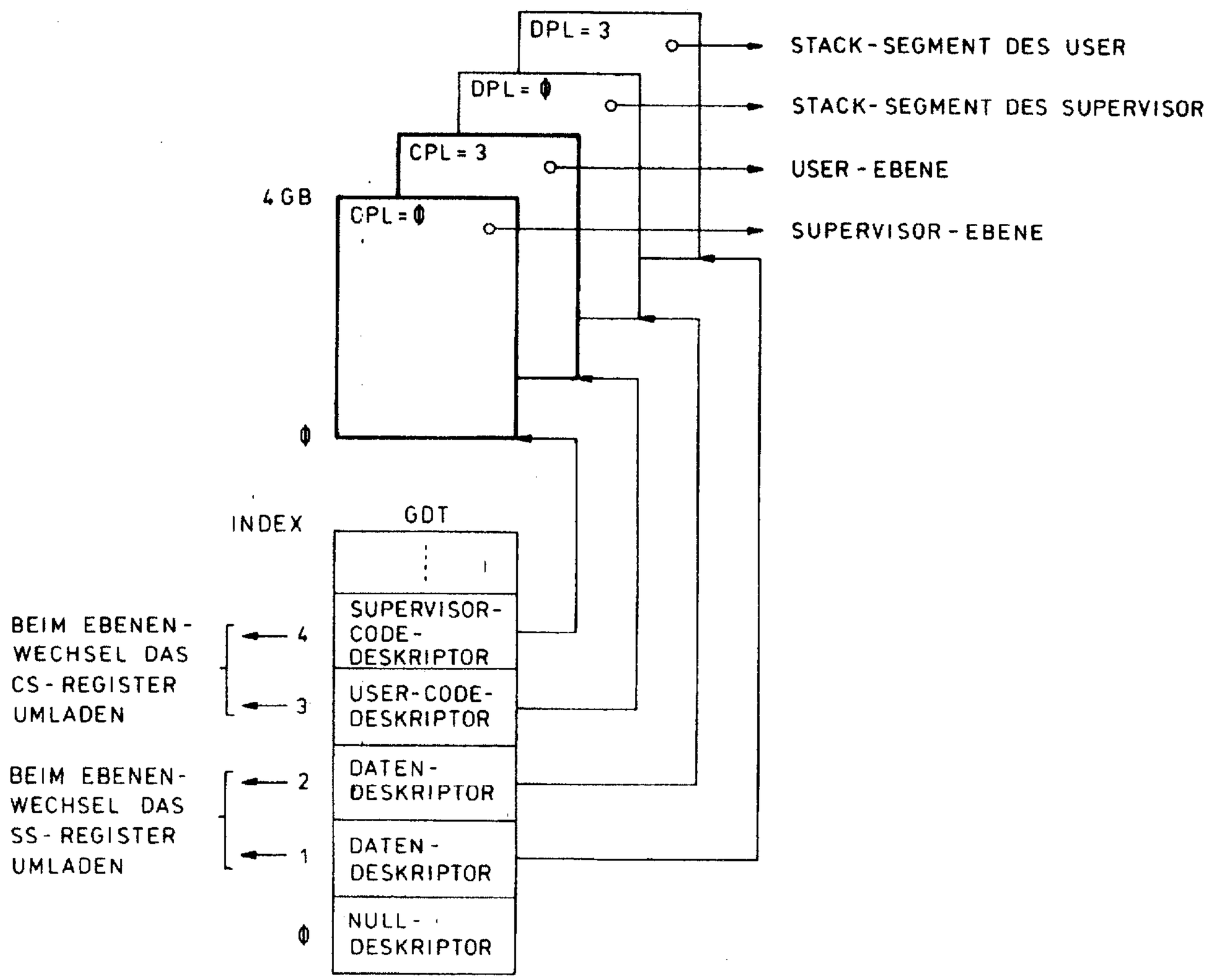
U/S	R/W	R/W	User-Zugriffsrechte	Supervisor-Zugriffsrechte
0	0	0	keine	lesen/schreiben ausführen
0	1	0	keine	lesen/schreiben ausführen
1	0	0	lesen ausführen	lesen/schreiben ausführen
1	1	0	lesen/schreiben ausführen	lesen/schreiben ausführen
0	0	1	keine	lesen ausführen
0	1	1	keine	lesen/schreiben ausführen
1	0	1	lesen ausführen	lesen ausführen
1	1	1	lesen/schreiben ausführen	lesen/schreiben ausführen

U/S = 0; Supervisor
U/S = 1; User
R/W = 0; lesbar
R/W = 1; beschreibbar
WP = 0; **kein** Schreibschutz
WP = 1; Schreibschutz

3.19.4 Zugriff auf Stack-Daten

Ein Code-Segment kann auf Stack-Daten nur dann zugreifen, wenn sich das Stack-Segment auf der ***gleichen*** Privileg-Ebene wie das Code-Segment befindet (DPL = CPL). Dies bedeutet, daß z. B. ein Supervisor-Code-Segment in der Privileg-Ebene 0 (CPL = 0) Stack-Operationen nur dann ausführen kann, wenn das SS-Register einen Daten-Segment-Deskriptor mit dem Wert DPL = 0 enthält.

Im Gegensatz dazu kann ein User-Code-Segment in der Privileg-Ebene 3 (CPL = 3) Stack-Operationen nur dann ausführen, wenn das SS-Register einen Daten-Segment-Deskriptor mit dem Wert DPL = 3 enthält. Das SS-Register muß daher umgeladen werden, wenn der Prozessor entweder die Supervisor-Ebene oder die User-Ebene verläßt. Dies verdeutlicht das folgende Bild:

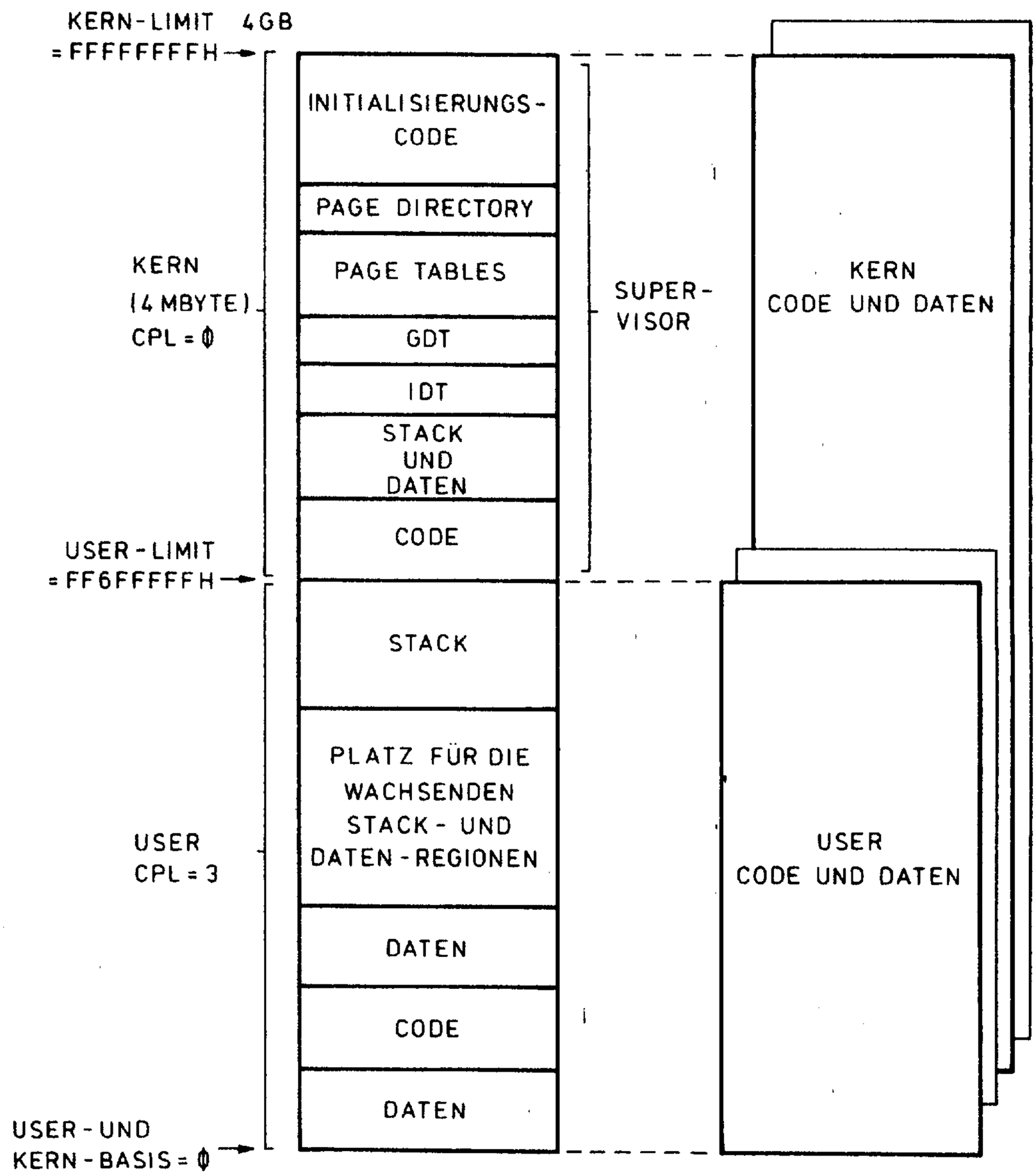


3.19.5 Redundanter Segment- und Page-Schutz

Das folgende Bild illustriert den linearen Adreßraum einer hypothetischen Betriebssystem-Implementation für den 80486.

Es ist zu erkennen, daß die typischen Kern-Daten-Strukturen des Betriebssystems die oberen 4 MBytes des 4 Gigabyte linearen Adreßraums besetzen. Wenn sich der Prozessor in der Privileg-Ebene 0 (CPL = 0) aufhält, kann der Kern alle privilegierten Befehle ausführen.

Der User belegt den Adreßbereich 0 ... FF6FFFFFH und hält sich in der Privileg-Ebene 3 auf.



Die angegebene Struktur wird über Segment-Deskriptoren arrangiert, deren Attribute in den folgenden beiden Tabellen zusammengefaßt sind:

Kern-Segment-Deskriptoren:

Attribute	**Code**	**Daten**
Base	0H	0H
Limit	0FFFFFH	0FFFFFH
Granularity Big Present	1B 1B 1B	1B 1B 1B
Privileg-Ebene	00B	00B
Executable Conforming/Expand Down Read/Write	1B 0B 1B	0B 0B 1B

User-Segment-Deskriptoren:

Attribute	Code	Daten
Base	0H	0H
Limit	0FF6FFH	0FF6FFH
Granularity Big Present	1B 1B 1B	1B 1B 1B
Privileg-Ebene	11B	11B
Executable Conforming/Expand Down Read/Write	1B 0B 1B	0B 0B 1B

B: Binär

Es ist zu erkennen, daß der Kern und der User als Code- und Datensegmente beschrieben werden. Dabei sind die Code-Deskriptoren des Kern- und des User-Segments nahezu identisch. Das gleiche gilt für die Daten-Deskriptoren der beiden Segmente. Sie unterscheiden sich lediglich in der ***Privileg-Ebene*** und im ***Limit-Wert***.

Die Kern-Daten-Strukturen des Betriebssystems in der Privileg-Ebene 0 (oberer 4 MByte-Bereich) sind durch Supervisor-Pages, die Daten-Strukturen des Users in der Privileg-Ebene 3 durch User-Pages verfeinert. Bei **WP = 0** hat der Kern uneingeschränktes Read/Write-Zugriffsrecht auf **alle** Pages. Das ist deswegen so, weil die Bedingung

- Ziel-Daten(Code)-Segment DPL > = MAX (Task CPL, Selektor RPL)

erfüllt ist und weil

- wegen WP = 0 der Status von R/W **ohne** Wirkung ist.

Umgekehrt sind die Kern-Daten-Strukturen des Betriebssystems vor Zugriffen des Users **zweifach** gesichert:

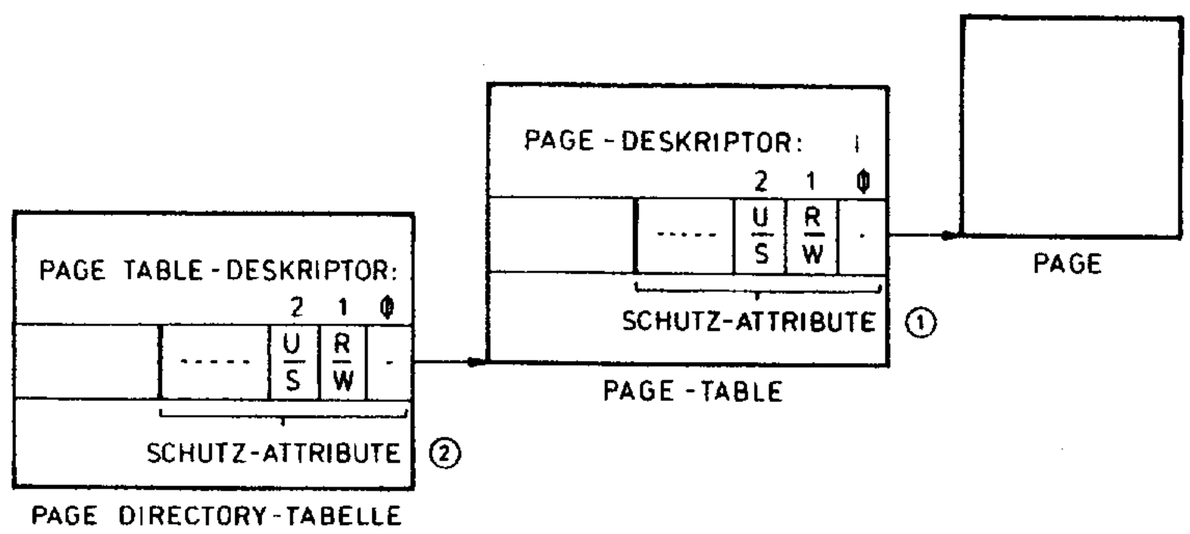
1. Code-Segmente in Privileg-Ebene 3 können ***nicht*** auf Daten in der Privileg-Ebene 0 zugreifen (Segment-Schutz).
2. User-Codes in der Privileg-Ebene 3 können ***nicht*** auf „Supervisor Pages" in den Privileg-Ebenen 0, 1 oder 2 zugreifen (Page-Schutz).

Weil in der angegebenen Struktur der Speicher des Kerns bereits über einen Segment-Schutz verfügt, wirkt der Page-Schutz als ***redundanter*** Schutz.

3.19.6 Kombination von „Page"- und „Page-Table"-Schutz-Attributen

Für Pages können sich die Schutz-Attribute U/S und R/W ihres Page-Deskriptors ① von denen ihres Page-Table-Deskriptors ② unterscheiden.

Der Prozessor prüft die Schutz-Attribute sowohl in der Page-Directory-Tabelle als auch in der „Page Table" und ermittelt daraus das sogenannte ***effektive*** Schutz-Attribut für die Page.



Dabei liefert der Prozessor immer das **numerisch kleinere** der beiden Attribute und gewährt damit der Page den größtmöglichen Schutz.

Beispiel: Nehmen wir an, der

- **Page-Table**-Deskriptor hat das Schutz-Attribut **10**, also U/S = 1 (User) und R/W = 0 (lesbar/ausführbar) und der
- **Page**-Deskriptor hat das Schutz-Attribut **01**, also U/S = 0 (Supervisor) und R/W = 1 (beschreibbar/ausführbar).

Hier bekommt die Page das effektive Schutz-Attribut **01**, also U/S = 0 (Supervisor) und R/W = 1 (beschreibbar/ausführbar).

Die folgende Tabelle zeigt, welcher Schutz bei **WP = 0** für alle möglichen Kombinationen von Page-Schutz-Attributen geliefert wird.

Page-Directory-Tabelle (PTD)		Page-Table (PD)		Effektives Schutzattribut der Page	
Ebene	Zugriffs-Typ	Ebene	Zugriffs-Typ	Ebene	Zugriffs-Typ
User	nur lesen	User	nur lesen	User	nur lesen
User	nur lesen	User	lesen und schreiben	User	nur lesen
User	lesen und schreiben	User	nur lesen	User	nur lesen
User	lesen und schreiben	User	lesen und schreiben	User	lesen und schreiben
User	nur lesen	Supervisor	nur lesen	User	nur lesen
User	nur lesen	Supervisor	lesen und schreiben	User	nur lesen
User	lesen und schreiben	Supervisor	nur lesen	User	nur lesen
User	lesen und schreiben	Supervisor	lesen und schreiben	User	lesen und schreiben
Supervisor	nur lesen	User	nur lesen	User	nur lesen
Supervisor	nur lesen	User	lesen und schreiben	User	nur lesen
Supervisor	lesen und schreiben	User	nur lesen	User	nur lesen
Supervisor	lesen und schreiben	User	lesen und schreiben	User	lesen und schreiben
Supervisor	nur lesen	Supervisor	nur lesen	Supervisor	lesen und schreiben
Supervisor	nur lesen	Supervisor	lesen und schreiben	Supervisor	lesen und schreiben
Supervisor	lesen und schreiben	Supervisor	nur lesen	Supervisor	lesen und schreiben
Supervisor	lesen und schreiben	Supervisor	lesen und schreiben	Supervisor	lesen und schreiben

PTD: Page-Table-Deskriptor
PD: Page-Deskriptor
User: U/S = 1

Supervisor: U/S = 0
nur lesen: R/W = 0
lesen und schreiben: R/W = 1

3.19.7 Behandlung von Page-Fehlern

Immer dann, wenn der 80486 eine lineare Adresse in eine physikalische Adresse übersetzt, bestehen folgende Gefahren:

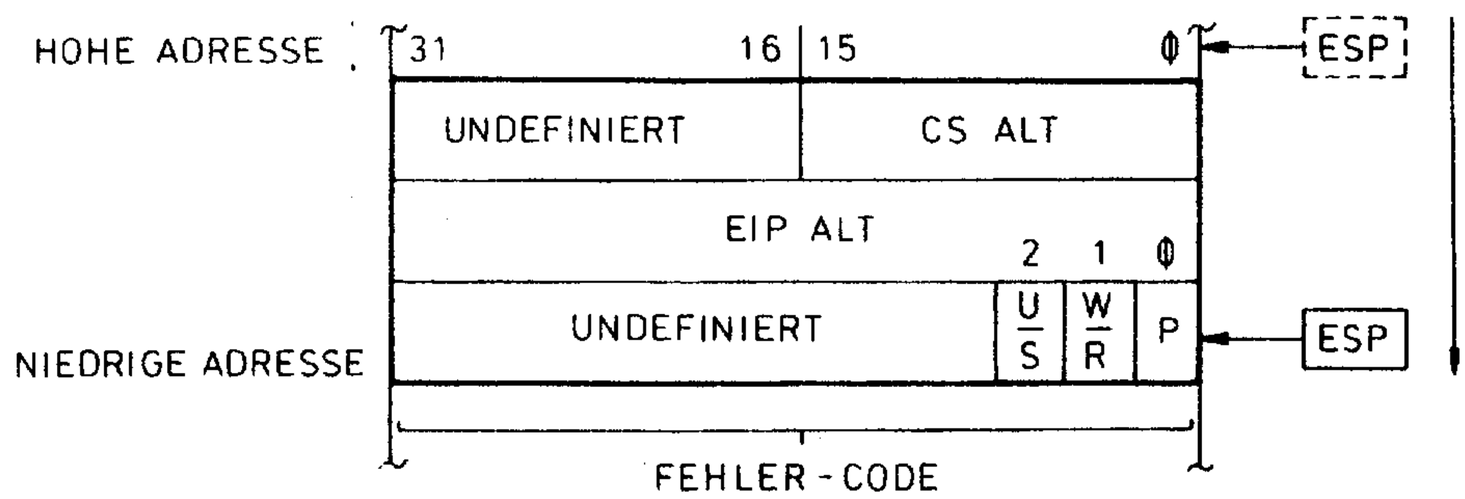
- daß eine Page-Schutz-Verletzung auftritt oder
- daß der 80486 auf eine ***nicht-vorhandene*** „Page Table" trifft oder
- daß der 80486 auf eine ***nicht-vorhandene*** Anwender-Page trifft.

Alle diese Fälle repräsentieren sogenannte Page-Fehler („Page Faults"), die eine Unterbrechung des Programms herbeiführen. Der Betriebssystem-Code, der auf einen solchen Page-Fehler reagiert, wird als „Page Fault Handler" bezeichnet.

Dieser Handler ist normalerweise als privilegierte Prozedur in der Ebene 0 implementiert und hält sich im Adreßraum derjenigen Task auf, die den Page-Fehler verursacht hat. Weil der „Page Fault Handler" im fehlerverursachenden Context der Task läuft, hat er bereits Zugriff auf die „Page Directory" und die „Page Table" der Task. Das uneingeschränkte Zugriffsrecht ist allerdings nur dann gegeben, wenn er auch Les- und Schreibzugriff auf die Segmente hat, in denen die „Page Directory", die „Page Tables" und die Anwender-Pages enthalten sind.

Ist der „Page Fault Handler" aufgerufen worden, wird er zunächst den Typ des Fehlers feststellen. Dabei assistiert ihm der 80486, der zu diesem Zweck einige Diagnose-Daten liefert.

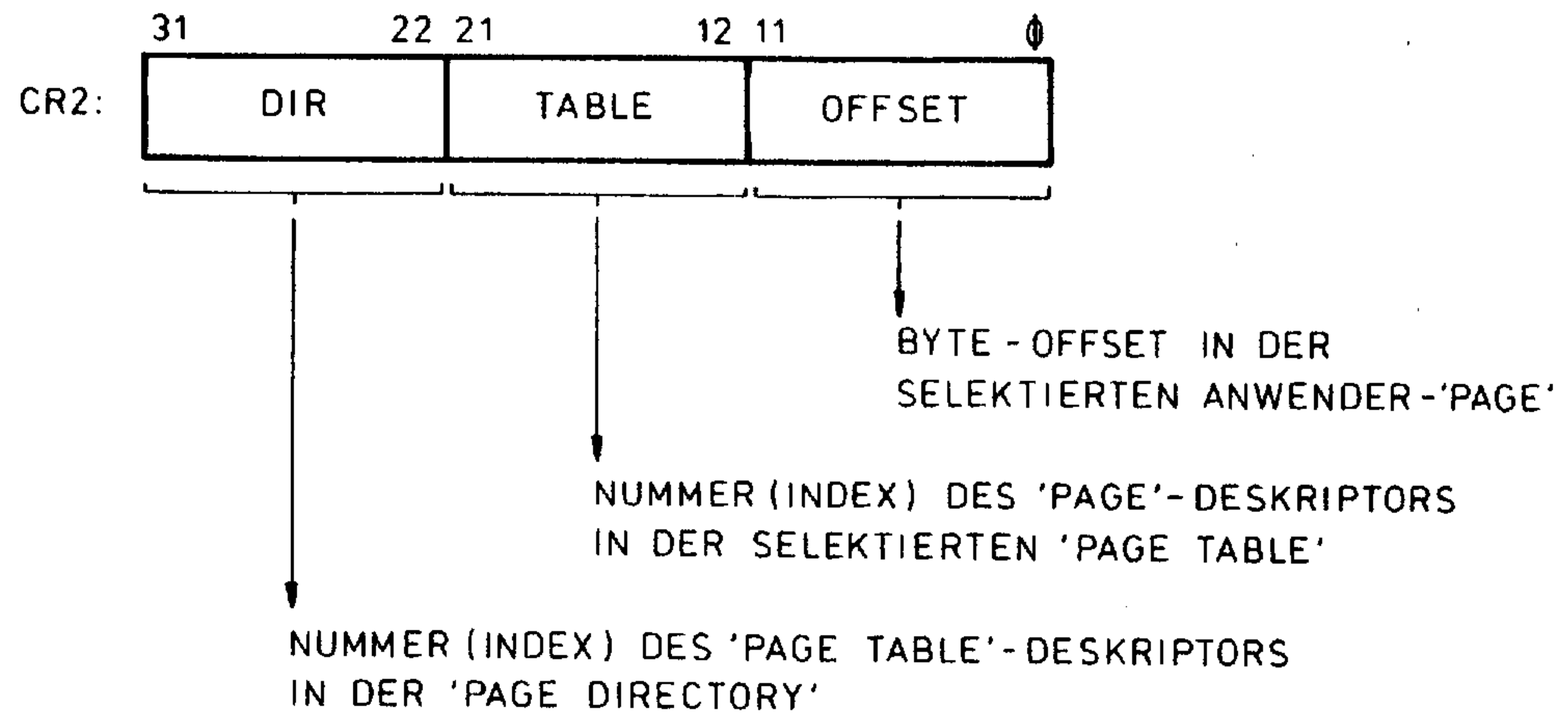
Diese Daten hinterlegt der 80486 im Stack des „Page Fault Handlers". Das Layout dieses Stack zeigt das folgende Bild:



Es ist ersichtlich, daß ab der augenblicklichen ESP-Position (Top of Stack) die ***logische*** Adresse CS:EIP des Befehls enthalten ist, der den Fehler verursacht hat. Danach folgt ein ***Fehler-Code,*** der den Typ des Fehlers beschreibt. Dabei haben die Felder in diesem Code folgende Bedeutung:

U/S:	**Prozessor-Modus** 0 = Der Fehler hat sich bei einem Zugriff ereignet, als der Prozessor im Supervisor-Modus aktiv war. 1 = Der Fehler hat sich bei einem Zugriff ereignet, als der Prozessor im User-Modus aktiv war.
W/R:	**Zugriffs-Typ** 0 = Der Fehler hat sich bei einem Lesezugriff ereignet. 1 = Der Fehler hat sich bei einem Schreibzugriff ereignet.
P:	**Page-Fehler-Interpretation** 0 = Der Fehler wurde durch eine nicht-vorhandene „Page Table" oder Anwender-Page verursacht. 1 = Der Fehler wurde durch eine Page-Schutz-Verletzung verursacht.

Damit der „Page Fault Handler" die am Page-Fehler beteiligten Page-Table- und Page Deskriptoren lokalisieren kann, stellt der 80486 das Spezial-Register ***CR2*** zur Verfügung. In diesem Register speichert der Prozessor die ***lineare*** Adresse, die beim fehlerverursachenden Zugriff benutzt wurde. Das Format des CR2-Registers zeigt das folgende Bild:



Der Zugriff auf das CR2-Register erfolgt durch die MOV-Speziel Befehle

- **MOV r32, CR2** ;Lade das „Machine Control Register" CR2
  ;in ein allgemeines 32-Bit-Register r32

und

- **MOV CR2, r32** ;Lade ein allgemeines 32-Bit-Register r32
  ;in das „Machine Control Register" CR2

Hat der „Page Fault Handler" im Fehler-Code zum Beispiel festgestellt, daß der Fehler durch einen Zugriff auf eine nicht-vorhandene „Page Table" oder Anwender-Page verursacht worden ist (P = 0), wird er die obigen MOV-Spezial-Befehle benützen, um die beteiligten „Page Tables"- und Page-Deskriptoren zu ermitteln. Jeder dieser Deskriptoren enthält das sogenannte ***Present***-Bit P. Der „Page Fault Handler" wird diese Bits prüfen, um festzustellen, ob den Page-Fehler eine nicht-vorhandene „Page Table" oder eine nicht-vorhandene Anwender-Page verursacht hat.

Die Position der Present-Bits ist in den Layouts der beiden Deskriptoren zu sehen:

'PAGE TABLE'-DESKRIPTOR:

31 … 12	11 10 9	8	7	6	5	4	3	2	1	0
PAGE TABLE-BASISADRESSE 31...12	AVAIL	0	0	D	A	PCD	PWT	U-S	R-W	P

'PAGE'-DESKRIPTOR:

31 … 12	11 10 9	8	7	6	5	4	3	2	1	0
ANWENDER-PAGE-BASISADRESSE 31...12	AVAIL	0	0	D	A	PCD	PWT	U-S	R-W	P

Das Present-Bit P zeigt an, ob eine „Page Directory" oder eine „Page Table" für die Adreß-Übersetzung benützt werden kann.

- ***P = 1 im Page-Table-Deskriptor*** zeigt an, daß die zugehörige „Page Table" ***vorhanden*** ist. Bei ***P = 0*** ist die zugehörige „Page Table" als ***nicht-vorhanden*** markiert.
- ***P = 1 im Page-Deskriptor*** zeigt an, daß die zugehörige Anwender-Page ***vorhanden*** ist. Bei ***P = 0*** ist die zugehörige Anwender-Page als ***nicht-vorhanden*** markiert.

Immer dann, wenn der 80486 eine lineare Adresse in eine physikalische Adresse übersetzt, prüft er der Reihe nach

- die Segment-Attribute,
- die Page-Table-Deskriptor-Attribute,
- die Page-Deskriptor-Attribute.

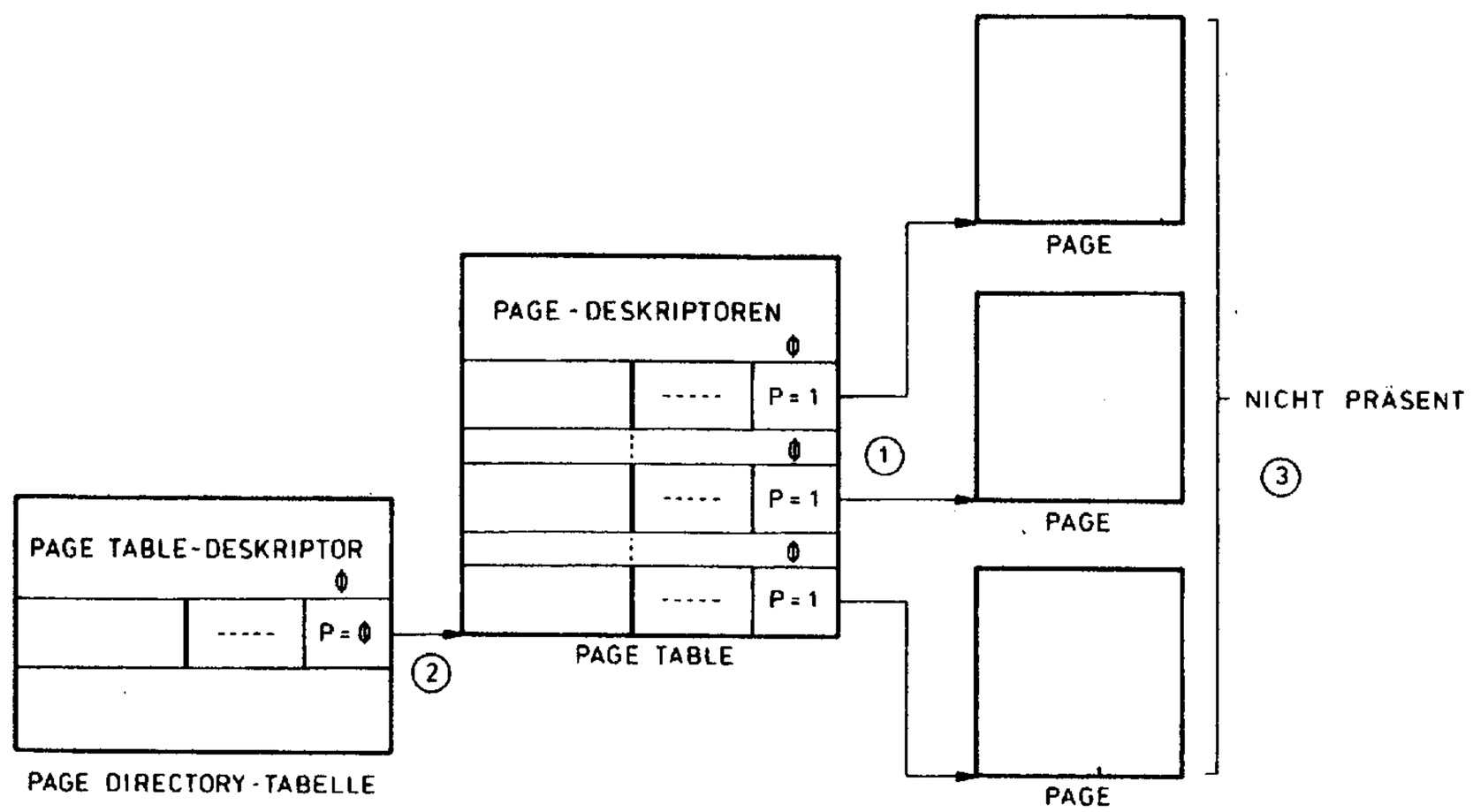
Wenn er in dieser Überprüfungskette einen Fehler entdeckt, führt er die nachfolgenden Überprüfungen ***nicht*** mehr durch.

Damit eine Anwender-Page als ***vorhanden*** betrachtet werden kann, müssen folgende Deskriptoren als präsent (P = 1) markiert sein:

- Deskriptor des Segments, der die Anwender-Page enthält
- Page-Table-Deskriptor
- Page-Deskriptor.

Unter der Voraussetzung, daß sich Anwender-Pages in präsenten Segmenten aufhalten, bestimmt das P-Bit im Page-Table-Deskriptor die Präsenz aller Anwender-Pages:

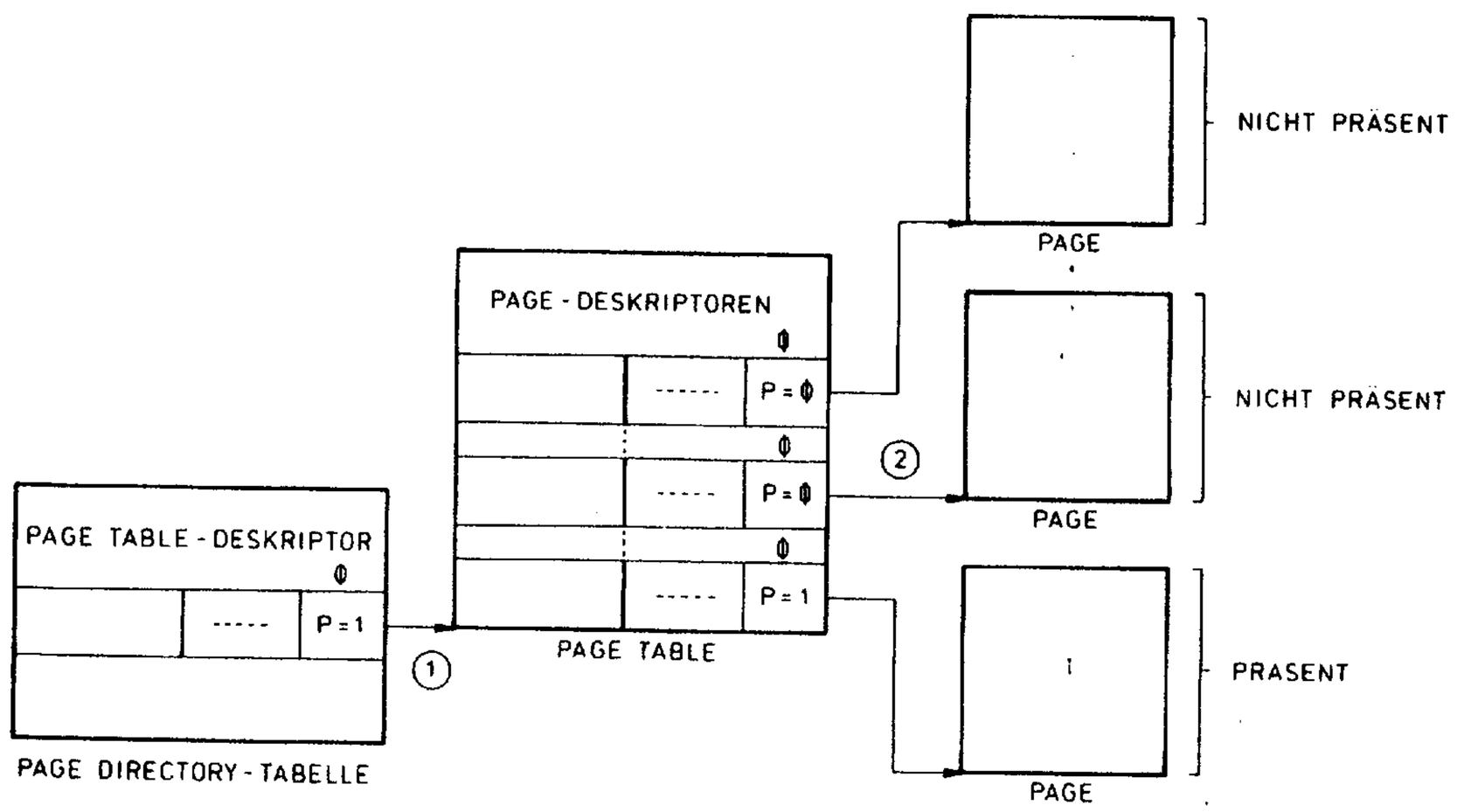
BEISPIEL 1:



Das angegebene Bild illustriert einen Fall, bei dem alle Anwender-Pages durch ihre Page-Deskriptoren als präsent gekennzeichnet sind (P = 1) ①.

Da aber das P-Bit im Page-Table-Deskriptor gelöscht ist (P = 0) ②, ist die „Page Table" nicht vorhanden. Aus diesem Grund sind ***alle*** Anwender-Pages als nicht-präsent markiert ③.

BEISPIEL 2:



Dieses Bild illustriert einen Fall, bei dem die „Page Table" durch ihren Deskriptor als präsent gekennzeichnet ist (P = 1) ①. Nun bestimmten die P-Bits in den Page-Deskriptoren, welche Anwender-Pages vorhanden sind oder nicht. Es ist zu erkennen, daß zwei Pages als nicht-präsent (P = 0) und eine „Page" als präsent (P = 1) markiert sind ②.

Wenn der „Page Fault Handler" wegen der Bedingung nicht-präsent aufgerufen worden ist, muß er feststellen, ob die Ursache beim Page-Table-Deskriptor oder beim Page-Deskriptor gelegen hat. Zu diesem Zweck prüft er die P-Bits in beiden Deskriptoren.

Abhängig von den P-Werten wird er dann Aktionen einleiten, die in der folgenden Tabelle zusammengefaßt sind:

P-Bit im „Page Table"-Deskriptor	P-Bit im Page-Deskriptor	Bedeutung für den „Page Fault Handler"	Aktion des Page Fault Handlers
0	0	Die „Page Table" und die Anwender-Page sind nicht präsent	Lesen der „Page Table" und derAnwender-Page vom sekundären Massenspeicher und in den Arbeitsspeicher einlagern
0	1	Die „Page Table" ist nicht präsent	Lesen der „Page Table" vom sekundären Massenspeicher und in den Arbeitsspeicher einlagern
1	0	Die Anwender-Page ist nicht präsent	Lesen der Anwender-Page vom sekundären Massenspeicher und in den Arbeitsspeicher einlagern
1	1	kommt nicht vor	keine

Damit der Befehl, der den Page-Fehler verursacht hat, erfolgreich wiederholt werden kann, muß der Handler nach dem Einlagerungs-Vorgang den Page-Table-Deskriptor oder den Page-Deskriptor oder beide als präsent markieren (P = 1).

Wenn nun anschließend der Prozessor den Handler mit IRETD (Doubleword-Version) verläßt, führt er den vorher fehlerverursachenden Befehl nochmals aus.

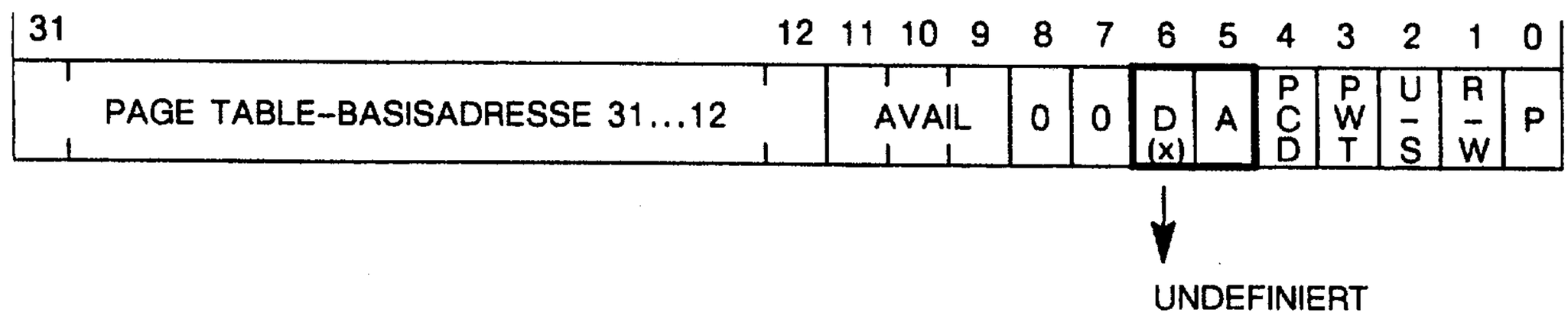
3.19.8 Ein- und Auslagern von Pages

Das Ein- und Auslagern von Pages übernimmt der sogenannte ***Swapper,*** der typischerweise als Prozedur implementiert ist und die Aufgabe des „Page Fault Handlers" vervollständigt. Er wird vom „Pade Fault Handler" immer dann aufgerufen, wenn eine Page ein- oder auszulagern ist.

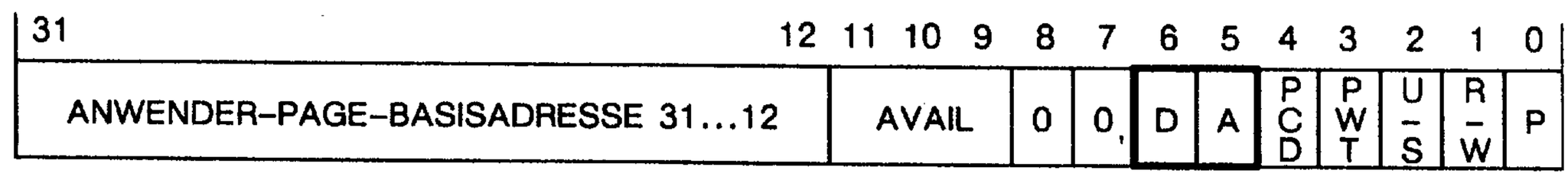
Dabei muß der Swapper Les- und Schreibzugriff auf die „Page Directories" und „Page Tables" haben. Damit er die Page finden und effizient erneuern kann, helfen ihm die Attribute ***A*** (Accessed) und ***D*** (Dirty) in den Page-Table-und Page-Deskriptoren.

Ihre Positionen sind in den Layouts der beiden Deskriptoren zu sehen:

'PAGE TABLE'-DESKRIPTOR:



'PAGE'-DESKRIPTOR:



Der 80486 setzt das Accessed-Bit A immer dann (A = 1), wenn ein ***Lese-*** oder ***Schreib***zugriff auf eine „Page Table" oder eine Page erfolgt.

Der Swapper hat nun die Möglichkeit, die Page-Referenz-Aktivitäten zu verfolgen, indem er periodisch das A-Bit im Page-Deskriptor testet.

Wenn er feststellt, daß das Accessed-Bit gelöscht ist (A = 0), weiß er, daß seit der letzten Überprüfung ***kein*** Zugriff auf die „Page" erfolgt ist. Eine solche Page ist eine gute Kandidatin für den sekundären Massenspeicher, so daß sie der Swapper ***auslagern*** kann.

Der 80486 setzt im Page-Deskriptor des ***Dirty***-Bit D, wenn er eine Anwender-Page ***beschreibt***.

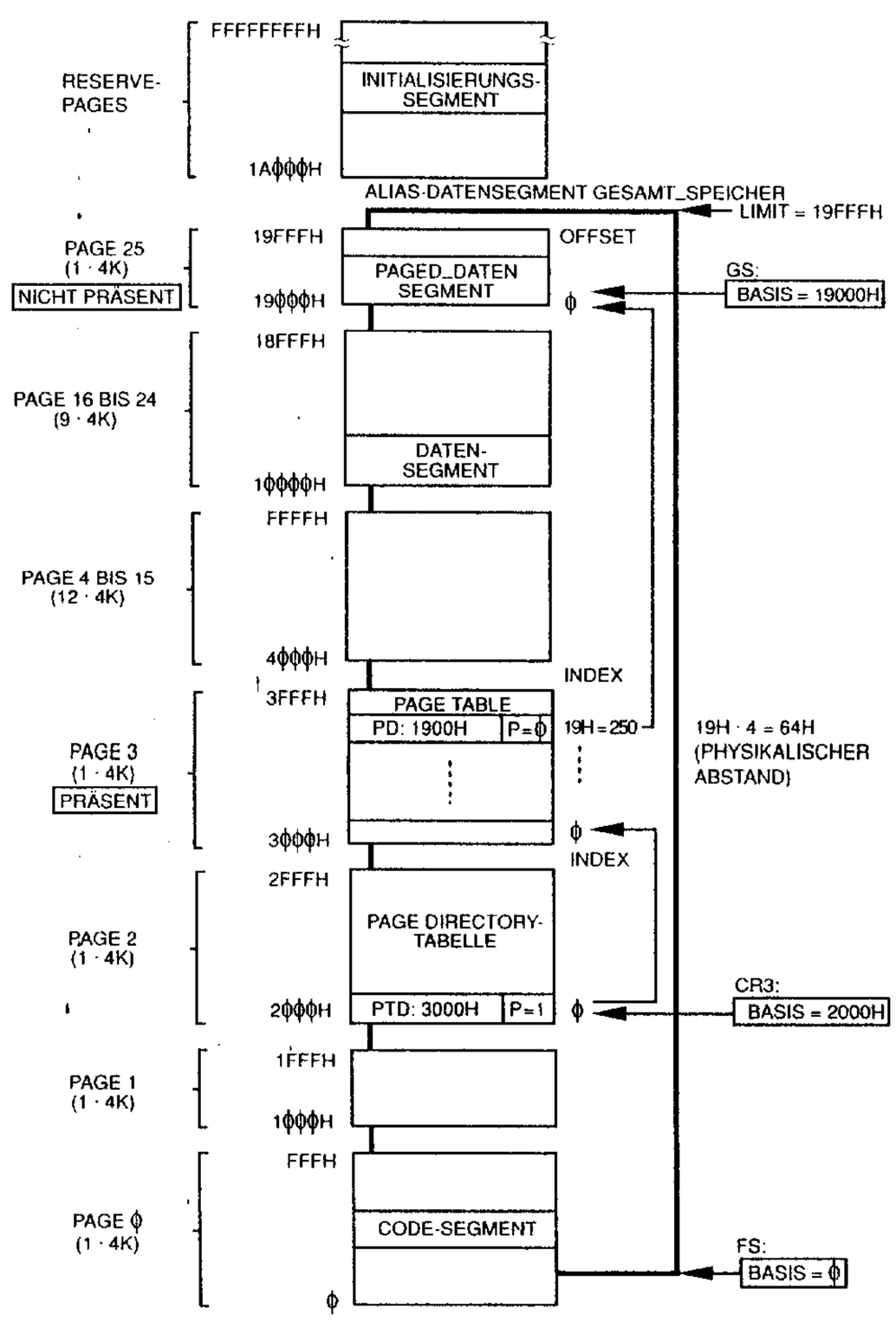
HINWEIS: Das Dirty-Bit ist im Page-Table-Deskriptor undefiniert.

Sobald der Swapper eine Page in den Arbeitsspeicher eingelagert hat, löscht er das Dirty-Bit (D = 0). Später kann er dann prüfen, ob die Page ab dem Einlagerungszeitpunkt vom Programm beschrieben worden ist.

- ***D = 1***
  Ist das Dirty-Bit gesetzt (D = 1), weiß der Swapper, daß die Page aktualisiert worden ist und daher ***keine*** Kopie im sekundären Massenspeicher existiert. Eine solche Page wird dann der Swapper zu einem geeigneten Zeitpunkt auslagern.
- ***D = 0***
  Ist das Dirty-Bit gelöscht (D = 0), weiß der Swapper, daß die Page ***nicht*** aktualisiert worden ist und deshalb ***eine*** Kopie im sekundären Massenspeicher existiert. Da der Swapper jetzt keine Kopie im sekundären Massenspeicher anfertigen muß, kann er den Speicherraum der Page zur Disposition stellen.

3.19.9 **Programmierbeispiel:** Zugriff auf Page-Table- und Page-Deskriptoren und deren Manipulation bei Page-Fehlern.

Das folgende ASM386/486-Programm repräsentiert das Fragment eines „Page Fault Handlers“ und nimmt Bezug auf ein Software-Arrangement, wie es im angegebenen Bild illustriert ist.



Die Struktur zeigt den flachen 4 Gigabyte physikalischen Adreßraum den 80486 als page-orientierten virtuellen Speicher.

- In ***Page 0*** befindet sich das Code-Segment.
- In ***Page16*** befindet sich das DATEN-Segment.
- In ***Page 25*** befindet sich das PAGED_DATEN-Segment
- In einer der Reserve-Pages befindet sich das INITIALISIERUNGS-Segment.

Die physikalische Basis-Adresse der „Page Directory“ kommt vom CR3-Register und liegt bei 2000H. Die physikalische Position der „Page Table“ kommt vom Page-Table-Deskriptor in der „Page Directory“;er liegt bei Index 0 und liefert als Page-Table-Basisadresse den Wert 3000H. Um die Software des „Page Fault Handlers“ zu vereinfachen, ist die „Page-Table“ als präsent markiert (P = 1).

In der „Page Table“ ist bei Index 19H (25D) der Deskriptor PD der Page 25 aufbewahrt. Er liefert als Basisadresse für diese Page den Wert 19000H und markiert sie als nicht-präsent (P = 0).

Das INITIALISIERUNGS-Segment schaltet den Prozessor in den „Protected Mode", gibt den Paging-Mechanismus frei und ruft das CODE-Segment in der Page 0 auf.

Im CODE-Segment versucht der Befehl

- **MOV EBX,GS:NICHT_PRAESENT_VAR**

die Variable NICHT_PRAESENT_VAR im PAGED_DATEN-Segment zu erreichen.

Er benützt dabei das GS-Register, das als Basisadresse genauso wie der Page-Deskriptor den Wert 19000H liefert. Das PAGED_DATEN-Segment ist also an einer Page-Grenze ausgerichtet. Da die Page als nicht-präsent markiert ist, wird der „Page Fault Handler" aufgerufen, der als NEAR-Prozedur im CODE-Segment enthalten ist.

Damit der „Page Fault Handler" die Page-Table- und Page-Deskriptoren durch einfache 32-Bit-Offset-Zeiger erreichen kann, benützt er das Alias-Datensegment GESAMT_SPEICHER. Dieses Segment wird vom Deskriptor im FS-Register beschrieben. Es hat die Basisadresse 0 und den Limitwert 19FFFH, umfaßt also Page 0 bis einschließlich Page 25.

Der „Page Fault Handler" benützt zunächst das CR2-Register, um den beim Page-Fehler beteiligten Page Table-Deskriptor in der „Page Directory" zu ermitteln. Da die „Page Table" gemäß Voraussetzung als präsent markiert ist, sind für ihn die Zugriffsrechte nicht von Interesse. Daher löscht er sie und bekommt damit die physikalische Basisadresse (3000H) der „Page Table".

Anschließend benützt er nochmals das CR2-Register, um den Page-Deskriptor in der „Page Table" zu ermitteln An diesem Deskriptor interessieren den „Page Fault Handler" nur die Zugriffsrechte. Daher löscht er die Page-Basisadresse 19000H, ruft den Swapper auf und stellt ihm die Page-Zugriffsrechte zur Verfügung.

Der Swapper liest als erstes die nicht-präsente Page vom sekundären Massenspeicher und lagert sie in den Arbeitsspeicher ein. Da ihm die Page-Zugriffsrechte zur Verfügung stehen, setzt er als nächstes das P-Bit und markiert sie damit als präsent. Anschließend wird mit IRETD (Doubleword-Version) der „Page Fault Handler" verlassen. Es erfolgt die Rückkehr zum Befehl MOV EBX,GS:NICHT_PRAESENT_VAR, der den Page-Fehler verursacht hat. Dieser Befehl wird nun wiederholt.

```
LOC      OBJ            LINE  SOURCE

                          1        NAME  B151
                          2
04C4B401                  3  PGPE   EQU    80000001          ;Protected Mode und Paging-Mechanismus
                          4                                   ;freigegeben
                          5  PUBLIC PAGE_DIRECTORY_BASIS
                          6
--------                  7  DATEN  SEGMENT RW USE32         ;Les- und beschreibbares Datensegment
                          8
00000000 ????????         9  PAGE_DIRECTORY_BASIS DD?        ;in dieser Variablen speichert BLD386
                         10                                   ;die physikalische Basisadresse der
                         11                                   ;Page Directory-Tabelle. Sie soll
                         12                                   ;2000H sein
--------                 13  DATEN  ENDS
                         14
```

```
--------                  15  PAGED DATEN SEGMENT RW USE32          ;Les- und beschreibbares Datensegment
                          16                                       ;Die Basisadresse = 19000H
                          17                                       ;(Festlegung durch BLD386)
0000 0000 26D2CBFF        18  NICHT_PRAESENT_VAR    DD -3419610D   ;Diese Variable haelt sich in einer
                          19                                       ;Page auf, die als nicht-praesent
                          20                                       ;markiert ist
--------                  21  PAGED_DATEN ENDS
                          22
--------                  23  GESAMT_SPEICHER SEGMENT RW USE32     ;Als Dummy-Segment definiertes Alias-
                          24                                       ;Datensegment. Es enthaelt 26 Pages
                          25                                       ;(Page 0 bis 25)
                          26                                       ;Basisadresse = 0
                          27                                       ;Der Limitwert = 19FFFH
                          28                                       ;(Festlegung durch BLD386)
--------                  29  GESAMT_SPEICHER ENDS
                          30
    ----                  31  INITIALISIERUNG SEGMENT ER USE16     ;Ausfuehrbares und lesbares Segment
                          32  ASSUME DS:DATEN
0000 6667A100000000     R 33  INIT:   NOV EAX,PAGE_DIRECTORY_BASIS ;EAX = Basisadresse der Page Directory-
                          34                                       ;Tabelle
    0007 660F22D8         35        MOV CR3,EAX                    ;In CR3 speichern
    000B 660F20C0         36        NOV EAX,CR0                    ;EAX bekommt augenblicklichen
                          37                                       ;Maschinen-Status
    000F 6600D01B4C404    38        OR EAX,PGE                     ;Setze PG- und PE-Bit
    0015 660F22C0         39        MOV CR0,EAX                    ;Protected Mode und Paging-Mechanismus
                          40                                       ;einschalten
0019 66EA00000000----   R 41        JMP FAR PTR START              ;Sprung in das Anwender-Code-Segment
                          42
--------                  43  INITIALISIERUNGS ENDS
***WARNING #377 IN 43; (PASS 2) SEGMENT CONTAINS PRIVILEGED INSTRUCTION(S)
                          44
--------                  45  CODE   SEGMENT E0 USE32              ;Nur ausfuehrbares Segment
                          46
00000000 B80000----     R 47  START: MOV EAX,PAGED_DATEN           ;EAX=Selektor des PAGED_DATEN-Segments
00000005 8EE8             48        MOV GS,AX                      ;GS zeigt zum PAGED_DATEN_Segment
                          49                                       ;(Basisadresse=19000H)
00000007 658B1D00000000   50        MOV EBX,GS:NICHT_PRAESENT_VAR;Dieser Befehl versucht, die Variable
                          51                                       ;NICHT_PRAESENT_VAR, die sich in einer
                          52                                       ;nicht-praesenten Page aufhaelt, in das
                          53                                       ;EBX-Register zu laden
                          54
                          55  ;************ An dieser Stelle wird das Programm unterbrochen und ************
                          56  ;************ der Page Fault Handler aufgerufen ************
                          57
0000000E                  58  PAGE_VAULT_HANDLER PROC NEAR
                          59
0000000E B80000----     R 60        MOV EAX,GESAMT_SPEICHER        ;EAX = Selektor des Alias-Datensegments
                          61                                       ;GESAMT_SPEICHER
00000013 8EE0             62        MOV FS,AX                      ;FS zeitt zum GESAMT_SPEICHER-Segment
                          63                                       ;(Basisadresse=0)
                          64
                          65  ;************ Page Table-Deskriptor und die physikalische Basisadresse ************
                          66  ;************ der Page Table ermitteln ************
                          67
00000015 0F20D0           68        MOV EAX,CR2                    ;EAX = Lineare Adresse (00019000H) der
                          69                                       ;Variablen NICHT_PRAESENT_VAR
                          70                                       ;*Index des Page Table-Deskriptors im
                          71                                       ;DIR-Feld=0. Page Table 0 selektieren
                          72                                       ;*Index des Page-Deskriptors im TABLE-
                          73                                       ;Feld=19H (25D). Page 25 selektieren
                          74                                       ;*Abstand der Variablen innerhalb der
                          75                                       ;Page ist im OFFSET-Feld mit 0 fest-
                          76                                       ;gelegt
```

```
00000018 C1E816          SHR EAX,22                    ;Index des Page-Deskriptors im
                                                       ;DIR-Feld isolieren. EAX bekommt als
                                                       ;Index den Wert 0
0000001B C1E002          SHL EAX,2                     ;Physikalischen Abstand relativ zur
                                                       ;Page Directory-Basis bilden.
                                                       ;(Multiplikation des Index mit 4)
                                                       ;EAX=0x4=0
0000001E 0F20D9          MOV ECX,CR3                   ;Physikalische Basisadresse der
                                                       ;Page Directory-Tabelle laden
                                                       ;ECX=2000H
00000021 01C8            ADD EAX,ECX                   ;Physikalische Adresse des Page Table-
                                                       ;Deskriptors bilden
                                                       ;EAX=2000H
00000023 648B18          MOV EBX,DWORD PTR FS: [EAX]   ;EBX bekommt den Page Table-Deskriptor
00000026 81E300F0FFFF    AND EBX, 0FFFFF000H           ;Zugriffsrechte loeschen. EBX enthaelt
                                                       ;die physikalische Basisadresse 3000H
                                                       ;der Page Table

;************ Page Deskriptor und die Page-Zugriffsrechte ermitteln ************

0000002C 0F20D0          MOV EAX,CR2                   ;EAX=lineare Adresse (00019000H)
                                                       ;der Variablen NICHT_PRAESENT_VAR
0000002F C1E80C          SHR EAX,12                    ;Index des Page-Deskriptors im TABLE-
                                                       ;Feld isolieren. EAX bekommt als Index
                                                       ;den Wert 19H (25D)
00000032 C1E002          SHL EAX,2                     ;Physikalischen Abstand relativ zur
                                                       ;Page Table-Basis bilden
                                                       ;(Multiplikation des Index mit 4)
                                                       ;EAX=19Hx4=64H
00000035 25FF0F0000      AND EAX,0FFEN                 ;Index des Page Table-Deskriptors
                                                       ;(wenn er ungleich 0 ist)  loeschen
                                                       ;EAX=64H
0000003A 01C3            ADD EBX,EAX                   ;Physikalische Basisadresse des Page-
                                                       ;Deskriptors bilden
                                                       ;EBX=3000H+64H=3064H
0000003C 648B0B          MOV ECX,DWORD PTR FS:[EBX]    ;ECX bekommt den Page-Deskriptor
0000003F 81E1FF0F0000    AND ECX,0FFEN                 ;Page-Adresse 19000H loeschen und damit
                                                       ;Zugriffrechte isolieren. ECX enthaelt
                                                       ;die augenblicklichen Page-Zugriffs-
                                                       ;rechte (P=0)

;****** An dieser Stelle ruft der Page Fault Handler den Swapper auf. ******
;****** Dieser liest die nicht-praesente Page vom sekundaeren ******
;****** Massenspeicher und lagert sie in den Arbeitsspeicher ein ******

00000045 0F20D0          MOV EAX,CR2                   ;EAX=lineare Adresse (00019000H) der
                                                       ;Variablen NICHT_PRAESENT_VAR
00000048 2500F03F00      AND EAX,0003FF000H            ;DIR- und OFFSET-Felder in der linearen
                                                       ;Adresse loeschen (wenn sie ungleich 0
                                                       ;sind)
                                                       ;EAX=00019000H
0000004D 09C8            OR EAX,ECX                    ;Bilde Page-Deskriptor durch
                                                       ;Hinzufuegen der augenblicklichen
                                                       ;Zugriffsrechte
0000004F 0D01000000      OR EAX,1                      ;Markiere die Page als praesent
00000054 648903          MOV DWORD PTR FS:[EBX],EAX    ;Speichere den neuen Page-Deskriptor
                                                       ;in der Page-Table
00000057 CF              IRETD                         ;Rueckkehr zu dem Befehl, der den
                                                       ;Page-Fehler verursacht hat
                                                       ;und wiederholen

00000058                 PAGE_FAULT_HANDLER ENDP
```

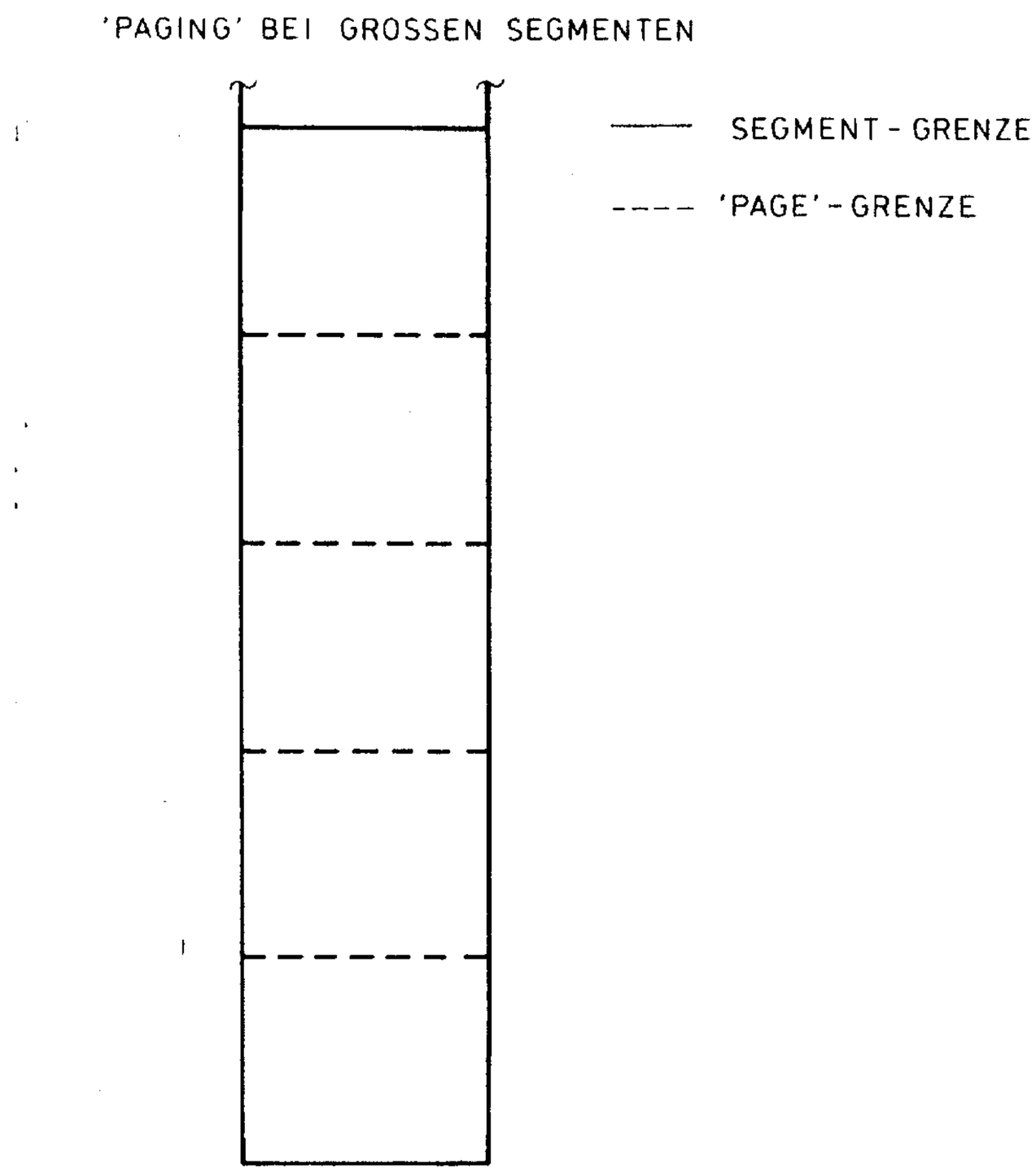
```
--------                    140   CODE   ENDS
*** WARNING #377 IN 140, (PASS 2) SEGMENT CONTAINS PRIVILEGED INSTRUCTION(S)
                            141
                            142          END
ASSEMBLY COMPLETE,  2 WARNING, NO ERRORS.
```

3.19.10 Auswahlkriterien für Segment- oder Page-Schutz

Der Typ des Schutzes (Segment- oder Page-Schutz), der für ein System gewählt wird, hängt vom Grad der Segment-Orientierung ab und kann zwischen folgenden zwei Extremen liegen.

Betriebssysteme mit wenigen großen Segmenten

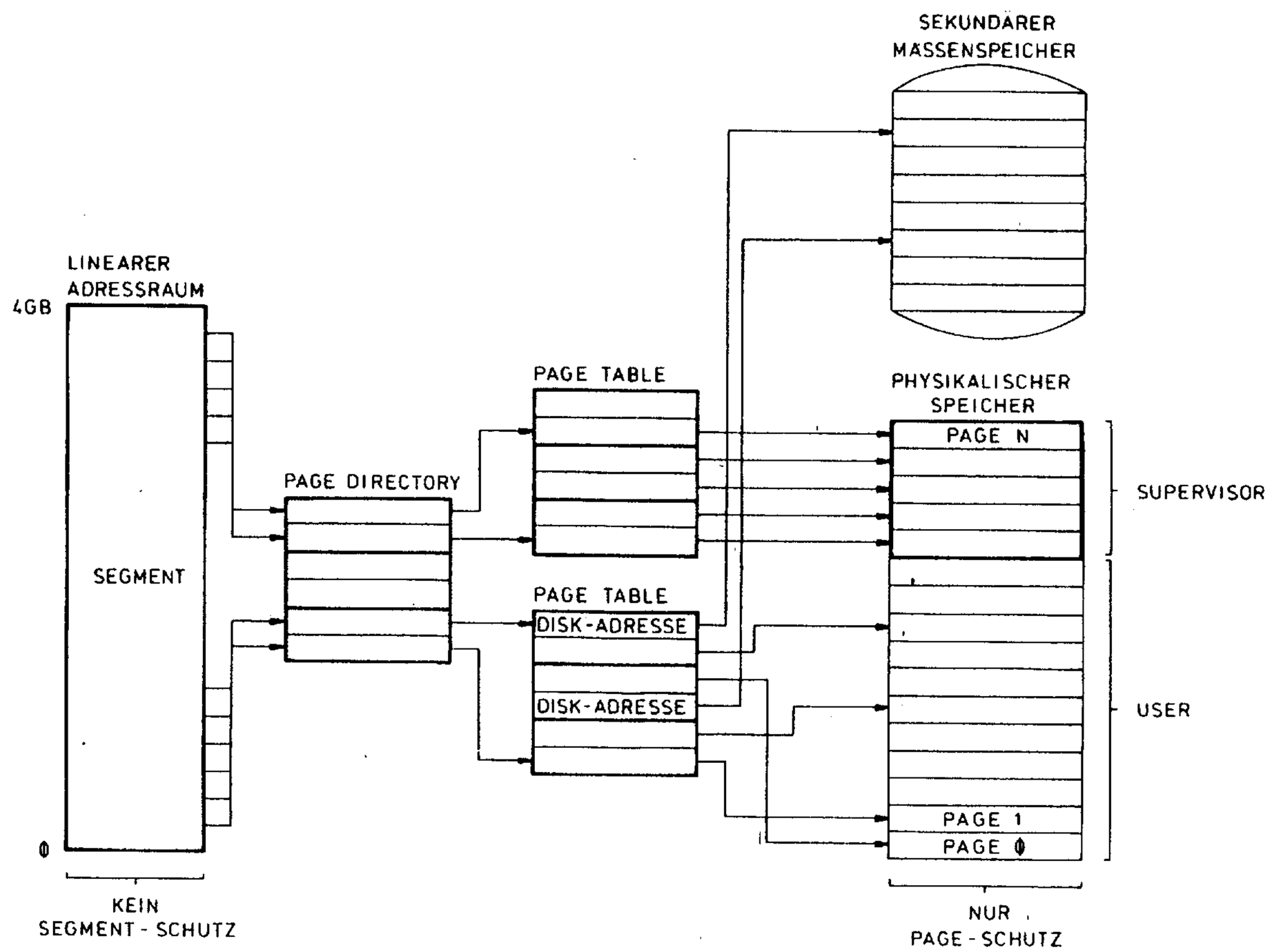
Ein Betriebssystem, das nur einige ***große*** Segmente definiert, wird Pages als integrale Bestandteile der Segmente benützen. Dabei beginnt jedes Segment an einer Page-Grenze und ist mindestens eine Page lang. Eine solche Konfiguration zeigt das folgende Bild:



Hier ist die ***Page*** die ***Zentral-Einheit*** und dient:

- der Zuweisung des physikalischen Speichers,
- dem Schutz (nur Page-Schutz),
- zum Ein- und Auslagern in virtuellen Speicher-Systemen.

Wie in einem solchen System der lineare Adreßraum auf den physikalischen Speicher abgebildet wird, ist im folgenden Bild zu sehen.



Das Bild zeigt, daß der lineare Adreßraum, der von der Segmentierungs-Einheit des Prozessors definiert wird, ein ***einzelnes*** Segment mit einer Länge im Bereich von 0 bis maximal 4 Gigabytes enthält.

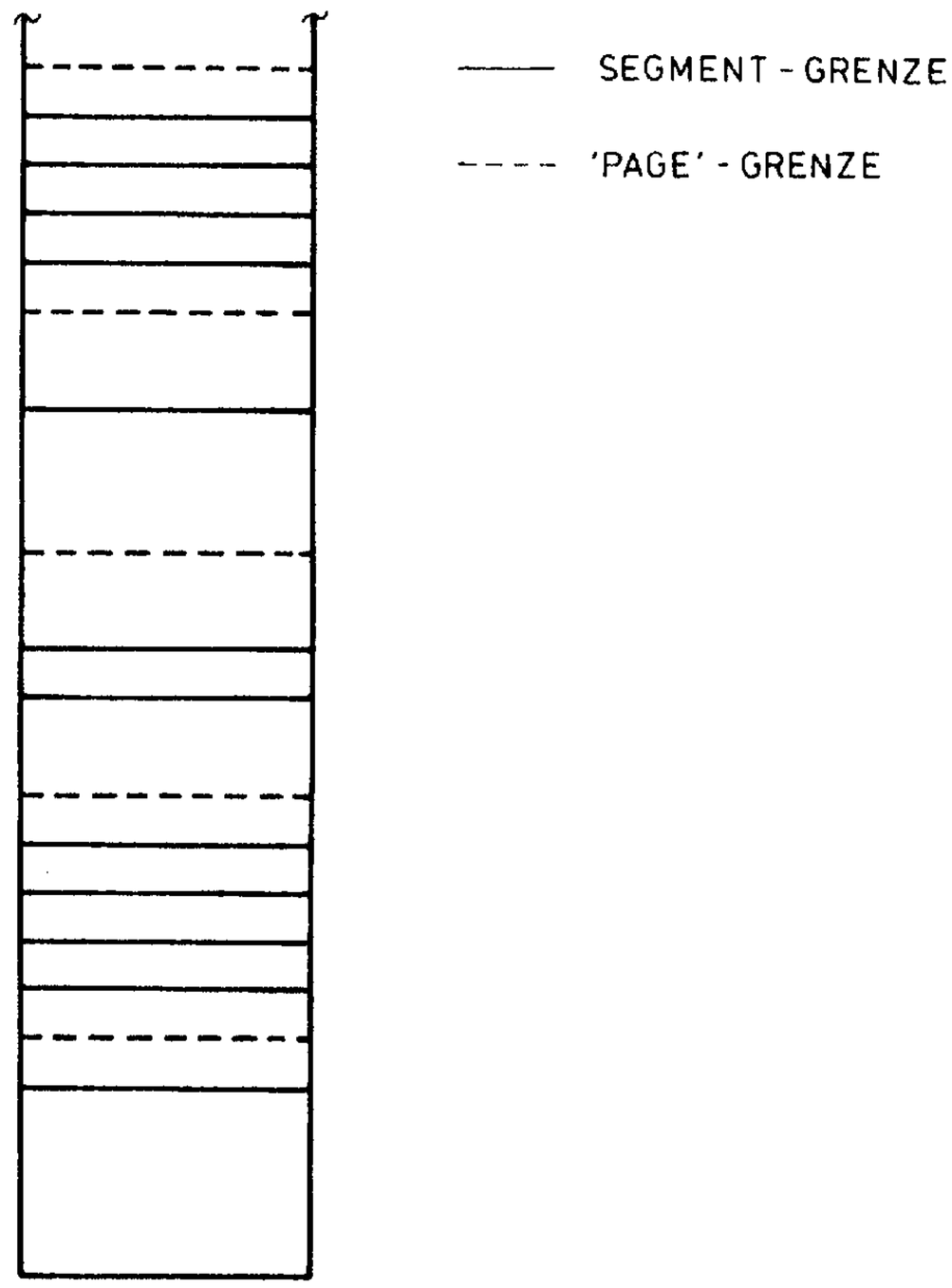
Da das System die Speicher-Adressierbarkeit über den ***Page-Schutz*** regelt, beansprucht es ***keinen*** Segment-Schutz. Daher bildet der nachgeschaltete Paging-Mechanismus das Segment auf „Supervisor Pages" und „User Pages" mit individuellen Zugriffsrechten ab. Der Betriebssystem-Code kann damit vor unautorisierten Zugriffen des Users geschützt werden. Eine weitere wesentliche Funktion der Pages liegt in der Unterstützung der virtuellen Speicheradressierung. Die können daher zusätzlich zum Ein- und Auslagern benützt werden. Zu diesem Zweck enthält im Beispiel eine der „Page Tables" die erforderlichen Disk-Adreßzeiger.

Betriebssysteme mit sehr vielen kleinen Segmenten

> Ein Betriebssystem, das hunderte oder tausende von Segmenten mit einer Durchschnittslänge von einigen 10 oder 100 Bytes definiert, würde Speicherplatz verschwenden, wenn es diese Segmente an Page-Grenzen ausrichten würde.

Ein solches System wird statt dessen die Segmente ***ohne*** Berücksichtigung der Segment- und Page-Grenzen auf die Pages abbilden. Es besteht außerdem die Möglichkeit, zusammengehörige Segmente als „Cluster" in der gleichen Page aufzubewahren. Dies alles illustriert das folgende Bild:

'PAGING' BEI KLEINEN SEGMENTEN

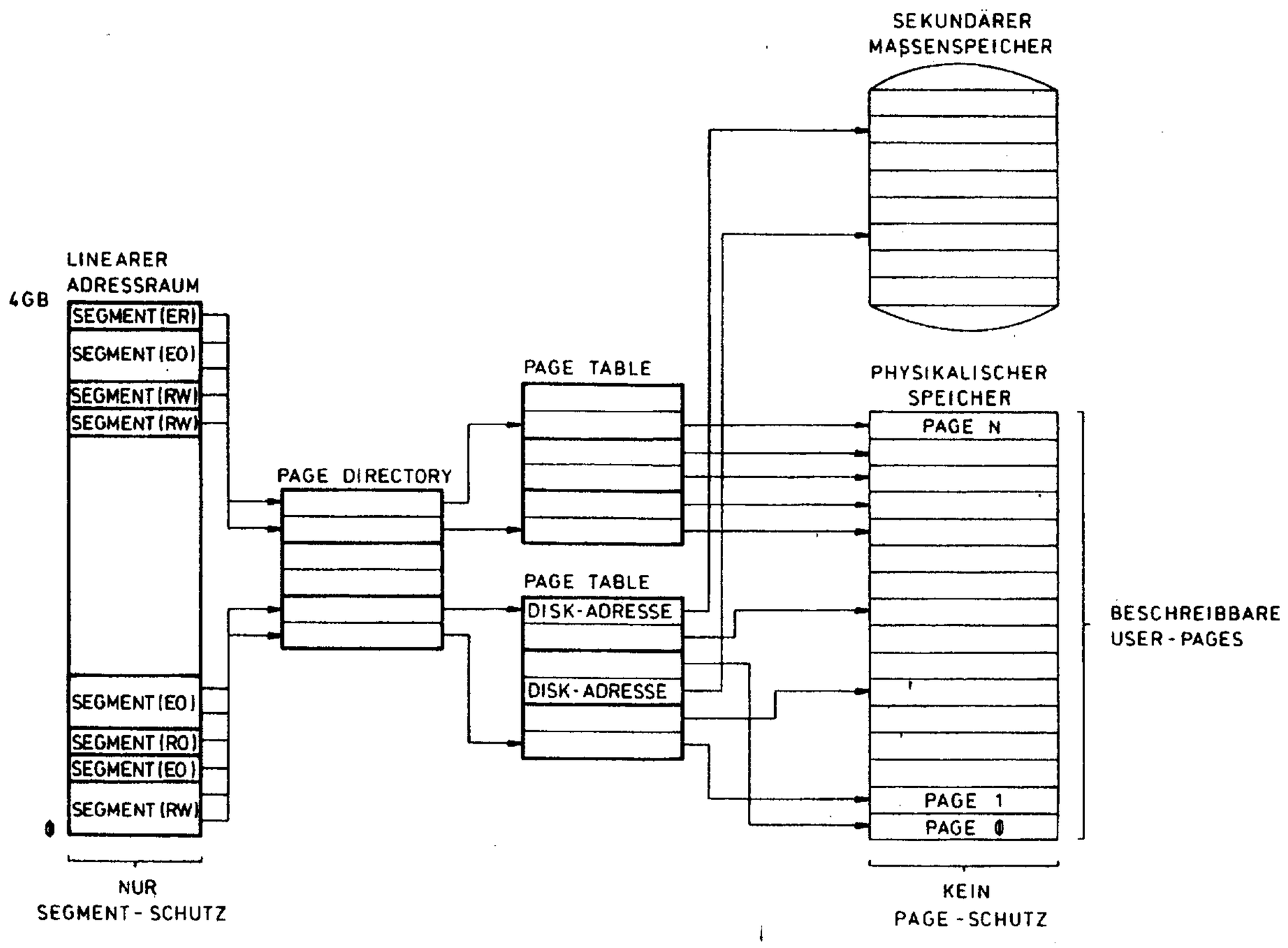


Hier ist das ***Segment*** die ***Zentral-Einheit*** und dient:

- der Zuweisung des linearen Adreßraums,
- dem Schutz (nur Segment-Schutz).

Da nur der Segment-Schutz existiert, definieren die Page-Table- und Page-Deskriptoren ***alle*** Pages als ***beschreibbare „User Pages“*** und schalten damit den Page-Schutz aus. Sollte das System einen page-orientierten virtuellen Speicher unterstützen, werden die Pages nur zum Ein- und Auslagern benützt.

Wie in einem solchen System der lineare Adreßraum auf den physikalischen Speicher abgebildet wird, ist im folgenden Bild zu sehen:



Es zeigt, daß der lineare Adreßraum, der von der Segmentierungs-Einheit des Prozessors definiert wird, ***viele*** Segmente mit unterschiedlichen Längen und individuellen Schutz-Attributen (ER, EO, RO, RW) enthält.

Da das System die Speicher-Adressierbarkeit über den ***Segment-Schutz*** regelt, beansprucht es ***keinen*** Page-Schutz. Daher bildet der nachgeschaltete Paging-Mechanismus die Segmente auf beschreibbare „User Pages" ab. Ihre wesentliche Funktion liegt aber in der Unterstützung der virtuellen Speicheradressierung, so daß sie ausschließlich zum Ein- und Auslagern benützt werden. Zu diesem Zweck enthält im Beispiel eine der „Page Tables" die erforderlichen Disk-Adreßzeiger.

Ein Speicher, der in der angegebenen Weise arrangiert ist, wird als ***hybrid*** bezeichnet, da das „Memory Management" die „Überkreuzung" von zwei Einheiten unterschiedlicher Herkunft (Segmente und Pages) unterstützt.