KAPITEL 1
TASK-MANAGEMENT

1.1 Task-Grundoperationen

Jedes Software-System des x86/Pentium im Protected Mode besteht aus dem Betriebssystem und den Anwendungen, die von der Architektur als eine Vielzahl von Tasks betrachtet werden.
Der Begriff Task ist noch nicht allgemeingültig definiert und so bezieht man sich am häufigsten auf das Konzept «Programm in Ausführung». Ein Programm ist zunächst eine «leblose», logische Einheit. Erst wenn der Prozessor das Programm ausführt und damit zum «Leben erweckt», wird es zur aktiven Einheit, und wir nennen eine solche Einheit dann Task.
Eine wesentliche Kernfunktion des Betriebssystems ist, die Aktivitäten der individuellen Tasks zu koordinieren. Oft muß dabei von einer Task auf eine andere umgeschaltet werden. Diese Aufgabe unterstützt eine auf dem Chip integrierte Hochgeschwindigkeits-Hardware.
Besteht ein Software-System aus einer Vielzahl von Tasks, dann konkurrieren die einzelnen Tasks um den Prozessor. Es kann aber immer nur eine Task mit der CPU in Verbindung gebracht werden, während die restlichen Tasks dann warten müssen. Wir fordern daher vom Betriebssystem ein Steuermodul, das solche Verbindungen herstellt. Die Frage ist, nach welchem Kriterium sich dieses Steuermodul für die eine oder andere Task entscheidet. So könnte z. B. jede Task eine bestimmte **Priorität** haben. Wie in diesem Fall die Aktion des Steuermoduls aussieht, ist schematisch im folgenden Bild 1 illustriert:

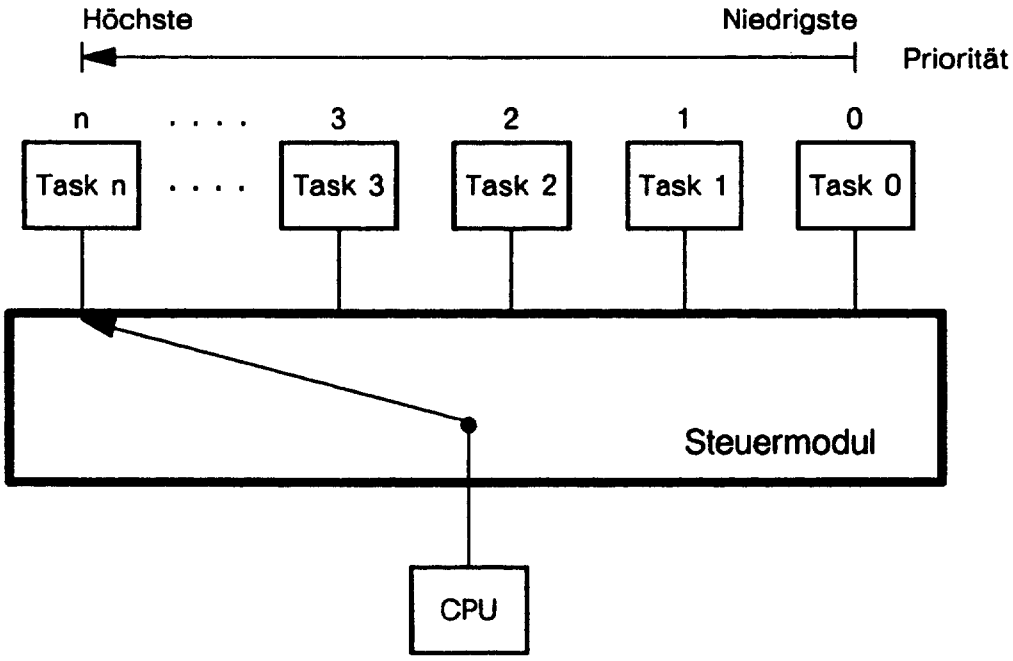


Bild 1: Jede Task hat eine bestimmte Priorität

Hier hat die Task 0 die **niedrigste** Priorität 0 und die Task n die **höchste** Priorität n. Das Steuermodul des Betriebssystems beginnt seine Arbeit, indem es als erstes die höchstpriorisierte Task n mit der CPU verbindet. Wenn sie beendet ist, dann bringt das Steuermodul die Task mit der nächstniedrigeren Priorität in den Besitz der CPU usw.

In einem Software-System können aber auch alle Tasks gleichwichtig sein und somit **gleiche** Priorität haben. Ein solches System zeigt das folgende Bild 2:

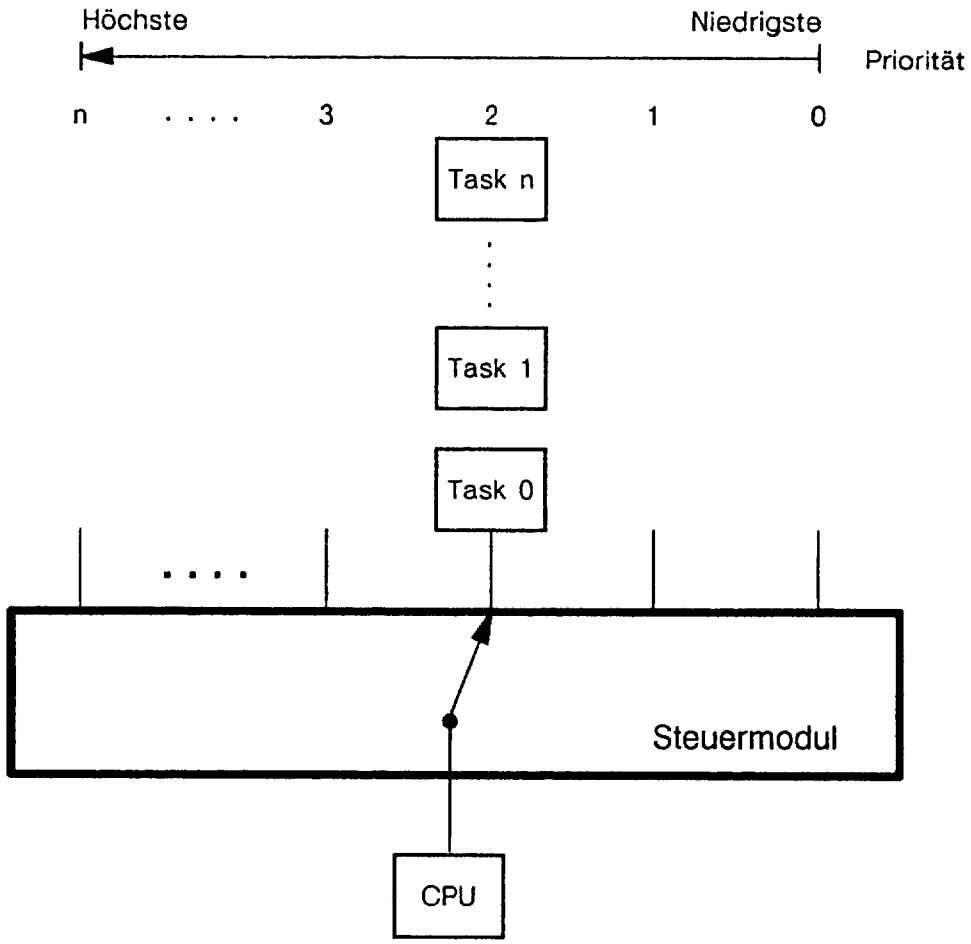


Bild 2: Alle Tasks haben gleiche Priorität

Im Beispiel stellt das Steuermodul fest, daß alle Tasks die Priorität 2 haben und somit die restlichen Prioritäten nicht beanspruchen. Sie befinden sich in einer sogenannten **Warteschlange**, und das Steuermodul verbindet als erstes die Task 0 mit der CPU, danach die Task 1 und zuletzt die Task n. Die Tasks nehmen also nach dem Prinzip FIFO (First In First Out) der Reihe nach die CPU in Besitz.

Betrachten wir das Steuermodul etwas genauer. Es verfügt darüber, welche Task dem Prozessor zugewiesen wird. Sinnvollerweise erledigen diese Aufgabe zwei Teilmodule. Das erste Modul **entscheidet** darüber, in welcher Reihenfolge die Tasks den Prozessor in Besitz nehmen dürfen. Es verkörpert die Entscheidungsstrategie und steuert somit den Ablauf des Systems. Oft wird die Routine in diesem Modul auch als **Scheduler** (Ablaufsteuerer) bezeichnet. Die schnelle tatsächliche Verbindung zwischen der ausgewählten Task und dem Prozessor übernimmt das zweite Modul. Es enthält eine spezielle Zuteilungsroutine, die auch als **Dispatcher** (schneller Erlediger) bekannt ist.

Die Aufgabenverteilung des Steuermoduls auf den Scheduler und den Dispatcher ist schematisch im folgenden Bild 3 dargestellt:

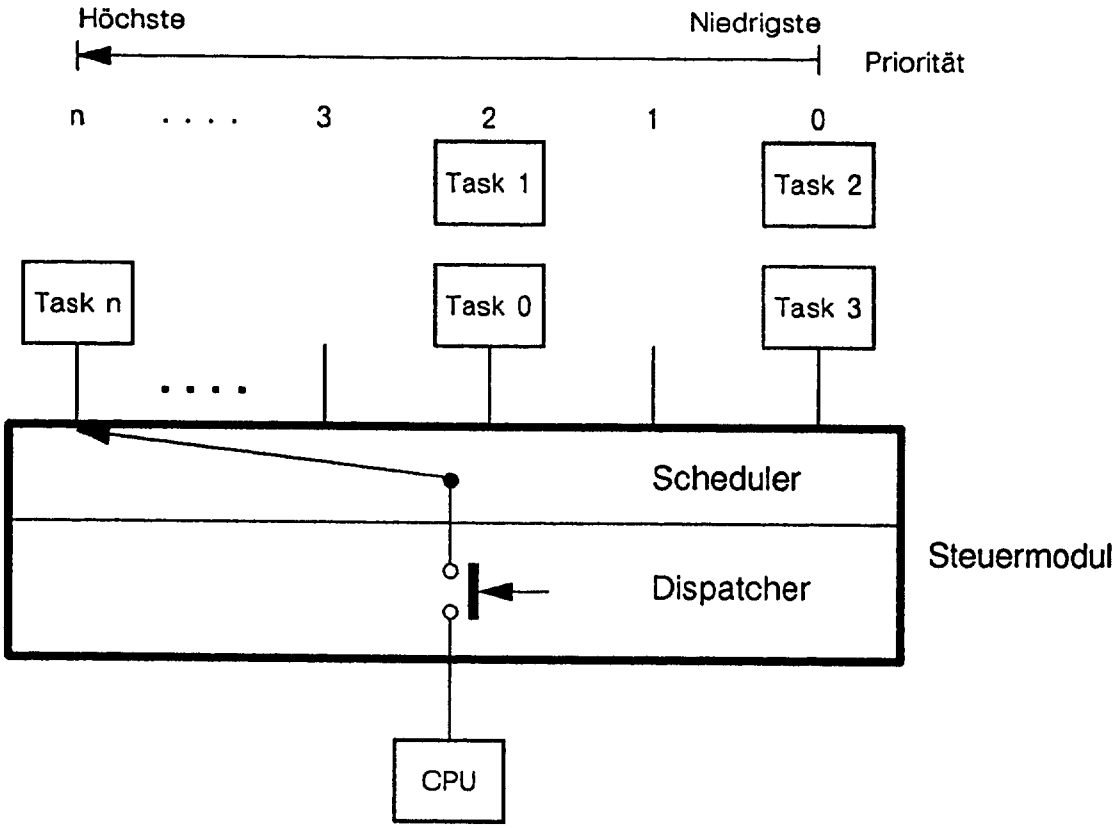


Bild 3: Aufgabenverteilung im Steuermodul

In jedem Multitasking-System gibt es mindestens eine Task, die **aktiv** ist; dann eine Anzahl von Tasks, die **bereit** sind, aktiv zu werden; und weitere Tasks, die auf ein bestimmtes Ereignis **warten**. Mit anderen Worten, jede Task befindet sich in einem bestimmten Zustand. Nehmen wir zur Verdeutlichung wieder unser bekanntes Schema:

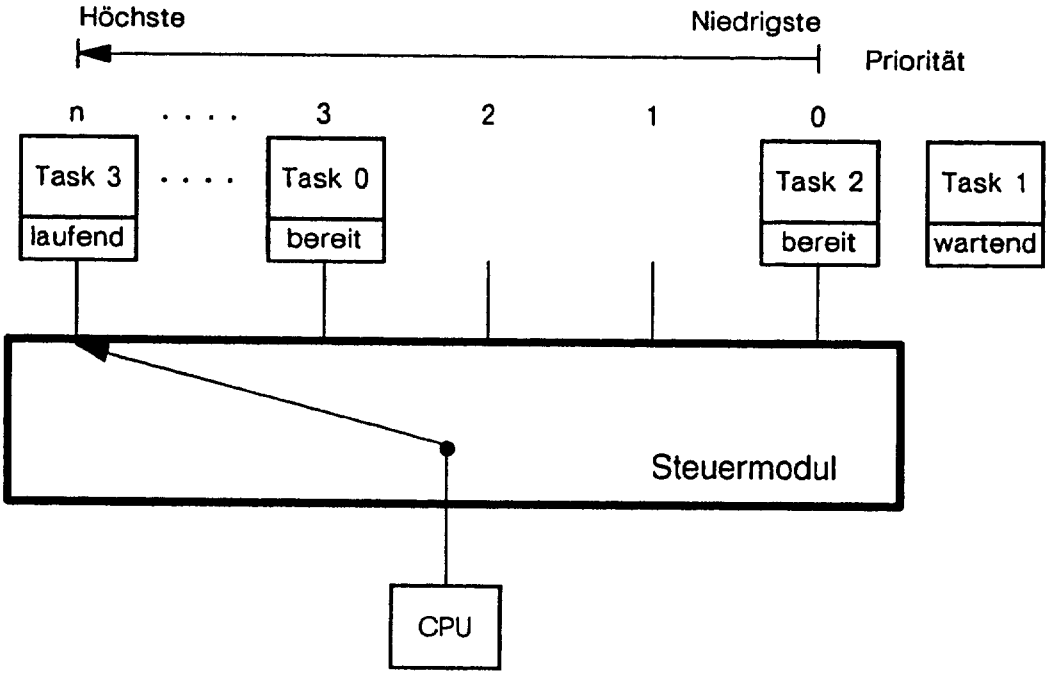


Bild 4: Tasks in verschiedenen Zuständen

Hier bewerben sich die Tasks 0 und 2 um die CPU und möchten ihre Arbeit aufnehmen; sie befinden sich im Zustand **bereit** (ready). Dagegen wartet die Task 1 auf ein bestimmtes Ereignis; sie befindet sich im Zustand **wartend** (waiting). Von den bereiten Tasks wählt das Steuermodul die Task 3 aus: Nimmt diese Task die CPU in Besitz, dann verläßt sie den Zustand bereit und geht in den Zustand **laufend** (running) über. In diesem Zustand bleibt sie solange, bis sie entweder freiwillig terminiert oder dazu gezwungen wird.
Während die Task 3 läuft, bleiben die Tasks 0 und 2 weiterhin lauffähig. Ihr Zustand ändert sich nicht, sie sind lediglich eine Zeit lang gestoppt.
Wenn die Task 3 terminiert, weil sie auf ein bestimmtes Ereignis wartet (z. B. auf eine Botschaft von einer anderen Task), dann verläßt sie den Zustand laufend und geht in den Zustand wartend über. Wenn umgekehrt bei Task 1 das erwartete Ereignis eintrifft, dann verläßt sie den Zustand wartend, geht in den Zustand bereit und bewirbt sich zusammen mit Task 0 und Task 2 ebenfalls um die CPU. Die entstandene Situation ist im folgenden Bild 5 illustriert:

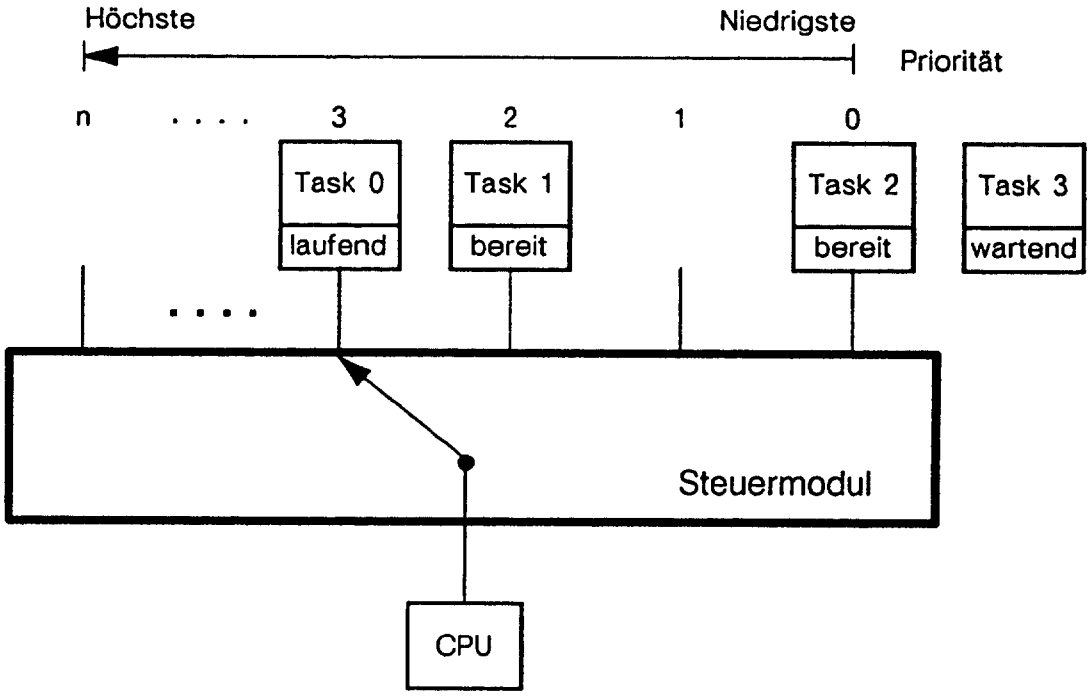


Bild 5: Veränderte Zustandssituation

Task 0 hat nun die höchste Priorität im System. Sie geht über in den Zustand laufend, während Task 1 und Task 0 im bereiten Zustand noch gestoppt sind. Task 3 hat sich «verabschiedet» und befindet sich im Augenblick im Zustand wartend.
Die Tasks ändern also im Verlauf ihres «Lebens» ständig ihre Zustände. So lassen sich zwischen den Zuständen bereit, wartend und laufend drei Übergänge (Transitionen) feststellen und als Zustandsdiagramm darstellen. Siehe Bild 6:

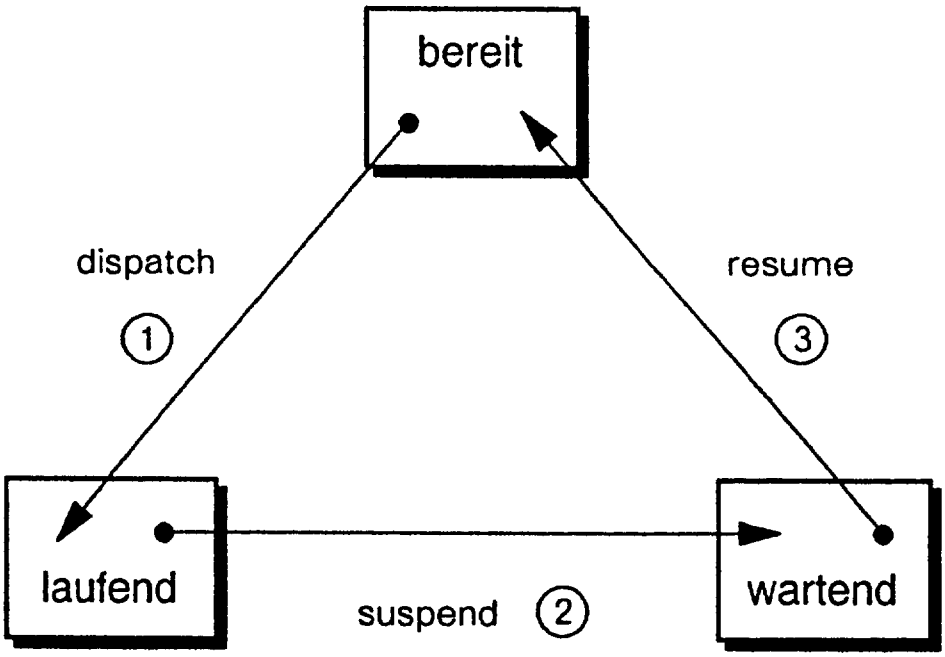


Bild 6: Zustandsdiagramm mit drei Zuständen

1, Der Scheduler wählt eine der arbeitsbereiten Tasks aus und der Dispatcher erledigt die schnelle Zuweisung an den Prozessor. Die ausgewählte Task verläßt den Zustand bereit und geht über in den Zustand laufend.
Wir können diesen Zustandsübergang formal folgendermaßen beschreiben:

dispatch (Taskname); bereit → laufend

2, In einem Multitasking-System arbeiten die Tasks oft eng zusammen und benutzen dabei einen gemeinsamen Speicher (enge Kopplung), den jede Task sowohl lesen als auch beschreiben kann. Ein solcher Speicher eignet sich daher als Kommunikationsort, um Informationen und Daten auszutauschen. Eine Task, die an aktuellen Daten im gemeinsamen Speicher interessiert ist, benutzt vorhandene Daten, oder wartet, bis welche eintreffen. Im letzteren Fall verläßt die Task den Zustand laufend und geht über in den Zustand wartend. Dieser Zustandsübergang lautet:

suspend (Taskname); laufend → wartend

3, Wenn aktuelle Daten im gemeinsamen Speicher eintreffen, dann verläßt die wartende Task ihren Zustand und geht über in den Zustand bereit. Dieser Zustandsübergang lautet:

resume (Taskname); wartend → bereit

Erst jetzt kann der Scheduler die wieder arbeitswillige Task auswählen und der Dispatcher sie dem Prozessor zuteilen. So setzt die Task an der Stelle, an der sie vorher in den Wartezustand gegangen ist, ihre Arbeit fort und ist sicher, daß sich jetzt im gemeinsamen Speicher aktuelle Daten befinden.

1.2 Task-Identifikation

Jede Task in einem x86/Pentium-Betriebssystem wird durch **zwei** Datenstrukturen identifiziert und zwar durch

- das Task-Status-Segment (TSS) und
- den Task-Control-Block (TCB). Siehe Bild 7:

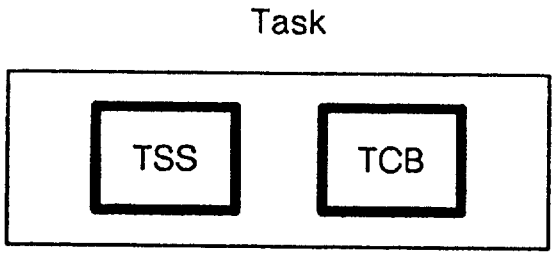


Bild 7: Identifikation einer Task

Das **Task-Status-Segment** ist ein Systemsegment, das von der x86/Pentium-Architektur unterstützt wird. Es enthält den gesamten Prozessor-Status (CPU-Register) und damit alle Informationen, die der **Prozessor** braucht, um eine Task zu managen. Das folgende Bild 8 zeigt das Layout des Task-Status-Segments einschließlich der mit ihm verbundenen Hardware-Struktur:

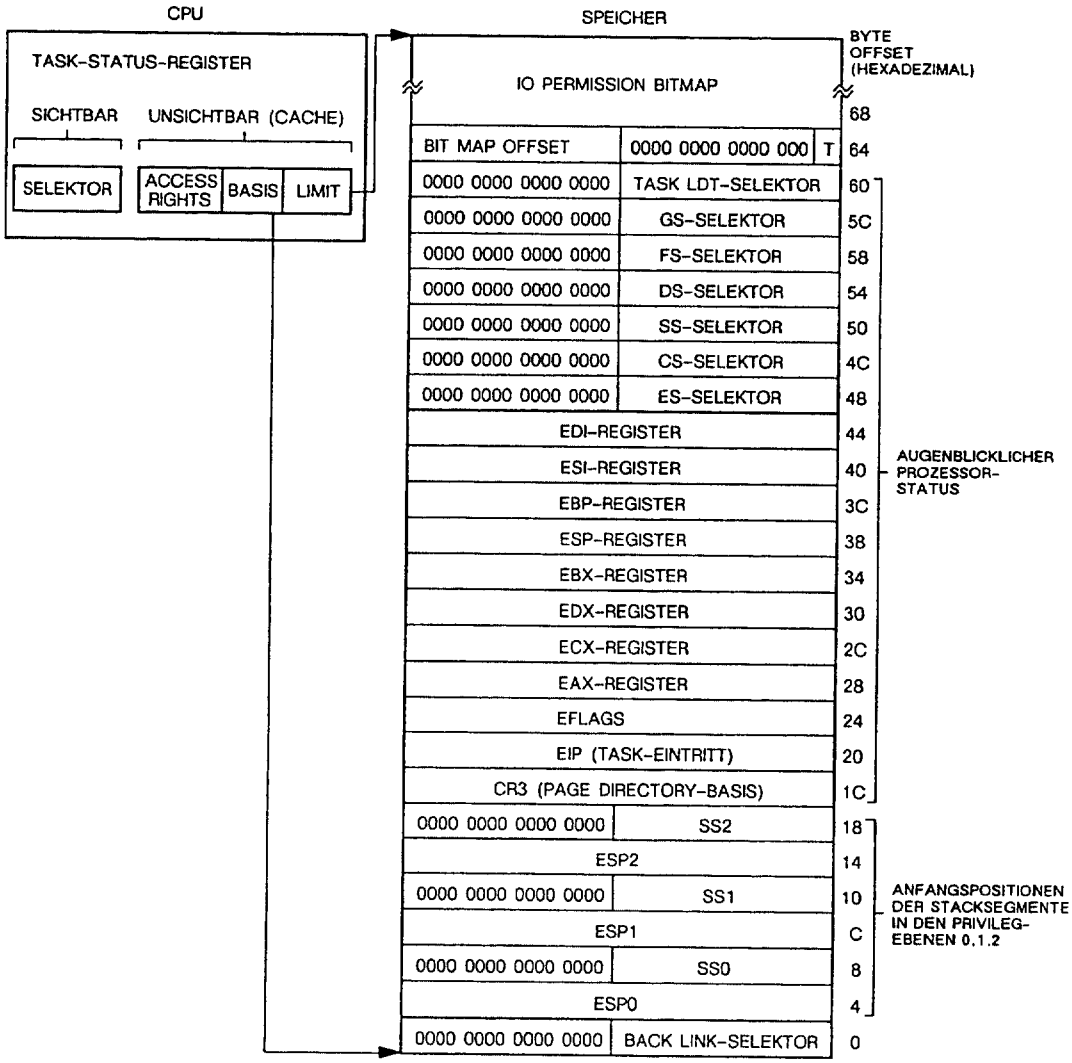


Bild 8: x86/Pentium-Task-Status-Segment

Wie zu sehen ist, besteht dieses Segment aus mindestens 26 Feldern zu je 32 Bits, die in zwei Klassen unterteilt sind:

Dynamische Felder:
Sie enthalten

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

Statische Felder:
Sie enthalten

- den Task-LDT-Selektor zur LDT (**L**okale **D**eskriptor-**T**abelle) der augenblicklichen Task.
- Das Register CR3, das die Basisadresse der Page-Directory-Tabelle für die augenblickliche Task enthält. Dieses Feld ist nur dann aktiv, wenn der Paging-Mechanismus eingeschaltet ist.
- Die Zeiger SS0:ESP0, SS1:ESP1 und SS2:ESP2, die die Anfangspositionen der Stack-Segmente in den Privileg-Ebenen 0 bis 2 festlegen.
- Das T-Bit (Debug Trap Bit), das bei einem Taskwechsel eine sogenannte Debug Exception bewirkt.
- Den I/O-Map-Offset.

Der **Task-Control-Block** TCB enthält alle Informationen, die das Steuermodul des **Betriebssystems** braucht, um eine Task zu managen. Dazu gehören die **Tasknummer**, der **Taskstatus**, die **Taskpriorität** und ein Zeiger (naechste_task), der die Verkettung von TCB's ermöglicht, wenn es im System mehrere Tasks mit gleicher Priorität gibt. Das folgende Bild zeigt das Layout des Task-Control-Blocks:

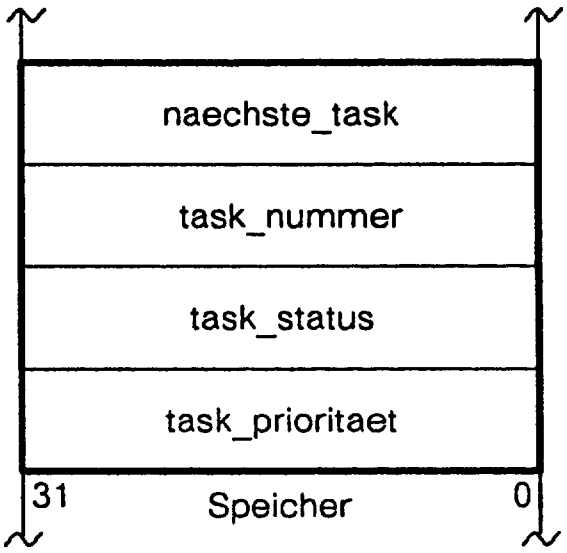


Bild 9: Task-Control-Block

Weil jede Task durch ein Task-Status-Segment und einen Task-Control-Block identifiziert wird, hat ein Multitasking-System eine Vielzahl solcher Datenstrukturen zu verwalten. Sie sind ausschließlich für das Steuermodul des Betriebssystems von Interesse: So benutzt der **Scheduler** die **Task-Control-Blöcke** und der **Dispatcher** die **Task-Status-Segmente**. Dem Scheduler werden arbeitsbereite Tasks über die zugehörigen Task-Control-Blöcke bekannt gemacht. Hat er einen der anstehenden Task-Control-Blöcke ausgewählt, dann aktiviert er den Dispatcher, der daraufhin die neue Task startet.
Die einzelnen Aktionen sind im folgenden Bild 10 illustriert:

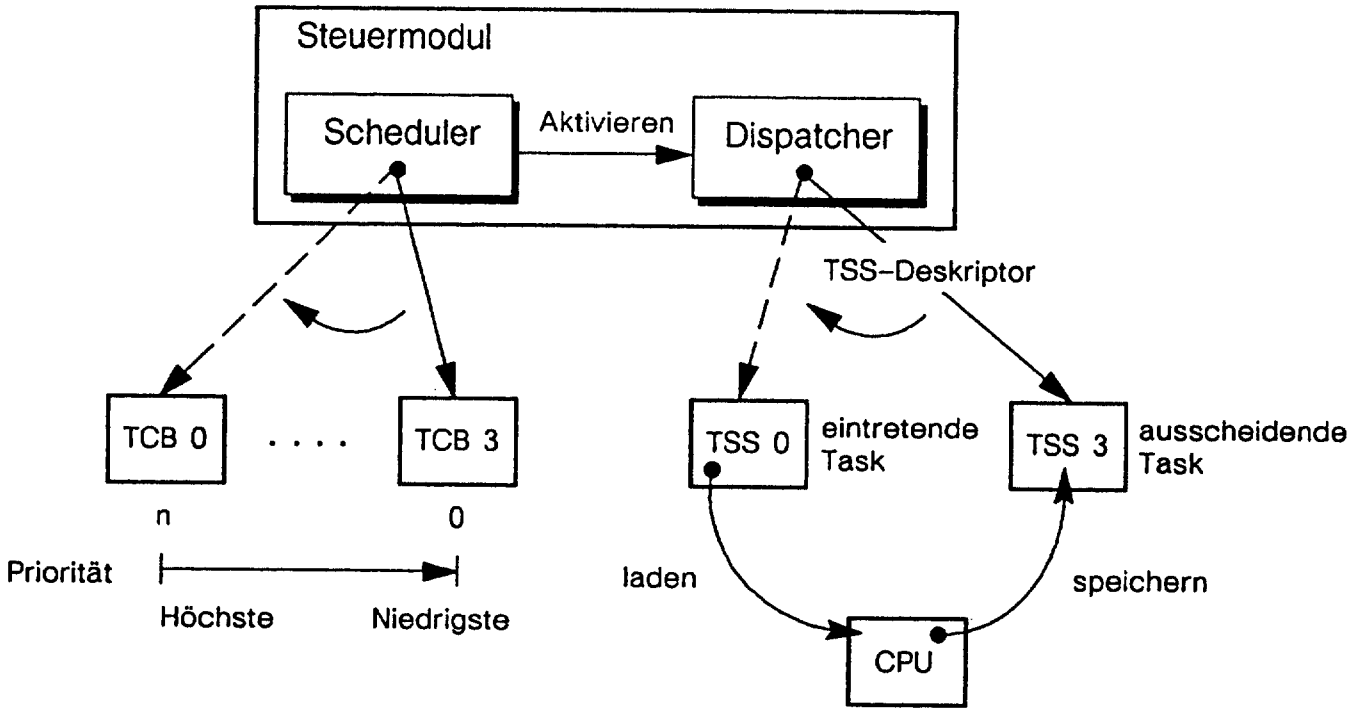


Bild 10: Scheduler- und Dispatcher-Aktionen

Nehmen wir an, die CPU führt im Augenblick die Task 3 auf der niedrigsten Prioritätsstufe 0 aus. Wenn in dieser Situation dem Scheduler der Task-Control-Block einer höherpriorisierten Task zugeführt wird, z. B. TCB 0 mit der höchsten Priorität n, dann muß ein Taskwechsel von Task 3 zur Task 0 erfolgen. Diesen Taskwechsel veranlaßt der Dispatcher, und er benutzt zu diesem Zweck den Task-Status-Segment-Deskriptor (TSS-Deskriptor) des Task-Status-Segments der neuen Task 0. Der Prozessor tritt daraufhin in Aktion: Er speichert den augenblicklichen CPU-Status (Register-Inhalte) im Task-Status-Segment TSS 3 der **ausscheidenden** Task 3 und lädt anschließend das Task-Status-Segment TSS 0 der **eintretenden** Task 0 in die CPU. Der Taskwechsel ist damit abgeschlossen, und die Task 0 beginnt ihre Arbeit.

1.3 Warteschlangen

Eines der wichtigsten Arbeitsmittel von Multitasking-Systemen sind Warteschlangen: So werden z. B. Task-Control-Blöcke von Tasks **gleicher** Priorität miteinander verkettet und als Warteschlange dem Scheduler zugeführt.
Eine Warteschlange, wir wollen sie ab jetzt **Queue** nennen, besteht aus einem Kopf, dem sogenannten **Queue-Header**, und der Anzahl verketteter Speicherelemente, die oft auch als **Poolelemente** bezeichnet werden.
Das Format des Queue-Headers zeigt das folgende Bild 11:

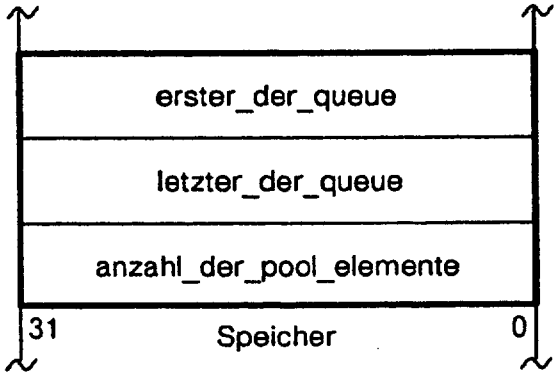


Bild 11: Queue-Header

Der Queue-Header enthält **zwei** Zeiger und **einen** Zähler, der die Anzahl der Poolelemente anzeigt.
Der Zeiger **erster_der_queue** zeigt auf das **erste** Poolelement der Queue, und der Zeiger **letzter_der_queue** auf das **letzte** Element. Wenn die Queue nur 1 Poolelement enthält, dann zeigen beide Zeiger auf den **Anfang** desselben Elements und der Zähler enthält den Wert 1. Dies ist im folgenden Bild 12 verdeutlicht:

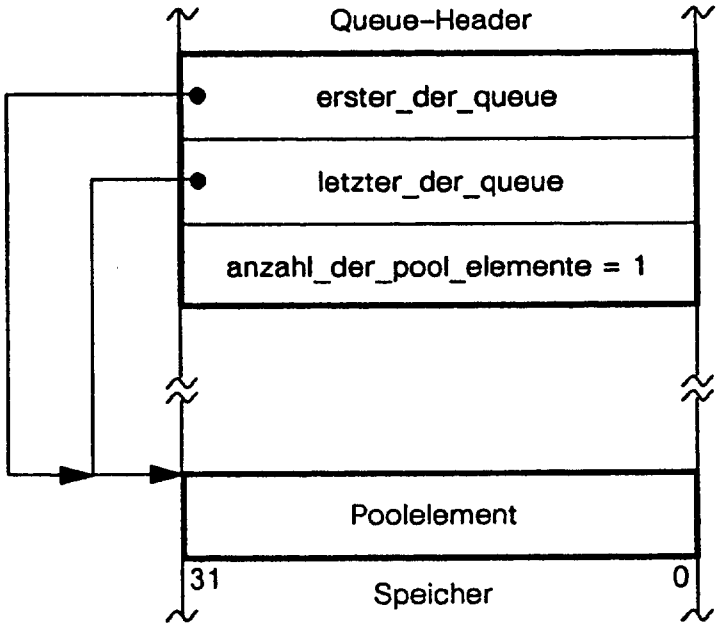


Bild 12: Queue mit 1 Poolelement

Wenn noch ein weiteres Element hinzukommt, dann ändert sich der Zeiger erster_der_queue **nicht**: Er zeigt weiterhin auf das **erste** Poolelement. Der Zeiger letzter_der_queue bekommt dagegen einen neuen Wert: Er zeigt jetzt auf das **neu** hinzugekommene Element, das den letzten Platz in der Queue einnimmt. Der Zähler hat den Wert 2. Siehe Bild 13:

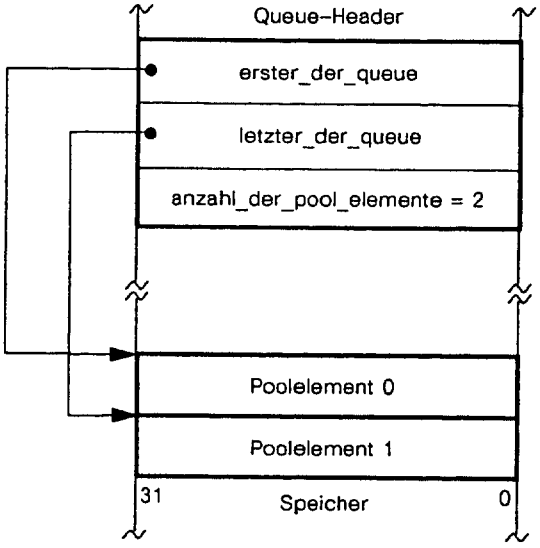


Bild 13: Queue mit zwei Poolelementen

Kommen noch weitere Poolelemente hinzu, dann stehen wir vor einem Dilemma, denn die beiden Zeiger können nur das erste und das letzte Poolelement bekannt machen. Es ist außerdem nicht festzustellen, in welcher Reihenfolge die weiteren Poolelemente angeordnet sind. Siehe Bild 14:

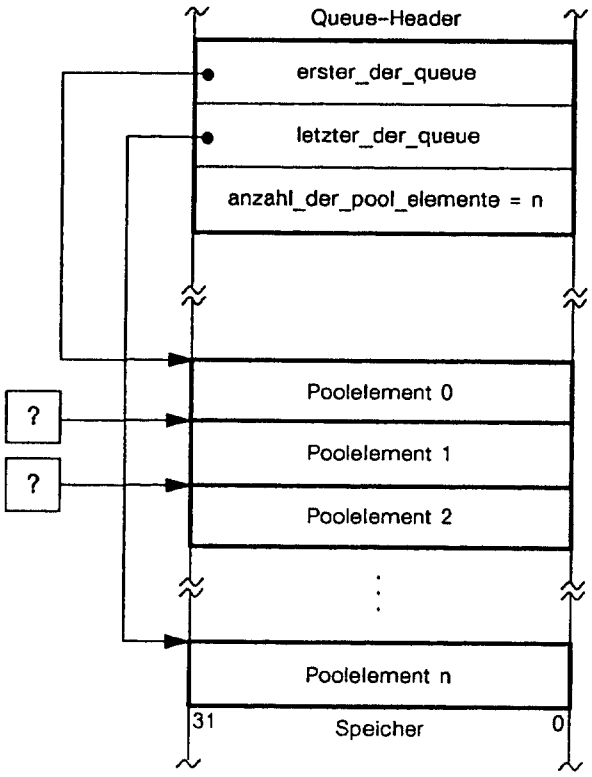


Bild 14: Queue mit n Poolelementen

Wie zu sehen ist, reicht der Queue-Header alleine nicht aus, um die Struktur einer Queue zu beschreiben. Das Problem ist aber gelöst, wenn wir die Poolelemente miteinander verketten: So brauchen wir in jedem Poolelement lediglich einen Zeiger, der auf seinen Nachfolger zeigt. Das Format eines solchen Poolelements zeigt folgendes Bild 15:

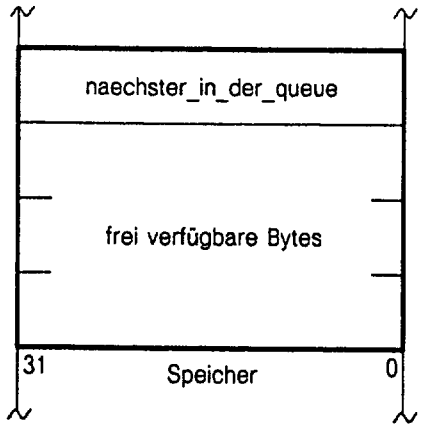


Bild 15: Vollständiges Poolelement

Es enthält den Zeiger **naechster_in_der_queue** und eine Anzahl von frei verfügbaren Bytes. Wenn wir später Poolelemente zur Definition Task-Control-Blöcken benutzen, dann werden wir in diese frei verfügbaren Bytes die Task-Nummer, den Task-Status und die Task-Priorität eintragen.
Eine vollständige Queue mit z. B. vier verketteten Poolelementen könnte jetzt folgendermaßen aussehen:

Siehe Bild 16 auf der nächsten Seite.

Das Bild zeigt, wie über den Zeiger naechster_in_der_queue die Reihenfolge der Poolelemente bestimmt wird. Weil das Poolelement 3 als letztes in der Queue **keinen** Nachfolger hat, ist dort der Zeiger naechster_in_der_queue mit **0** initialisiert.

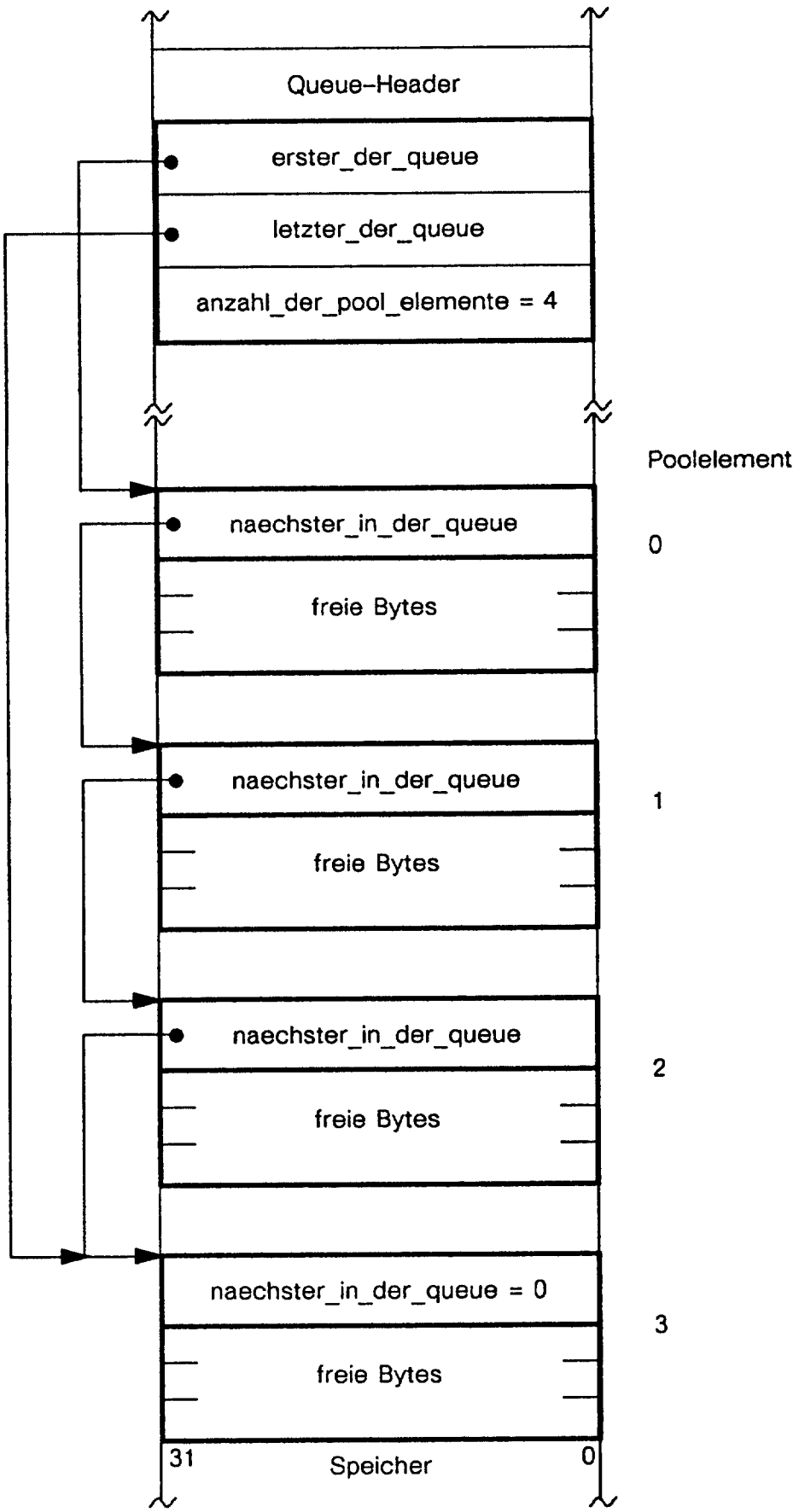


Bild 16: Verkettung von vier Poolelementen

1.4 Implementierung von Warteschlangen

Die Struktur einer Warteschlange, bestehend aus dem Queue-Header und einer Anzahl von Poolelementen, läßt sich in C folgendermaßen beschreiben:

```
typedef struct s3                                /* Queue-Header */
              {
                struct s4 *erster_der_queue;
                struct s4 *letzter_der_queue;
                short int anzahl_der_pool_elemente;
              } pool_kopf_struct;

typedef struct s4                                /* Poolelement */
              {
                struct s4 *naechster_in_der_queue;
                char betriebssystem_daten[10];
              } pool_element_struct;
```

Bild 17: Definition der Datentypen pool_kopf_struct und pool_element_struct (Datei struct1.h)

Hier sind zwei **neue** Datentypen vereinbart, nämlich **s3** und **s4**. Beide sind umbenannt, und zwar s3 in **pool_kopf_struct** und s4 in **pool_element_struct**. Eine Variable des Typs pool_kopf_struct repräsentiert die Struktur des **Queue-Headers**, und eine Variable des Typs pool_element_struct die Struktur eines **Poolelements**.
In einer Variablen des Typs pool_kopf_struct, also im Queue_Header zeigen die beiden Zeiger erster_der_queue und letzter_der_queue auf eine Variable des Typs pool_element_struct und damit auf ein Poolelement.
Die **short int**-Variable **anzahl_der_pool_elemente** im Datentyp pool_kopf_struct ist für den 32-Bit-Compiler iC386/486 eine **Zwei-Byte**-Variable und enthält die Anzahl der Poolelemente. Der Zeiger **naechster_in_der_queue** im Datentyp pool_element_struct zeigt auf eine Variable des gleichen Typs, also auf das nachfolgende Poolelement. Für die spätere Aufnahme von Betriebssystemdaten enthält dieser Datentyp außerdem noch ein **char**-array mit 10 Elementen.

1.4.1 Einketten und Ausketten von Poolelementen

Queues können nach verschiedenen Prinzipien «auf- und abgebaut» werden. Am weitesten verbreitet ist das Prinzip **FIFO** (First In First Out). Wie es funktioniert, zeigt das folgende Bild 18:

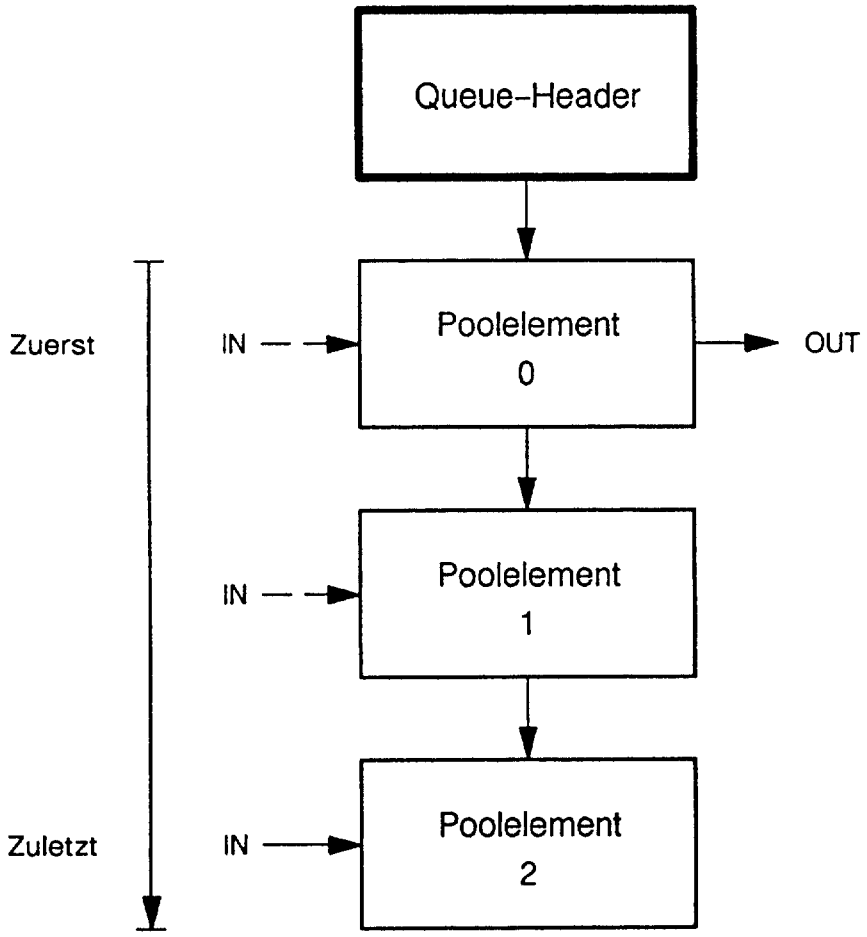


Bild 18: FIFO-Prinzip

Nach dem FIFO-Prinzip wird jedes **neue** Poolelement an das **Ende** der Queue eingekettet. So beginnt gemäß Bild 18 der Aufbau der Queue (IN) mit dem ersten Poolelement 0, danach folgt das Poolelement 1 und zuletzt das Poolelement 2. Je «näher» sich ein Poolelement am Queue-Header aufhält, umso länger verweilt es in der Queue. Aus diesem Grund beginnt der Abbau der Queue (OUT) mit dem Poolelement 0, danach folgt das Poolelement 1 und schließlich das Poolelement 2. Fazit: Dasjenige Poolelement, das **zuerst** in die Queue aufgenommen worden ist (First In), wird auch als **erstes** daraus entfernt (First Out).

1.4.1.1 Einketten nach FIFO

Das folgende C-Programm enthält die Betriebssystem-Funktion **einketten**, mit deren Hilfe ein neues Poolelement nach dem FIFO-Prinzip an das Ende der Queue eingehängt werden kann:

```
#define null (void *)    0
#include<struct1.h>

void einketten(pool_kopf_struct *pool_kopf_adr,
               pool_element_struct *neu_pool_element_adr)

{
 pool_element_struct *pool_element_adr;

 if (pool_kopf_adr -> anzahl_der_pool_elemente == 0)
    pool_kopf_adr -> erster_der_queue = neu_pool_element_adr;
 else {
       pool_element_adr   = pool_kopf_adr -> letzter_der_queue;
       pool_element_adr -> naechster_in_der_queue
       = neu_pool_element_adr;
      }

 pool_kopf_adr -> letzter_der_queue  = neu_pool_element_adr;
 pool_kopf_adr -> anzahl_der_pool_elemente ++;
 neu_pool_element_adr -> naechster_in_der_queue  = null;
}
```

Bild 19: Betriebssystem-Funktion **einketten** (Datei einkette.c)

Diese Funktion wird mit **zwei** Parametern aufgerufen. Der linke Parameter pool_kopf_adr zeigt auf eine Variable des Typs pool_kopf_struct und bekommt als aktuellen Parameter die Adresse des **Queue-Headers**.
Der rechte Parameter neu_pool_element_adr zeigt auf eine Variable des Typs pool_element_struct und bekommt als aktuellen Parameter die Adresse des **neuen Poolelements**.
Beide Datentypen sind in der Header-Datei struct1.h definiert (siehe Bild 17) und dem Programm mit #include «einverleibt». Die Funktion **einketten** beginnt mit der Anweisung

- `if(pool_kopf_adr -> anzahl_der_pool_elemente == 0)`

und überprüft damit, ob die Queue **leer** ist. Siehe Bild 20:

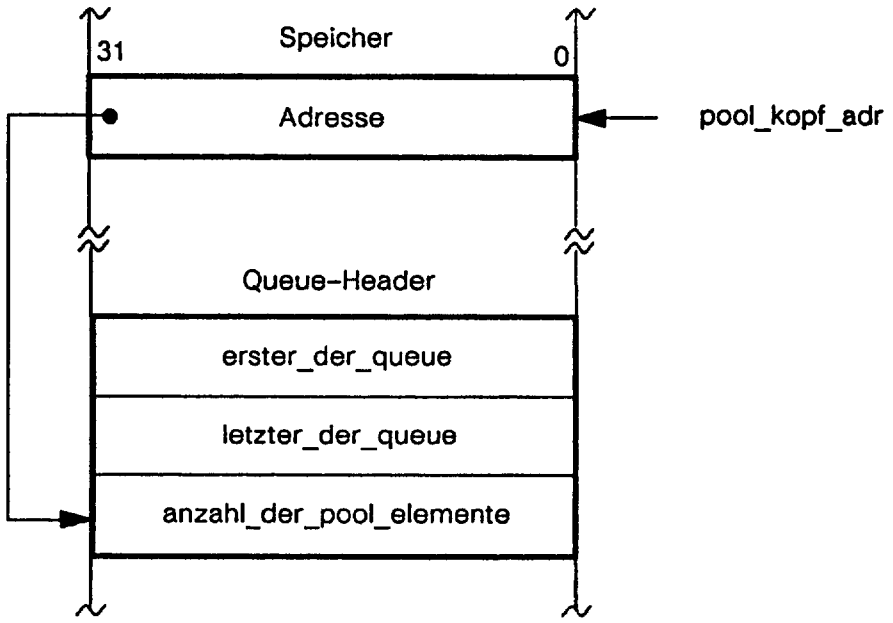


Bild 20: Funktion **einketten**: Queue leer ?

Für den Fall daß sie leer ist, speichert die Anweisung

- `pool_kopf_adr -> erster_der_queue = neu_pool_element_adr;`

die Adresse des **neuen** Poolelements in der Zeigervariablen erster_der_queue des Queue-Headers. Auf diese Weise wird das **erste** Poolelement in die Queue eingekettet. Siehe Bild 21:

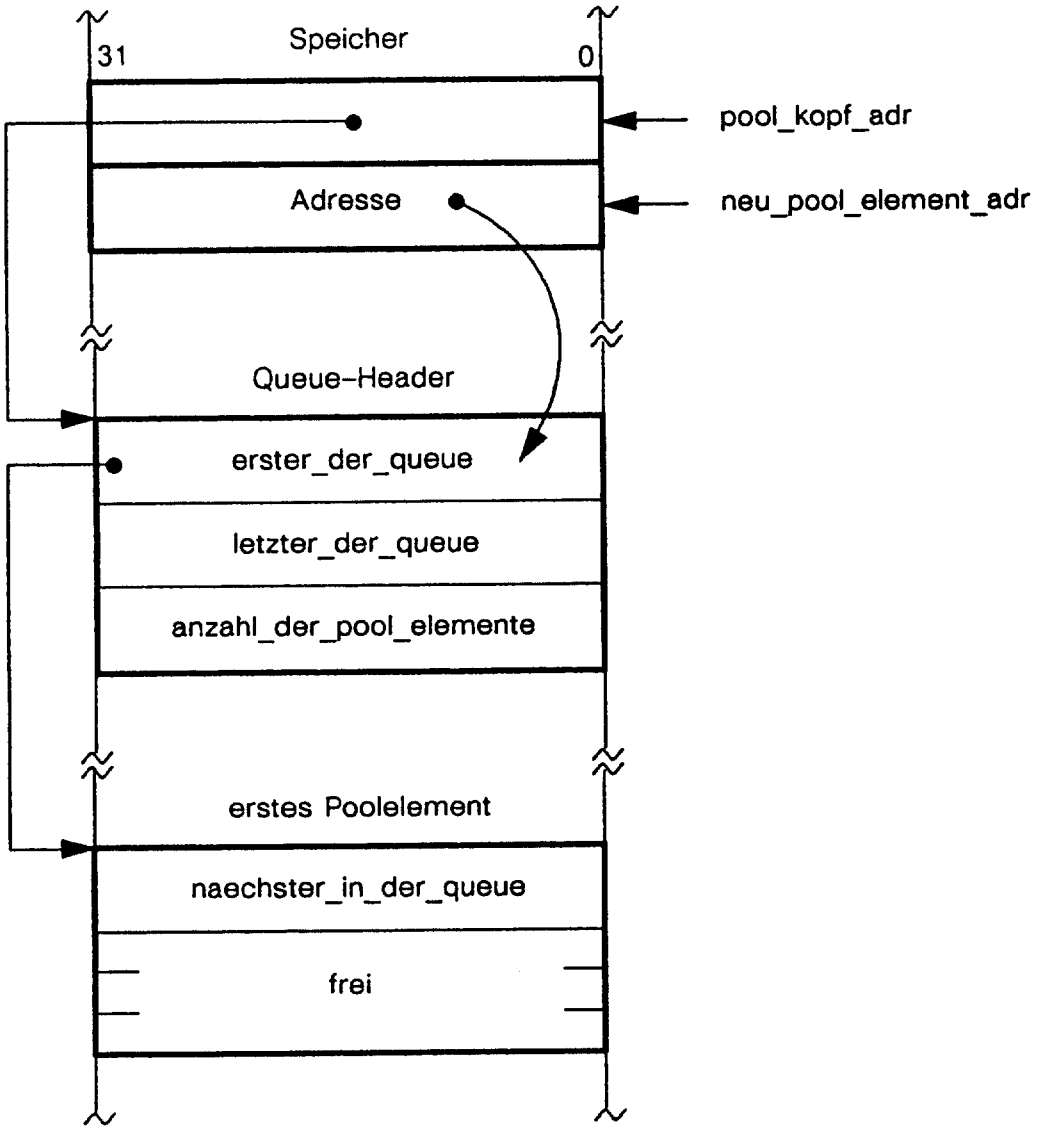


Bild 21: Funktion **einketten**: Erstes Poolelement einketten

Wenn die Funktion feststellt, daß die Queue **nicht** leer ist, dann enthalten die Zeigervariablen erster_der_queue und letzter_der_queue des Queue-Headers **aktuelle** Poolelement-Adressen. Ein neues Poolelement muß jetzt mit dem augenblicklich letzten Poolelement der Queue verbunden werden. Daher wird zunächst die Adresse des letzten Poolelements in der Zeigervariablen pool_element_adr zwischengespeichert. Dies geschieht durch die Anweisung

- `pool_element_adr = pool_kopf_adr -> letzter_der_queue;`

Wenn die Queue im Augenblick nur aus einem einzigen Poolelement besteht, dann zeigen danach **drei** Zeiger auf dieses Element. Siehe Bild 22:

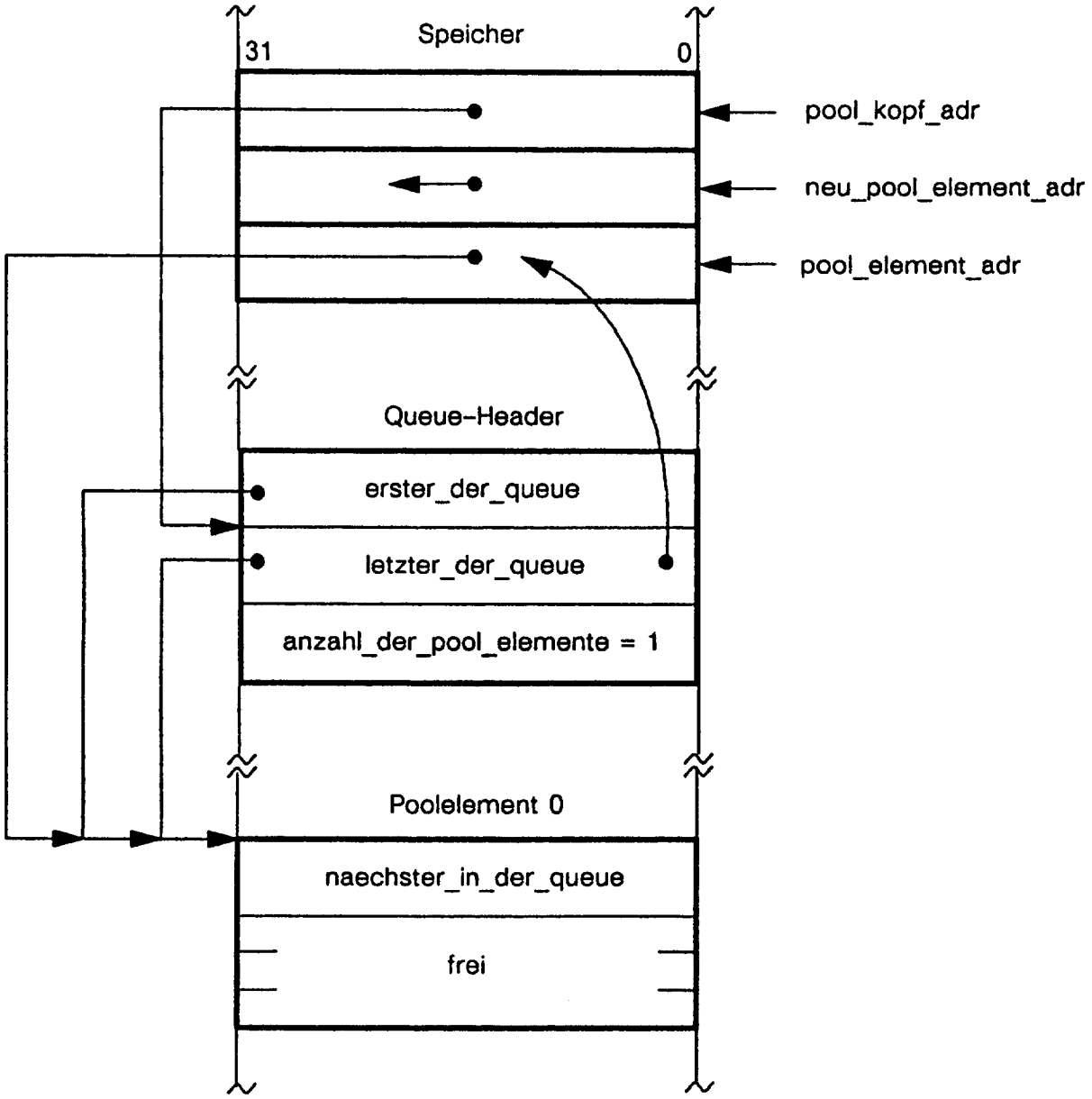


Bild 22: Funktion **einketten**: Adresse des letzten Poolelements zwischenspeichern

Mit der folgenden Anweisung

- `pool_element_adr -> naechster_in_der_queue`
  `= neu_pool_element_adr;`

wird jetzt das **neue** Poolelement an das Ende der Queue angehängt. Die Adresse dieses Elements wurde an die Funktion **einketten** übergeben und steht in der Zeigervariablen **neu_pool_element_adr**. Die obige Anweisung speichert diese Adresse in der Zeigervariablen **naechster_in_der_queue** des bisher letzten Poolelements und kettet so das neue Poolelement in die Queue ein. Siehe Bild 23:

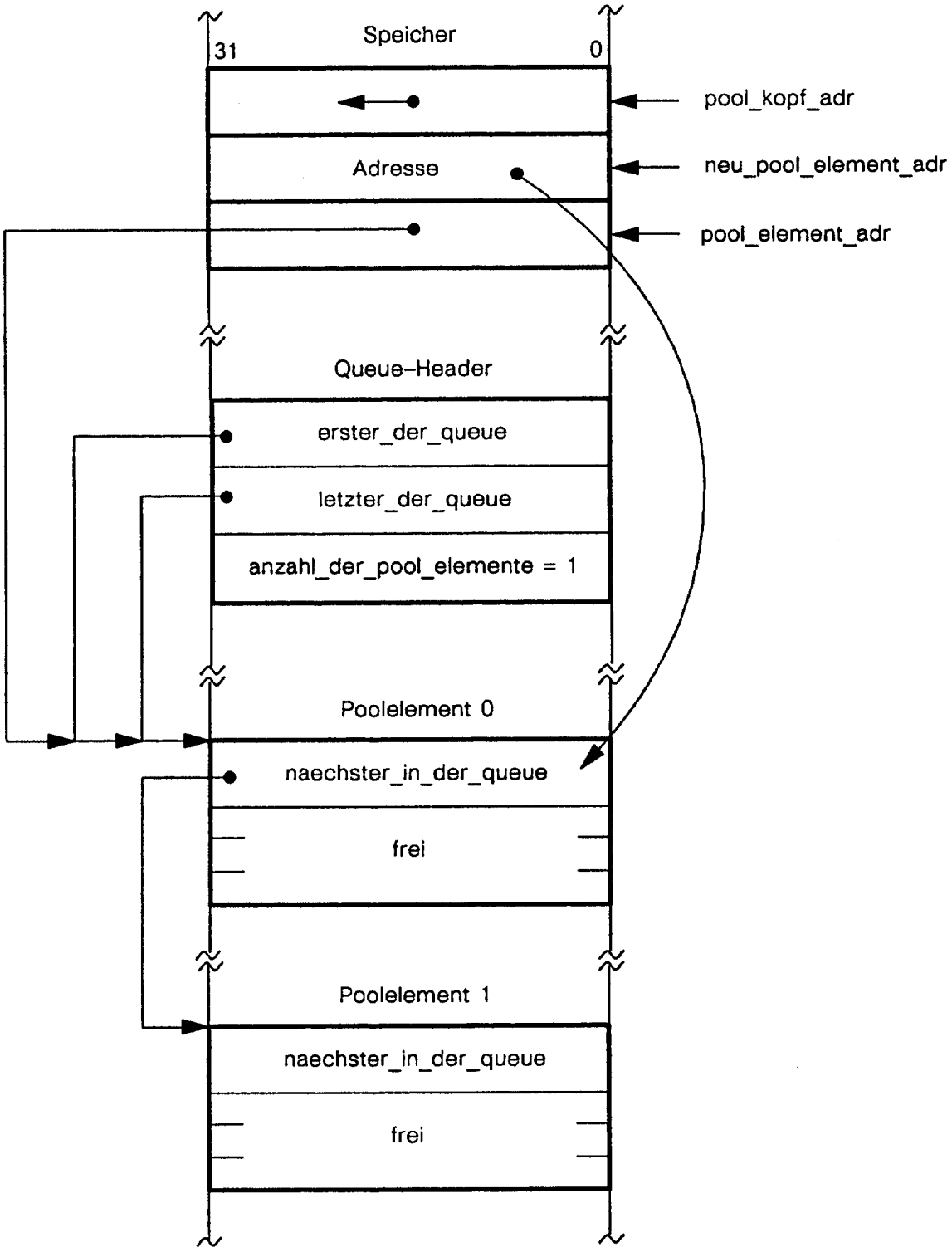


Bild 23: Funktion **einketten**: Einketten eines neuen Poolelements

Unabhängig davon, ob die Queue leer ist oder nicht, sorgt die Funktion **einketten** mit der Anweisung

- `pool_kopf_adr -> letzter_der_queue = neu_pool_element_adr;`

dafür, daß im Queue-Header der Zeiger **letzter_der_queue** auf das letzte Poolelement zeigt.

Nehmen wir zur Verdeutlichung unsere vorige Situation. Die Queue besteht dort aus zwei Poolelementen, aber der Zeiger **letzter_der_queue** zeigt noch immer auf das **erste** Poolelement. Die obige Anweisung überträgt nun die Adresse des neuen Poolelements von der Zeigervariablen **neu_pool_element_adr** in die Zeigervariable **letzter_der_queue** im Queue-Header und stellt damit die Verbindung zwischen dem Queue-Header und dem zuletzt eingehängten Poolelement her. Siehe Bild 24:

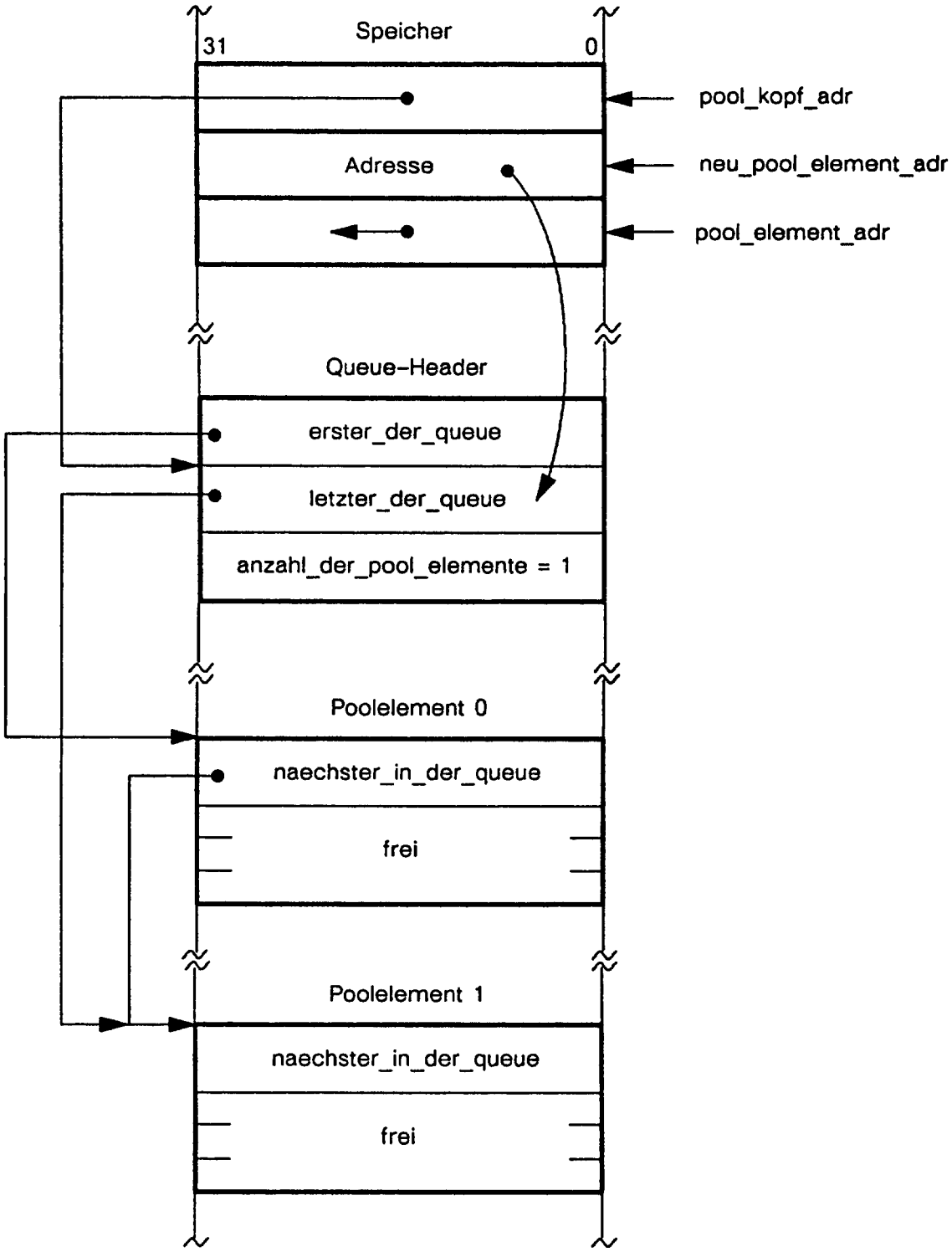


Bild 24: Funktion **einketten**: Verbindung zwischen Queue-Header und letztem Element herstellen

Was bleibt noch zu tun? Wird ein Poolelement in die Queue aufgenommen, dann muß der Poolelemente-Zähler inkrementiert werden. Dies erledigt die Anweisung

- pool_kopf_adr -> anzahl_der.pool_elemente ++;

Weil jedes neu hinzugekommene Poolelement das letzte der Queue sein könnte, wird seine Zeigervariable **naechster_in_der_queue** mit **0** initialisiert. Damit wird angezeigt, daß das betreffende Element **keinen** Nachfolger hat:

- neu_pool_element_adr -> naechster_in_der_queue = Null;

Die Aktionen der beiden Anweisungen illustriert Bild 25:

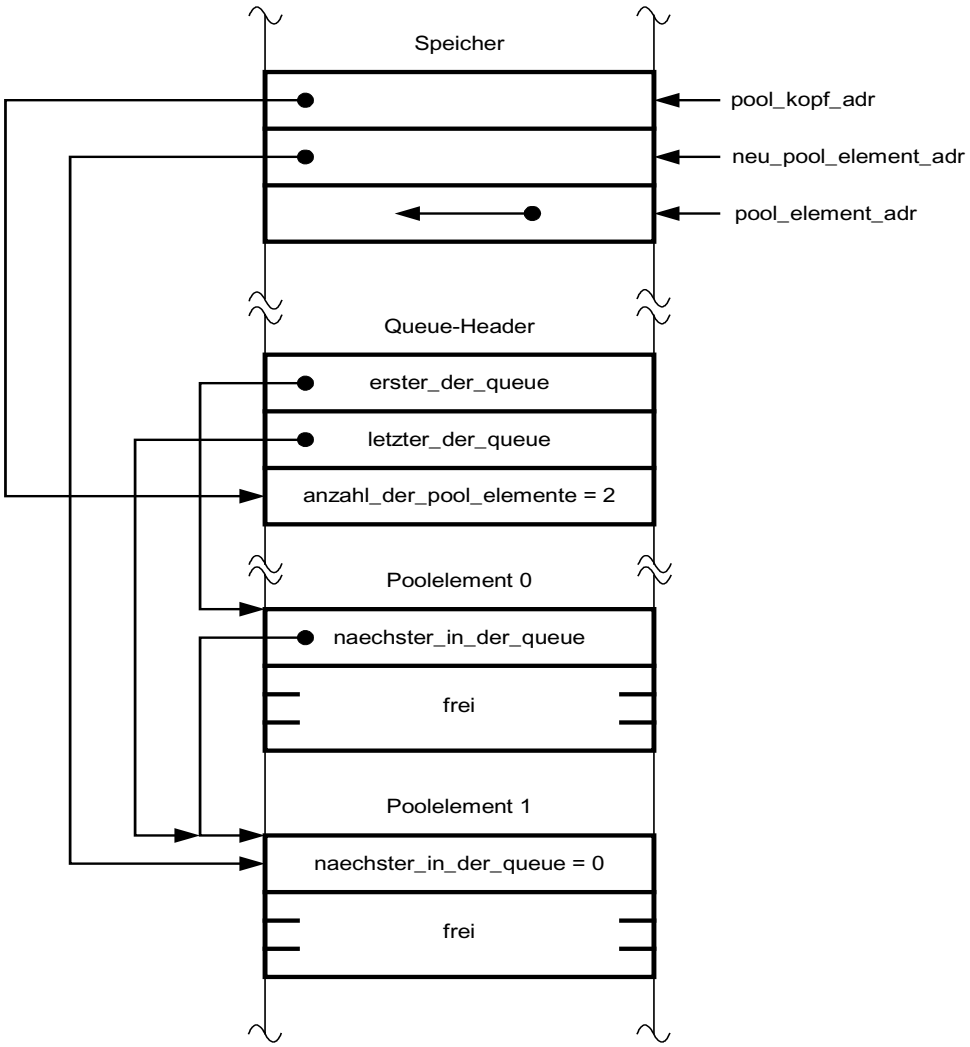


Bild 25: Funktion **einketten**: Anzahl der Poolelemente aktualisieren und Ende der Queue kennzeichnen

Die Funktion einketten ist nun beendet. Sie hat eine Queue aufgebaut, wie sie abschließend im folgenden Bild 26 dargestellt ist:

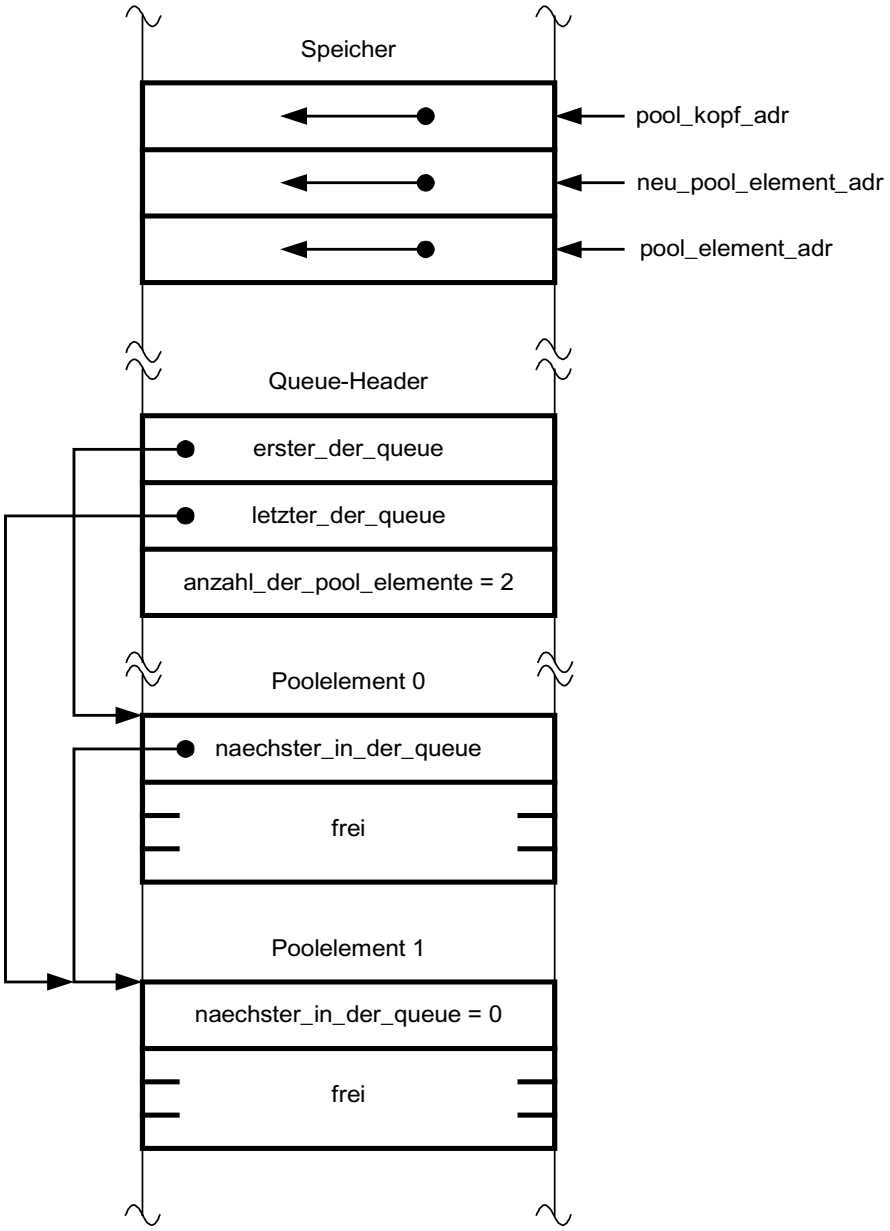


Bild 26: Funktion **einketten:** Resultierende Queue

1.4.1.2 Ausketten nach FIFO

Das folgende C-Programm enthält die Betriebssystem-Funktion **ausketten.,** mit deren Hilfe ein Poolelement nach dem FIFO-Prinzip am Anfang der Queue ausgekettet werden kann:

```
#define null (void *)    0
#include<struct1.h>

void ausketten(pool_kopf_struct *pool_kopf_adr)
{
 pool_element_struct *pool_element_adr;

 if (pool_kopf_adr -> anzahl_der_pool_elemente != 0)
 {
   pool_element_adr = pool_kopf_adr -> erster_der_queue;
   pool_kopf_adr -> erster_der_queue = pool_element_adr
                                     -> naechster_in_der_queue;
   pool_kopf_adr -> anzahl_der_pool_elemente --;

   if (pool_kopf_adr -> anzahl_der_pool_elemente == 0)
       {
        pool_kopf_adr -> erster_der_queue  = null;
        pool_kopf_adr -> letzter_der_queue = null;
       }
 }
}
```

Bild 27: Betriebssystem-Funktion **ausketten** (Datei auskette.c)

Wie zu erkennen ist, wird die Funktion **ausketten** mit einem **einzigen** Parameter aufgerufen, nämlich mit der Adresse des **Queue_Headers.** Denn genauso wie bei der Funktion **einketten**, so ist auch hier der Parameter pool_kopf_adr ein Zeiger, der auf eine Variable des Typs pool_kopf_struct zeigt.

Die Funktion beginnt mit der Anweisung

- `if(pool_kopf_adr -> anzahl_der_pool_elemente != 0)`

und überprüft damit, ob eine Queue existiert. Siehe Bild 27.1:

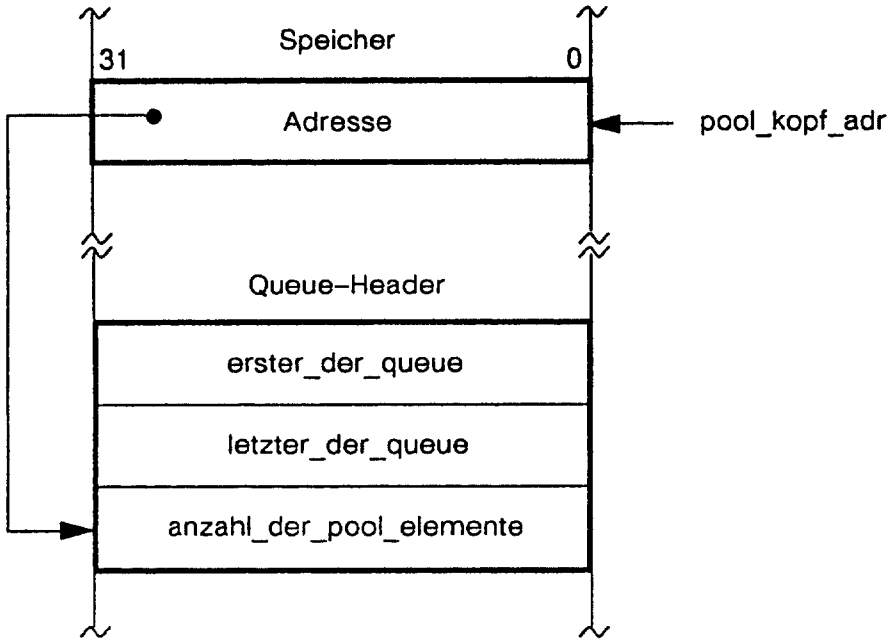


Bild 27.1: Funktion **ausketten**: Queue vorhanden ?

Wenn die Queue **leer** ist, dann beendet die Funktion sofort ihre Arbeit, und es erfolgt die Rückkehr zum Aufrufer. Im anderen Fall holt sich die Funktion mit der Anweisung

- `pool_element_adr = pool_kopf_adr -> erster_der_queue;`

vom Queue-Header die Adresse des ersten Poolelements und speichert sie in der Zeigervariablen **pool_element_adr**. Siehe Bild 28:

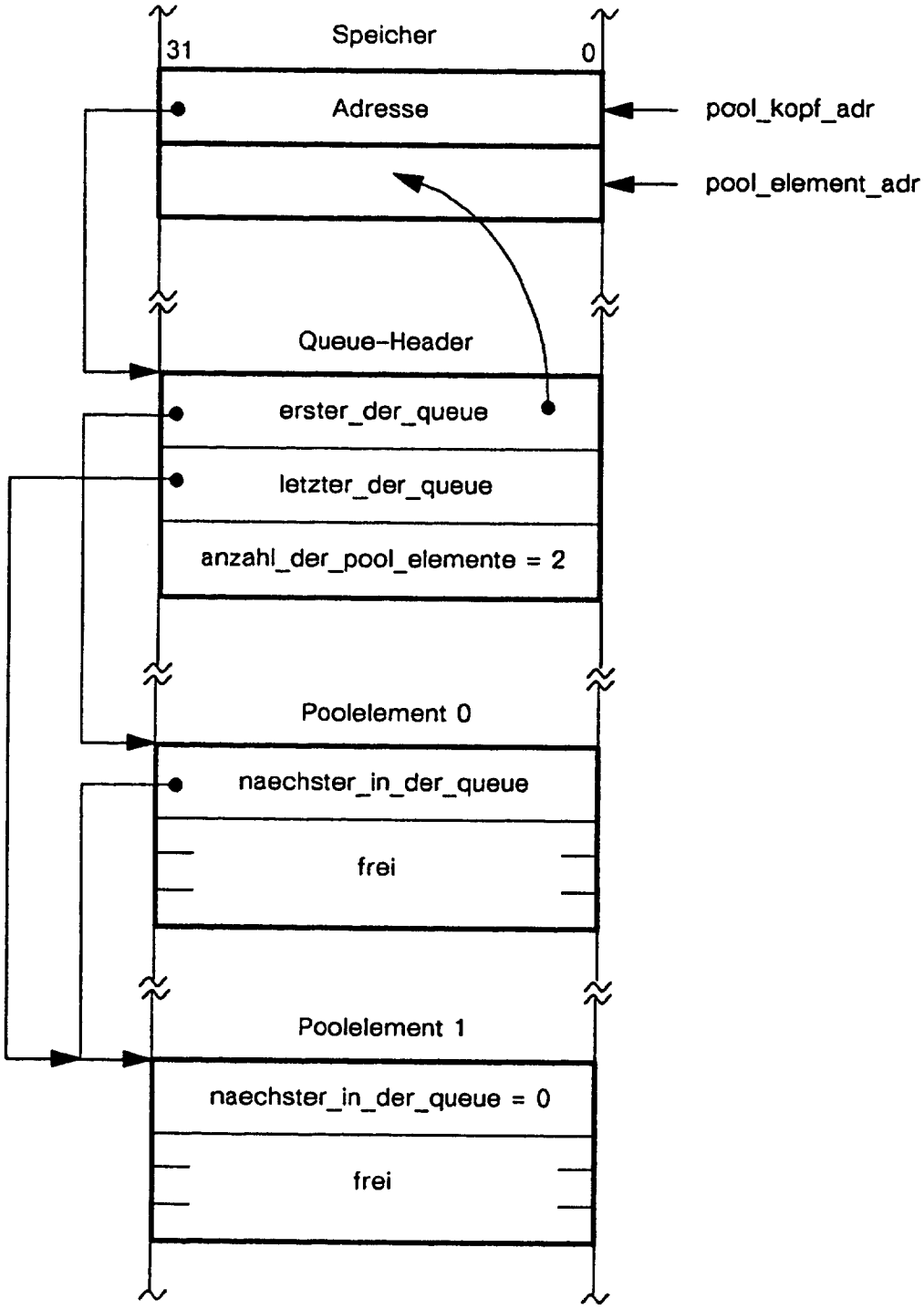


Bild 28: Funktion **ausketten**: Adresse des Poolelements 0 umspeichern

Danach holt die Anweisung

- `pool_kopf_adr -> erster_der_queue = pool_element_adr`
  `-> naechster_in_der_queue;`

vom **ersten** Poolelement der Queue aus der Zeigervariablen **naechster_in_der_queue** die Adresse seines Nachfolgers und speichert sie in der Zeigervariablen **erster_der_queue** des Queue-Headers. Siehe Bild 29:

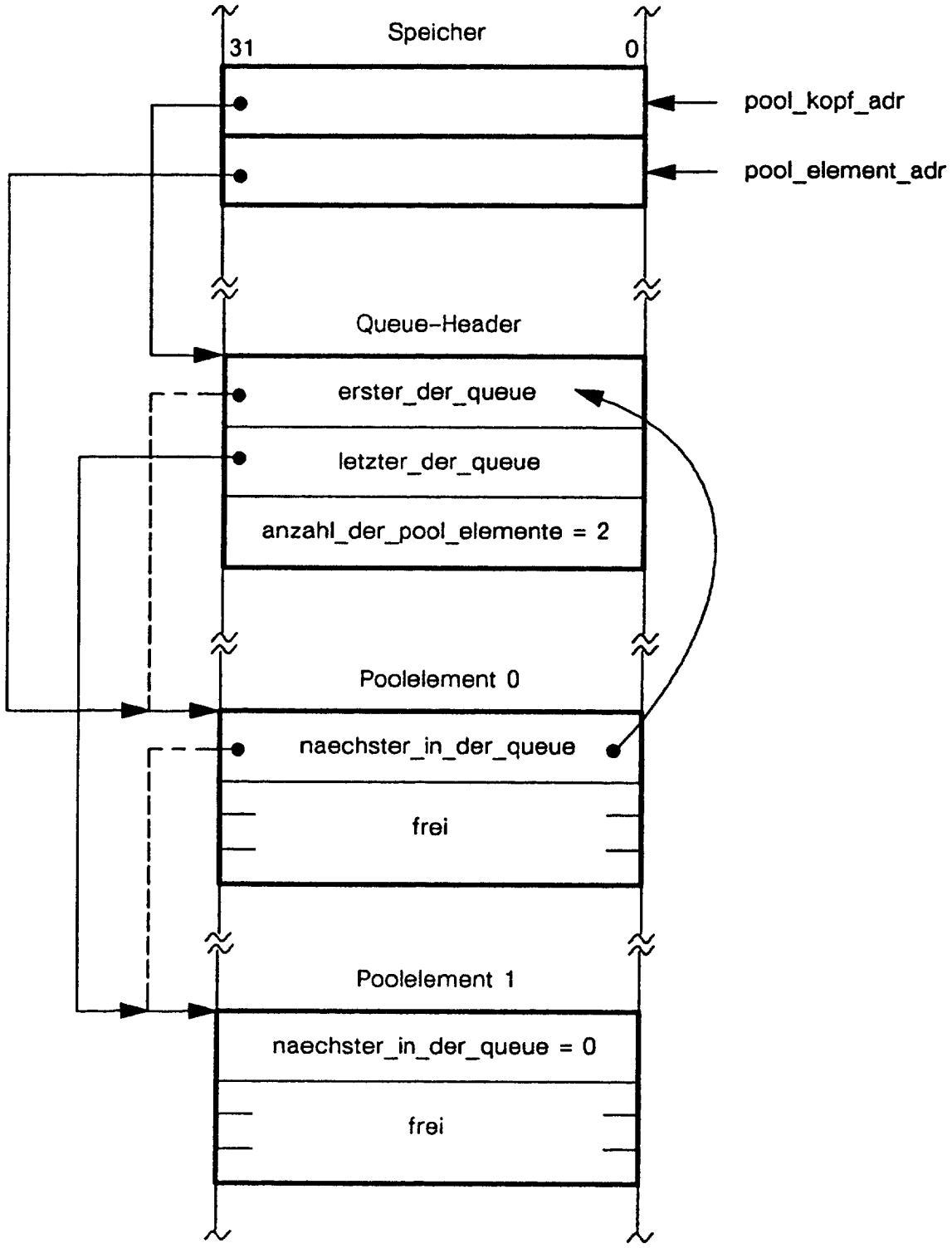


Bild 29: Funktion **ausketten**: Adresse des Nachfolgers im Queue-Header speichern

Weil im Queue-Header jetzt **erster_der_queue** auf das Poolelement 1 zeigt, ist das Poolelement 0 aus der Queue «entfernt». Die Funktion **ausketten** muß daher den Poolelemente-Zähler **dekrementieren**, und sie erledigt dies mit der Anweisung

- `pool_kopf_adr -> anzahl_der_pool_elemente --;`

Siehe Bild 30:

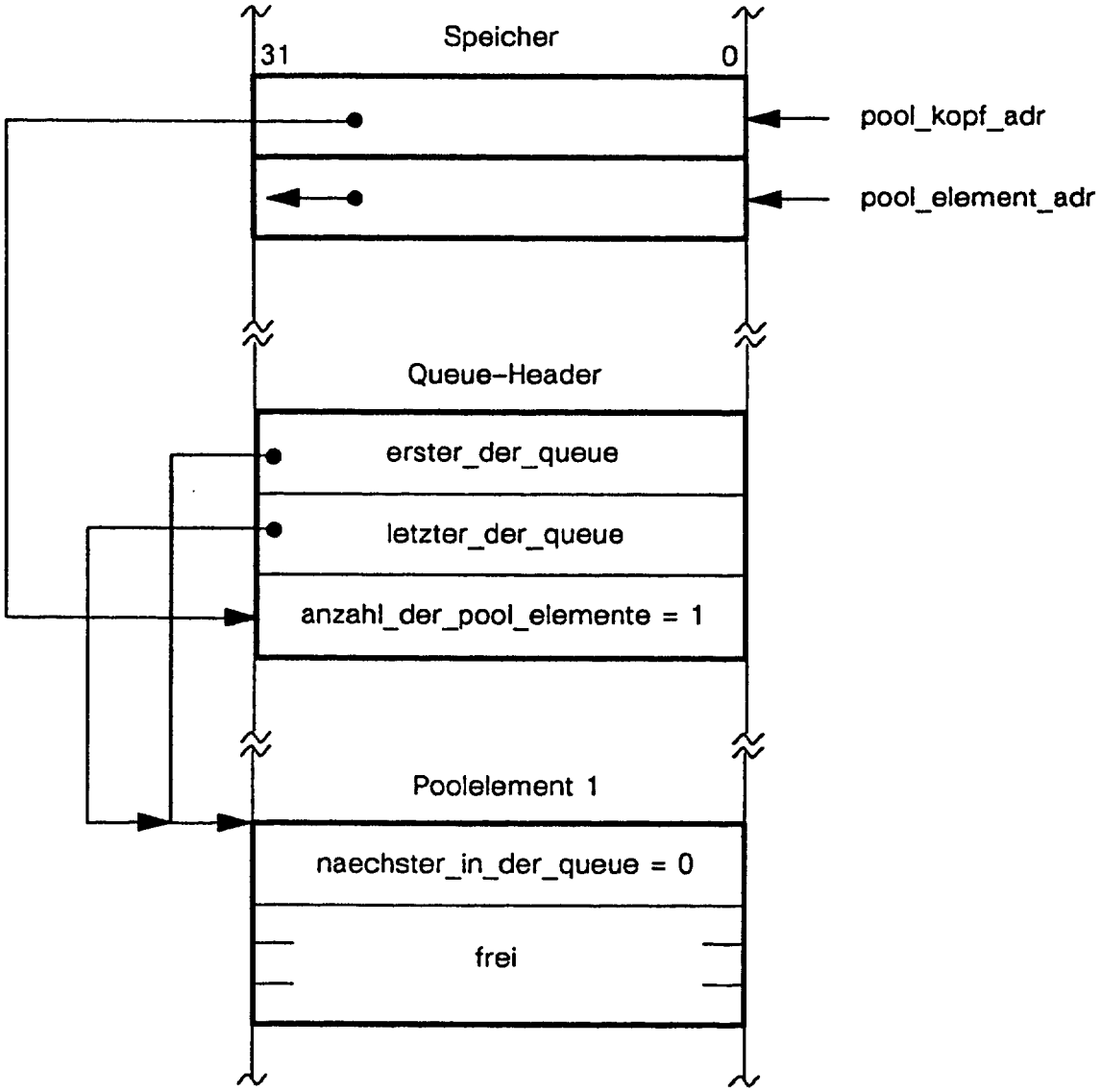


Bild 30: Funktion **ausketten**: Anzahl der Poolelemente aktualisieren

Anschließend prüft sie mit der Anweisung

- `if(pool_kopf_adr -> anzahl_der_poolelemente == 0)`

ob der Zähler den Wert **0** erreicht hat. Wenn ja, dann ist die Queue **leer**, und die Funktion führt daraufhin die Anweisungen

- `pool_kopf_adr -> erster_der_queue  = null;`
- `pool_kopf_adr -> letzter_der_queue = null;`

aus. Alle Felder des Queue-Headers sind jetzt mit **0** initialisiert und damit die Queue als leer gekennzeichnet. Siehe Bild 31:

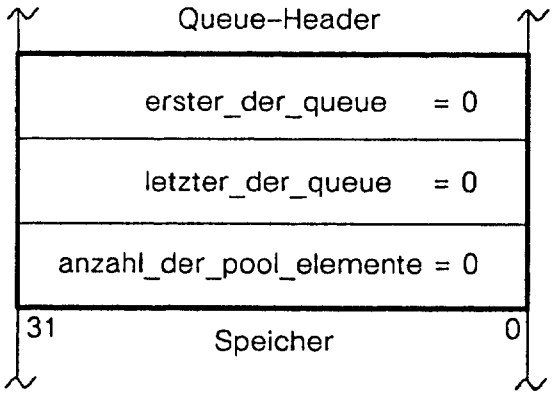


Bild 31: Als leer gekennzeichnete Queue

1.5 Implementierung von Task-Queues

Jedes x86/Pentium-Betriebssystem besteht aus einer Vielzahl von Tasks, die nach bestimmten Entscheidungskriterien dem Prozessor zugeführt werden.
Die Auswahl der Tasks trifft der Scheduler und er benutzt dabei die **Task-Control-Blöcke** (TCB). Ein Kriterium zur Auswahl einer Task ist ihre **Priorität**. So ist es möglich, jeder Task über ihren Task-Control-Block eine individuelle Priorität zu geben. Oft kommt es aber vor, daß eine Anzahl von Tasks **gleiche** Priorität haben. In einem solchen Fall müssen dann die Task-Control-Blöcke als sogenannte **Task-Queues** an den betreffenden Prioritätsstufen installiert werden.
So können abhängig von der Anzahl der Prioritätsstufen, die das Betriebssystem bereitstellt, die unterschiedlichsten Task-Queue-Konfigurationen entstehen. Das folgende Bild 32 zeigt hierfür ein Beispiel:

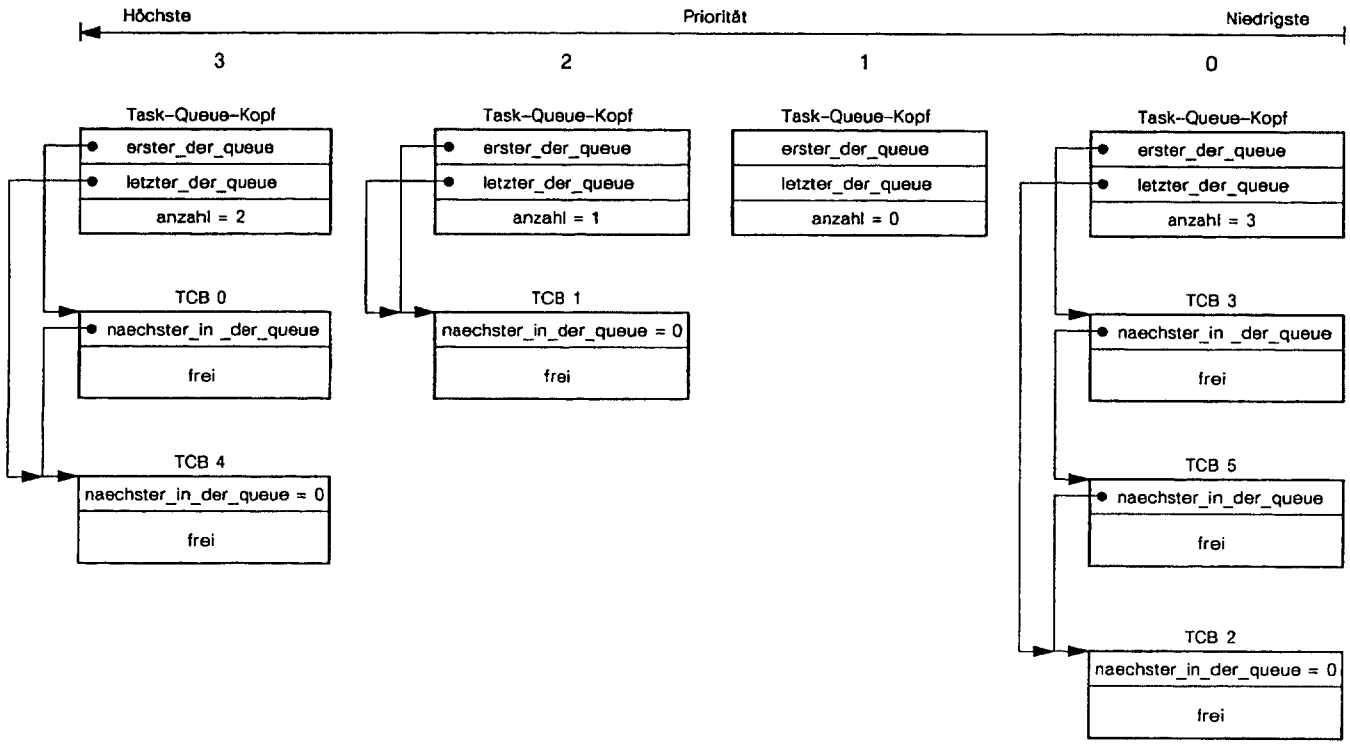


Bild 32: Task-Queue-Konfiguration

Wie zu sehen ist, stellt hier das Betriebssystem **vier** Prioritätsstufen im Bereich von 0 bis 3 zur Verfügung. Die Task-Queue

- auf der Prioritätsstufe 0 enthält drei Task-Control-Blöcke,
- die auf der Prioritätsstufe 1 ist leer,
- die auf der Prioritätsstufe 2 enthält einen Task-Control-Block und
- die auf der Prioritätsstufe 3 enthält zwei.

Wie kann nun eine solche Task-Queue-Konfiguration realisiert werden ? Jede Task-Queue besteht aus einem **Task-Queue-Kopf** und einer Anzahl von **Task-Control-Blöcken**. Ihre Struktur kann daher durch zwei **neue** Datentypen wie folgt beschrieben werden:

```
typedef struct s1
              {
                struct s2 *erster_der_queue;
                struct s2 *letzter_der_queue;
                short int anzahl_der_task_cbe;
              } task_queue_kopf_struct;

typedef struct s2
              {
                struct s2 *naechster_in_der_queue;
                char task_nummer;
                char task_status;
                char task_prioritaet;
              } task_cb_struct;
```

Bild 33: Definition der Datentypen task_queue_kopf_struct und task_cb_struct (Datei struct.h)

Die **neuen** Datentypen heißen **s1** und **s2**. Beide sind umbenannt, und zwar s1 in task_queue_kopf_struct und s2 in task_cb_struct. Eine Variable des Typs task_queue_kopf_struct repräsentiert einen **Task-Queue-Kopf** und eine Variable des Typs task_cb_struct repräsentiert einen **Task-Control-Block**. So zeigen die beiden Zeiger **erster_der_queue** und **letzter_der_queue** auf einen Task-Control-Block. Die **short int**-Variable **anzahl_der_task_cbe** in task_queue_kopf_struct zählt die Anzahl der Task-Control-Blöcke, und der Zeiger **naechster_in_der_queue** in einer Variablen des Typs task_cb_struct zeigt auf eine Variable des gleichen Typs und damit auf den nachfolgenden Task-Control-Block.
In task_cb_struct sind außerdem die **char**-Variablen **task_nummer**, **task_status** und **task_prioritaet** definiert. In einer Variablen dieses Typs (Task-Control-Block) trägt hier das Betriebssystem vom Anwender vergebene Nummern ein und charakterisiert so die betreffende Task. Wenn das Betriebssystem die char-Variable task_status mit **ready** initialisiert (z. B. ready = 1) und den Task-Control-Block bei einer bestimmten Prioritätsstufe (task_prioritaet = 0..3) in die Task-Queue einkettet, dann ist die zugehörige Task dem System bekannt gemacht, und sie kann vom Scheduler/Dispatcher-Gespann zur Ausführung gebracht werden.
Damit dies geschehen kann, muß das Betriebssystem eine Anzahl von Prioritätsstufen zur Verfügung stellen. So installiert unmittelbar nach dem Systemstart die Betriebssytem-Funktion **init** für jede Priorität einen **Task-Queue-Kopf**, wobei jeder dieser Köpfe eine Prioritätsstufe darstellt. Siehe Bild 34:

```
#define null (void*)        0
#define task_max            4
#define prioritaet_max      4
#include<struct.h>

task_queue_kopf_struct      task_queue_kopf[prioritaet_max];
task_cb_struct              task_cb[task_max];

char i;

void init(void)     /* Vier Prioritätsstufen installieren */
{
 for (i=0;i<prioritaet_max;i++)
     {
      task_queue_kopf[i].erster_der_queue    = null;
      task_queue_kopf[i].letzter_der_queue   = null;
      task_queue_kopf[i].anzahl_der_task_cbe = 0;
     }
}
```

Bild 34: Betriebssystem-Funktion init (Datei init.c)

Die Funktion benutzt die Datentypen task_queue_kopf_struct und task_cb_struct. Sie sind in der Header-Datei struct.h definiert (siehe Bild 33) und dem C-Programm mit #include «einverleibt». Im Programm sind zwei globale arrays vereinbart, und zwar

- `task_queue_kopf[prioritaet_max]`, dessen Elemente die Struktur des Task-Queue-Kopfs haben und
- `task_cb[task_max]`, dessen Elemente die Struktur des Task-Control-Blocks haben.

Betrachten wir zunächst das array der Task-Queue-Köpfe. In einer **for**-Schleife, deren Laufvariable i von **0 = i < prioritaet_max** läuft, werden **vier** Task-Queue-Köpfe mit den Prioritäten 0 bis 3 installiert und ihre Inhalte mit **0** initialisiert.

So stehen am Ende **vier** Prioritätsstufen in Form von vier **leeren** Queues zur Verfügung. Siehe Bild 35:

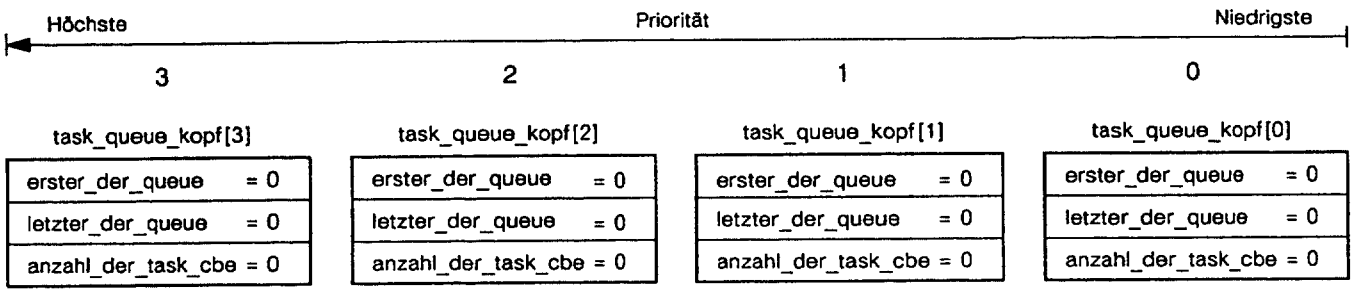


Bild 35: Bereitstellung von vier Prioritätsstufen

Ab jetzt können beliebig viele Task-Control-Blöcke bei der gewünschten Priorität in die Task-Queues eingekettet werden. Diese Arbeit erledigt die Betriebsystem-Funktion **task_init**, die im folgenden C-Programm enthalten ist:

```
#define task_max 4
#define prioritaet_max 4
#include<struct.h>

extern task_queue_kopf_struct task_queue_kopf[prioritaet_max];
extern task_cb_struct task_cb[task_max];

void task_init(char task_id,char task_status,char task_prioritaet)
{
 task_queue_kopf_struct *task_queue_kopf_adr;
 task_cb_struct *neu_task_cb_adr;

 task_cb[task_id].task_nummer     = task_id;
 task_cb[task_id].task_status     = task_status;
 task_cb[task_id].task_prioritaet = task_prioritaet;

 task_queue_kopf_adr =
              &task_queue_kopf[task_cb[task_id].task_prioritaet];
 neu_task_cb_adr = &task_cb[task_id];
 einketten(task_queue_kopf_adr,neu_task_cb_adr);
}
```

Bild 36: Betriebssystem-Funktion task_init (Datei task_ini.c)

Jede Task bekommt zur Identifizierung ihre individuelle Nummer, einen bestimmten Anfangsstatus und eine Priorität. Daher wird die Funktion task_init mit diesen drei Parametern aufgerufen. Dabei bekommt

- der char-Parameter **task_id** die Nummer der Task,
- der char-Parameter **task_status** den Anfangsstatus ready = 1 und
- der char-Parameter **task_prioritaet** die gewünschte Priorität 0..3.

Die Funktion benutzt die arrays task_queue_kopf[prioritaet_max] und task_cb[task_max], die beide **extern** im C-Modul init definiert sind (siehe Bild 34).
Wegen **task_max = 4**, initialisiert task_init ein array von **vier** Task-Control-Blöcken und kettet sie in die Queue ein. Die Initialisierung dieser Blöcke übernehmen die Anweisungen

- `task_cb[task_id].task_nummer = task_id;`
- `task_cb[task_id].task_status = task_status;`
- `task_cb[task_id].task_prioritaet = task_prioritaet;`

Soll das Multitasking-System aus maximal vier Tasks bestehen, dann liefern diese Anweisungen, abhängig von den übergebenen Parametern, z. B. das folgende array von Task-Control-Blöcken:

task_cb[3]	task_cb[2]	task_cb[1]	task_cb[0]
naechster_in_der_queue	naechster_in_der_queue	naechster_in_der_queue	naechster_in_der_queue
task_nummer = 3	task_nummer = 2	task_nummer = 1	task_nummer = 0
task_status = 1	task_status = 1	task_status = 1	task_status = 1
task_prioritaet = 3	task_prioritaet = 3	task_prioritaet = 0	task_prioritaet = 0
Task 3	Task 2	Task 1	Task 0

Bild 37: Array mit vier Task-Control-Blöcken

Wie zu sehen ist, identifiziert jeder Task-Control-Block **seine** Task durch die **task_nummer** (0..3), den **task_status** (ready = 1) und die **task_prioritaet** (0..3). Hier haben Task 0, Task 1 die Priorität 0 und Task 2, Task 3 die Priorität 3.
Die Funktion task_init sorgt nun dafür, daß jeder dieser Task-Control-Blöcke bei der spezifizierten Priorität eingekettet wird. Zu diesem Zweck ruft sie die Betriebssystem-Funktion **einketten** auf und übergibt ihr als Parameter zwei Adressen:

Der linke Parameter task_queue_kopf_adr ist vom Typ task_queue_kopf_struct und bekommt wegen

- `task_queue_kopf_adr = &task_queue_kopf[task_cb[task_id].task_prioritaet];`

die **Adresse** eines **Task-Queue-Kopfes**.
Der rechte Parameter neu_task_cb_adr ist vom Typ task_cb_struct und bekommt wegen

- `neu_task_cb_adr = &task_cb[task_id];`

die **Adresse** des einzukettenden **Task-Control-Blocks**.
Wie diese Adressen ermittelt werden, illustriert das folgende Bild 38 an einem Beispiel:

Siehe Bild 38 auf der nächsten Seite.

Es sei **task_nummer = 2**; dann soll wegen **task_prioritaet = 3** (siehe Bild 37) der Task-Control-Block 2 bei der Prioritätsstufe 3 bzw. beim Task-Queue-Kopf 3 in die Queue eingehängt werden.
Aus task_cb[2].task_prioritaet = 3 (1) ergibt sich die Adresse des Task-Queue-Kopfs 3 zu &task_queue_kopf[3].
Diese Adresse wird in der Zeigervariablen task_queue_kopf_adr gespeichert (2).
Die Adresse des Task-Control-Blocks 2 ergibt sich wegen task_nummer = 2 zu &task_cb[2], und sie wird in der Zeigervariablen neu_task_cb_adr gespeichert (3) Für die Funktion **einketten** stehen jetzt alle aktuellen Parameter bereit; sie kann mit

- `einketten(task_queue_kopf_adr,neu_task_cb_adr);`

aufgerufen werden.

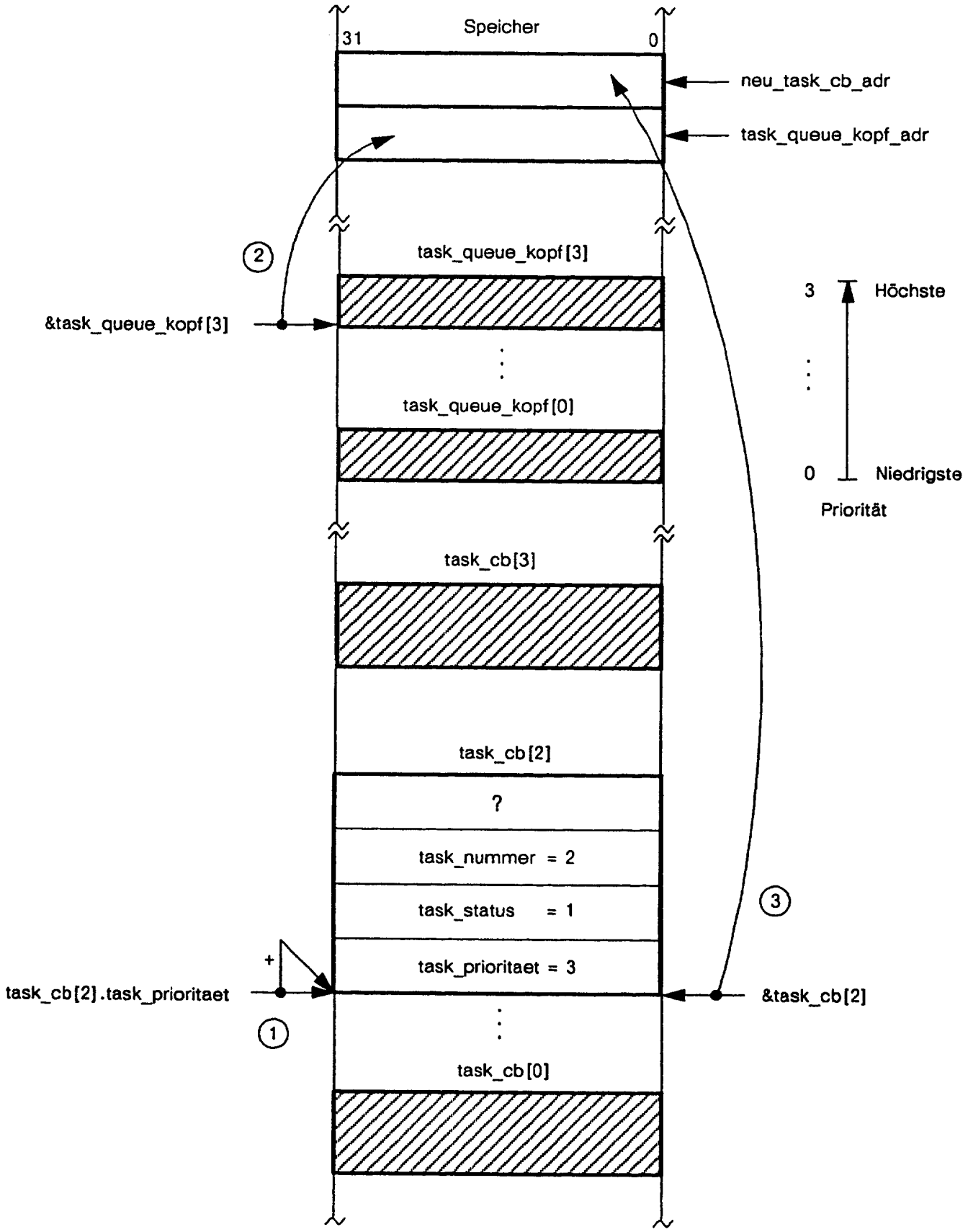


Bild 38: Adressen &task_cb[2] und &task_queue_kopf[3] ermitteln

1.6 Scheduling nach Prioritäten

Die Grundidee des Scheduling nach Prioritäten ist einfach. Jede Task erhält ihre individuelle Priorität, und der Scheduler wählt die Task mit der augenblicklich höchsten Priorität aus. Auf welche Weise dies geschieht, zeigt das folgende C-Programm, das die Betriebssystem-Funktion **scheduler** enthält:

```
#define prioritaet_max 4
#define task_max       4
#include<struct.h>

extern task_queue_kopf_struct task_queue_kopf[prioritaet_max];
char prioritaet,aktuell_task_id;

void scheduler(void)
{
 task_cb_struct *aktuell_task_cb_adr;

 while(1)
 {
   prioritaet = prioritaet_max - 1;
   while (task_queue_kopf[prioritaet].anzahl_der_task_cbe == 0)
         prioritaet --;

   aktuell_task_cb_adr =
                 task_queue_kopf[prioritaet].erster_der_queue;
   aktuell_task_id     = aktuell_task_cb_adr -> task_nummer;

   dispatcher();                    /* Aufruf des Dispatchers */
 }
}
```

Bild 39: Betriebssystem-Funktion scheduler (Datei schedule.c)

Der Scheduler benutzt die Task-Queue-Köpfe, die er, beginnend bei der **höchsten** Priorität, der Reihe nach adressiert. In jedem Task-Queue-Kopf überprüft er die Anzahl der Task-Control-Blöcke. Ist die Anzahl = **0**, dann sind bei dieser Prioritätsstufe **keine** Task-Control-Blöcke eingekettet, und er **dekrementiert** die Priorität. Diese Prozedur wiederholt sich solange, bis der Scheduler eine **nicht-leere** Task-Queue findet; dort ermittelt er die **task_nummer** im ersten eingeketteten Task-Control-Block und wählt so die Task mit der augenblicklich höchsten Priorität aus. Am Beispiel der folgenden Task-Queue-Konfiguration soll dieser Scheduling-Algorithmus verdeutlicht werden.

Hier verwaltet das Betriebssystem **vier** Anwender-Tasks, die auf drei Prioritätsstufen verteilt sind:

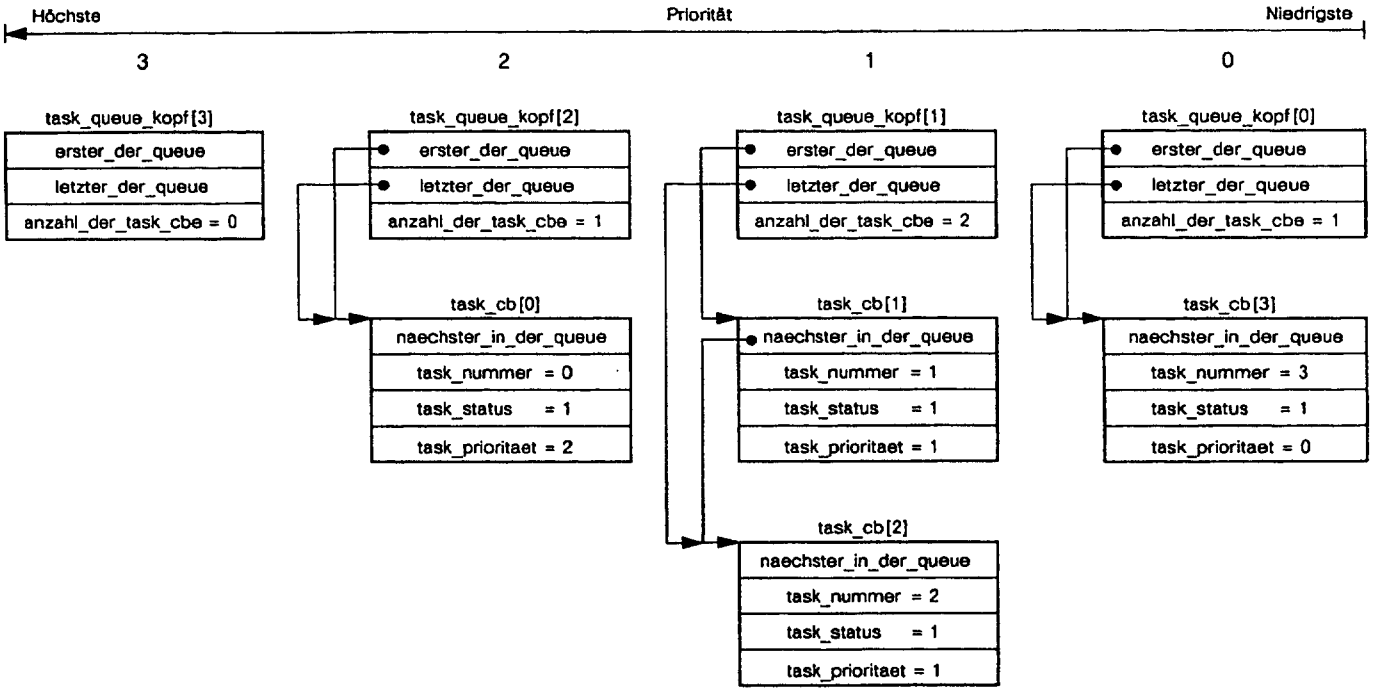


Bild 40: Scheduling nach Prioritäten

Wenn der Scheduler beginnt, dann stellt er mit

- `while (task_queue_kopf[prioritaet].anzahl_der_task_cbe == 0)`

fest, daß die Queue bei der Prioritätsstufe 3 **leer** ist. Er dekrementiert daher mit

- `prioritaet --;`

die Priorität und überprüft die nächstniedrigere Prioritätsstufe. Jetzt erkennt er, daß bei der Prioritätsstufe 2 die Queue **nicht** leer ist. Er ermittelt daraufhin mit den beiden Anweisungen

```
• aktuell_task_cb_adr =
                task_queue_kopf[prioritaet].erster_der_queue;
• aktuell_task_id     = aktuell_task_cb_adr -> task_nummer;
```

die **task_nummer** im ersten eingeketteten Task-Control-Block und speichert sie in der globalen Variablen **aktuell_task_id**. Dies zeigt das Bild 41 auf der folgenden Seite.
Anschließend ruft der Scheduler den **Dispatcher** auf, der diejenige Task startet, deren Nummer der Scheduler in der Variablen aktuell_task_id eingetragen hat. Im obigen Beispiel (Bild 40) ist dies wegen task_nummer = 0 die Task 0.
Ist sie beendet, dann ist die Task-Queue auch bei der Prioritätsstufe 2 leer, und der Scheduler findet erst wieder bei der Prioritätsstufe 1 eine nicht leere Queue. Nach dem FIFO-Prinzip wählt er dort zuerst die Task 1 (task_nummer = 1) aus, anschließend die Task 2 (task_nummer = 2), und der Dispatcher startet sie in dieser Reihenfolge.
Wenn auch diese Tasks beendet sind, dann findet der Scheduler nur noch bei der Prioritätsstufe 0 eine nicht leere Queue vor. So wählt der Scheduler als letztes die Task 3 aus (task_nummer = 3) und veranlaßt ihren Start.

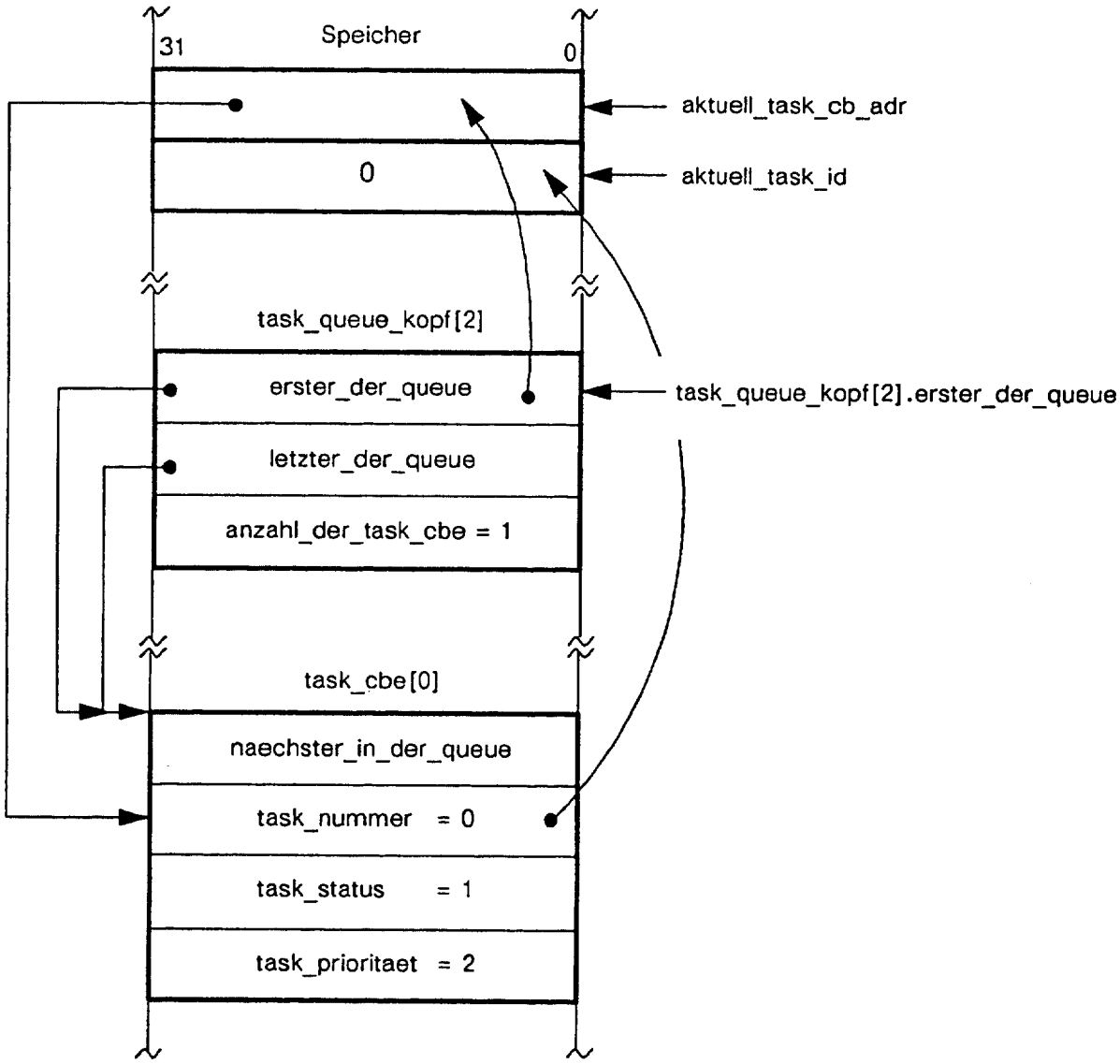


Bild 41: Ermittlung der aktuellen Tasknummer

Nach Beendigung dieser Task sind **alle** Queues leer, und der Scheduler «sucht» vergebens. Das Problem ist dabei, daß die char-Variable **prioritaet** weiter über die Werte 3,2,1,0 den Wert **–1** (FFH) erreicht; als Folge prüft der Scheduler mit der üblichen Anweisung

- `while (task_queue_kopf[-1].anzahl_der_task_cbe == 0)`

einen Speicherplatz, der zu **keinem** Task-Queue-Kopf gehört. Liegt dieser Speicherplatz außerhalb der Datensegment-Obergrenze, dann stellt die **Limit**-Überprüfung des x86/Pentium-Prozessors eine Schutzverletzung fest und generiert die Ausnahme **«Interrupt 13»**. Anderenfalls ermittelt der Scheduler eine Nummer, die zufällig in diesem Speicherplatz enthalten ist, und der Dispatcher versucht daraufhin eine Task zu starten, die **nicht** definiert ist. Auch hier stellt der Prozessor eine Schutzverletzung fest.
Um diese Probleme zu vermeiden, enthalten viele Multitasking-Systeme eine sogenannte **Idle-Task**. Sie läuft in einer Endlos-Schleife an der Prioritätsstufe 0 und wird immer dann gestartet, wenn die restlichen Prioritätsstufen leer sind.

1.7 Die Multitasking-Softwarestruktur

Ein Multitasking-System besteht aus einer **Start-Task**, einer Anzahl von **Anwender-Tasks** und der **Betriebssystemkern-Task**. Die Start- und Anwender-Tasks halten sich in der in der Privileg-Ebene **3** auf, und die Betriebssystemkern-Task in der Privileg-Ebene **0**.

Um die Adreßräume der Anwender-Tasks voneinander zu trennen, hat jede ihre eigene **l**okale **D**eskriptor-**T**abelle (LDT), deren Deskriptoren, nur die Segmente ihrer Anwender-Tasks spezifizieren. Somit sind Zugriffe auf Segmente in anderen Task ausgeschlossen. Die Betriebssystem-Funktionen sind in der Betriebssystemkern-Task zusammengefaßt. Weil viele dieser Funktionen den Anwender-Tasks über Call-Gate-Deskriptoren (System-Calls) zugänglich sein müssen, werden ihre Segmente durch Deskriptoren in der **g**lobalen **D**eskriptor-**T**abelle (GDT) spezifiziert. Jede Anwender-Task läuft daher **im Adreßraum** des Betriebssystemkerns, während sie eine Betriebssystem-Funktion in Anspruch nimmt. Im neueren Sprachgebrauch (z.B bei WINDOWS NT) wird eine solche Task dann als **Thread** bezeichnet.

Nachdem der Prozessor in den Protected-Mode umgeschaltet worden ist, befindet er sich in der Privileg-Ebene 0. Hier werden als erstes einige Hardware-Initialisierungen vorgenommen, und danach erfolgt ein Wechsel zur **Start-Task** in der Privileg-Ebene **3**. Diese Task benutzt der Anwender, um dem Betriebssystemkern die gewünschte Anzahl von Anwender-Tasks bekannt zu machen. Nachdem dies geschehen ist, erfolgt ein Wechsel zurück zur Betriebssystemkern-Task, wo ab jetzt der Scheduler den Ablauf des Multitasking-Systems bestimmt.

Im folgenden Bild 42 ist die zugehörige Software-Struktur illustriert:

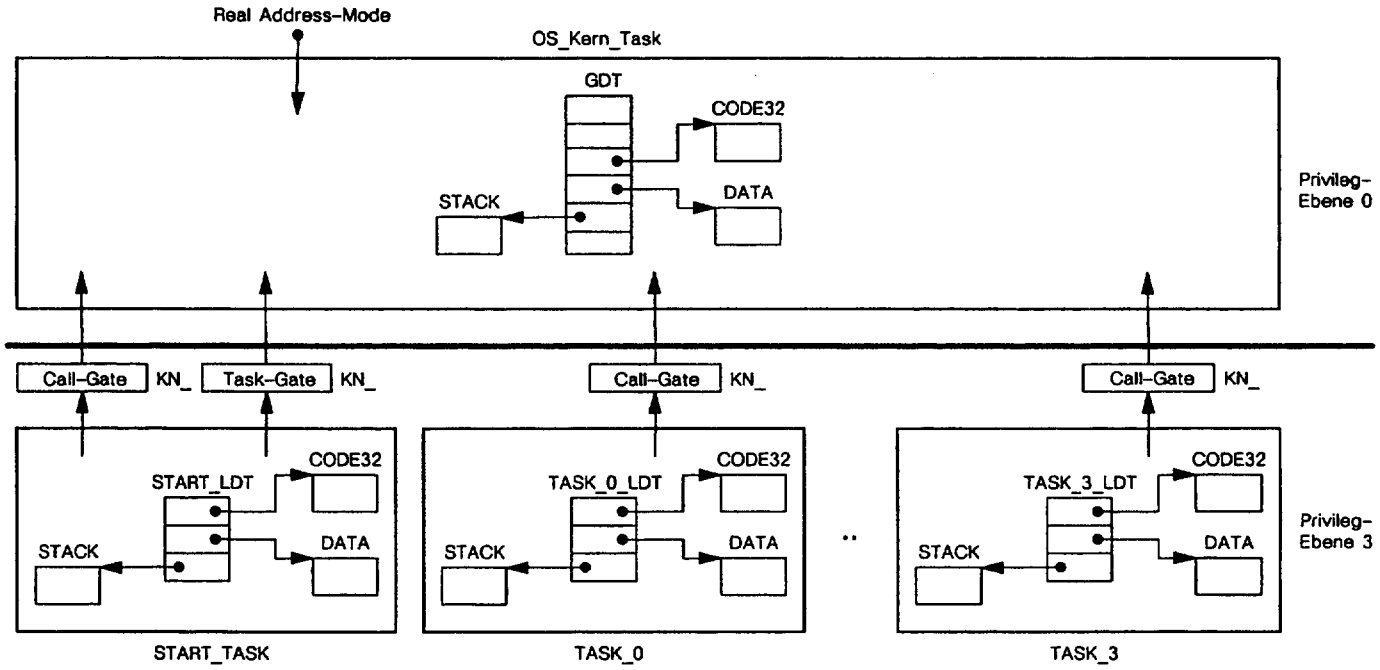


Bild 42: Mit dem «System-Builder» konfigurierte Multitasking-Software-Struktur

Die dargestellte Software-Struktur wird mit Hilfe des «System-Builders» BLD386 entworfen. Wie zu erkennen ist, sind im Beispiel insgesamt **sechs** Tasks definiert und zwar

- die **OS_Kern_Task** in der Privileg-Ebene 0 (OS = Operating System),
- die **Start_Task** in der Privileg-Ebene 3 und
- die vier Anwender-Tasks **Task_0**..**Task_3** ebenfalls in der Privileg-Ebene 3.

Der Adreßraum der OS_Kern_Task wird durch die GDT bestimmt und die Adreßräume der Start- und Anwender-Tasks durch individuelle LDTs. So definieren die lokalen Deskriptor-Tabellen

- **Start_LDT** den Adreßraum der Start_Task und
- **Task_0_LDT**..**Task_3_LDT** die Adreßräume von Task_0..Task_3.

Die Call-Gate-Deskriptoren zum Aufruf der Betriebssystem-Funktionen müssen allen Tasks auf der Privileg-Ebene 3 zur Verfügung stehen. Sie halten sich daher in der GDT auf und sind alle durch die Vorsilbe **KN_** (**K**er**n**el) gekennzeichnet.

Neben diesen Call-Gate-Deskriptoren enthält die GDT auch einen **Task-Gate**-Deskriptor. Er realisiert den Taskwechsel von der Start_Task zum Scheduler der OS_Kern-Task und wird daher auch nur von der Start_Task benutzt.

Jede einzelne Task setzt sich aus einer Reihe von **Modulen** zusammen, die ihrerseits aus Code-, Daten- und Stack-Segmenten bestehen. In Anlehnung an das 32-Bit-**Compact**-Modell der Segmentierung heißen in allen Tasks die Code-Segmente **CODE32**, die Daten-Segmente **DATA** und die Stack-Segmente **STACK**.

1.8 Implementierung eines Multitasking-Basissytems

Unmittelbar nach dem Umschalten in den Protected Mode befindet sich der Prozessor im Assembler-**Haupt**modul **task_ur** der OS_Kem_Task und führt in der Privileg-Ebene 0 die Assembler-Prozedur **task_ur_proc** aus.
Diese Prozedur ruft als erstes mit

- call init

die Betriebssystem-Funktion **init** auf, die insgesamt vier **Prioritäts**stufen 0..3 einrichtet und außerdem eine Anzahl Datenstrukturen wie z. B. semaphore, pipes, mailboxes usw. initialisiert. Diese Datenstrukturen werden wir später genauer behandeln. Danach ruft task_ur_proc mit

- call ioinit

die Betriebssystem-Funktion **ioinit** auf, die die programmierbaren Bausteine einer Zeitgeber-Hardware initialisiert.
Im folgenden Bild 43 sind diese Funktionsaufrufe illustriert:

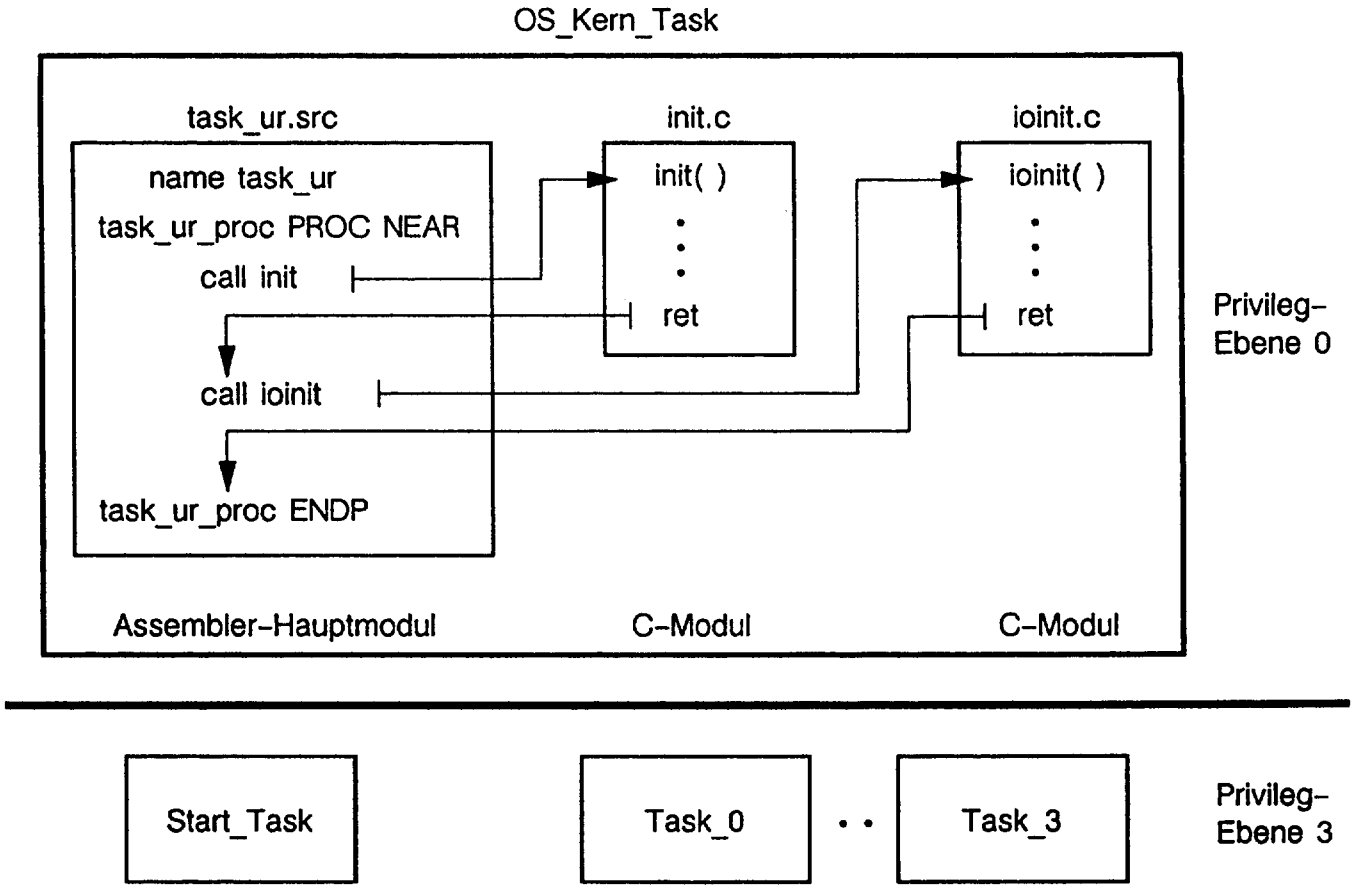


Bild 43: Aufruf der Betriebssystem-Funktionen init und ioinit beim Start des Multitasking-Systems

1.8.1 Erzeugung von Zeittakten

Da jede **Anwender**-Task nur für eine gewisse **Zeitspanne** (time slice) den Prozessor besitzen darf, muß nach Ablauf dieser Zeitspanne die Task dem Prozessor entzogen und auf eine andere Task umgeschaltet werden. Auf diese Weise wird die CPU-Zeit auf mehrere Tasks verteilt, und es sieht für den Benutzer so aus, als ob die Anwender-Tasks **parallel** ablaufen würden. In einem Multitasking-System müssen solche zeitgesteuerten Aktionen für jede Anwender-Task möglich sein. Unterstützt werden diese Aktionen von einer programmierbaren Zeitgeber-Hardware, die periodisch einen Impuls liefert und damit die Anwender-Tasks Task_0..Task_3 fortwährend unterbricht. Oft weden diese periodischen Unterbrechungen auch **Clock-Ticks** oder **Zeittakte** genannt. Das Modell der Zeitgeber-Hardware zeigt das folgende Bild 44:

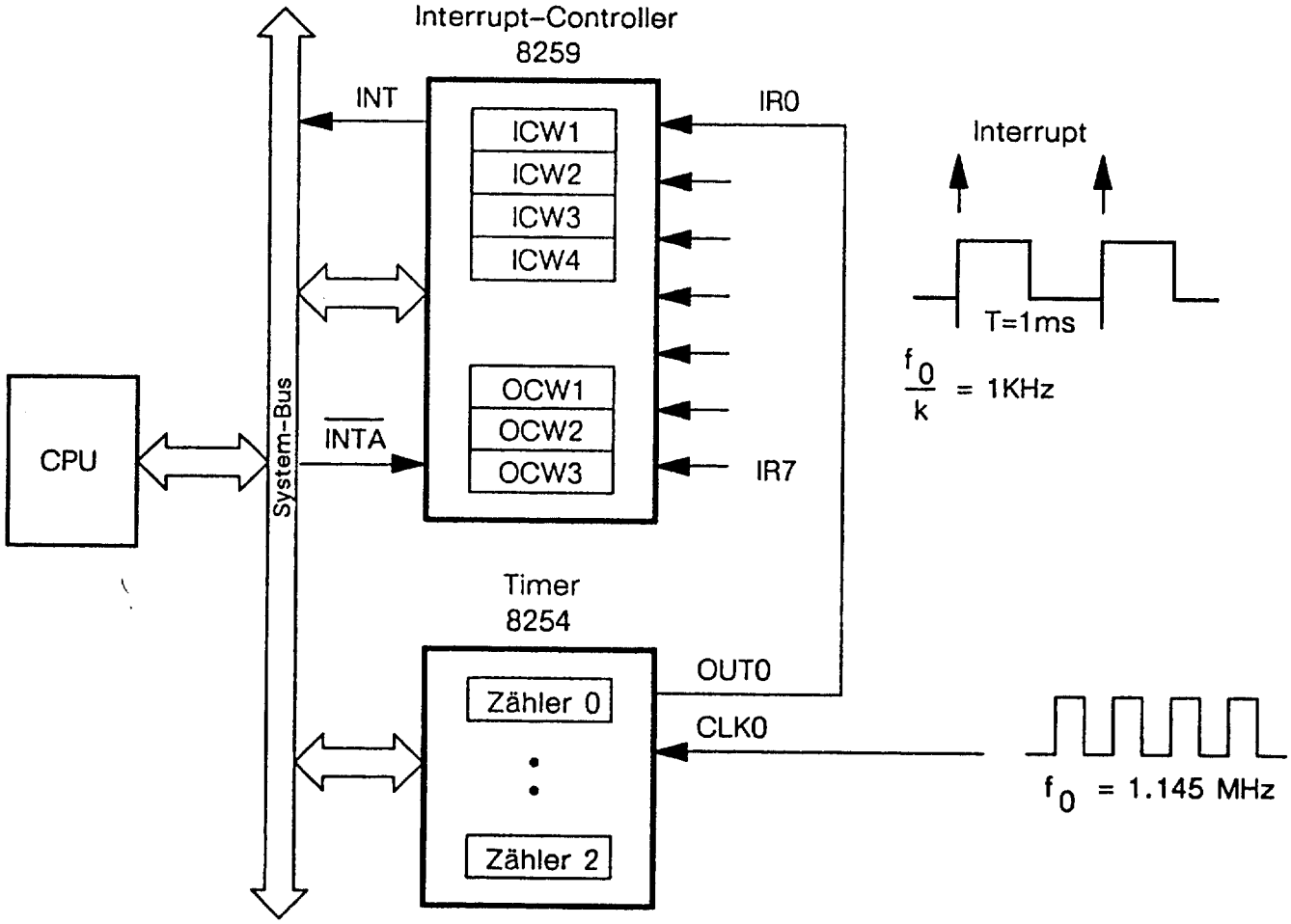


Bild 44: Zeitgeber-Hardware

Wie zu sehen ist, besteht sie aus zwei programmierbaren Bausteinen, nämlich

- dem **Timer**-Baustein 8254 und
- dem **Interrupt**-Controller 8259

Der Timer-Baustein enthält **drei** unabhängig voneinander arbeitende 16 Bit-Abwärtszähler (Zähler 0..Zähler 2), wobei hier nur der Zähler 0 benutzt wird. An seinem Clock-Eingang CLK 0 steht ein periodisches Signal mit einer Frequenz von $\mathbf{f_0 = 1.145\ MHz}$ ($1.145 \cdot 10^6$ Hz) an. Die Betriebssystem-Funktion **ioinit** programmiert den Zähler 0 so, daß er als Rechteckgenerator arbeitet und am Ausgang **OUT 0** ein periodisches Signal mit einer Periodendauer $\mathbf{T = 1ms}$ ($1 \cdot 10^{-3}$ s) ausgibt. Der Zähler 0 wird also mit dem Wert $\mathbf{k = T \cdot f_0}$ initialisiert:

- $k = 1 \cdot 10^{-3}s \cdot 1.145 \cdot 10^6/s = \mathbf{1145}$

Weil der Ausgang OUT 0 mit dem **I**nterrupt-**R**equest-Eingang IR0 des Interrupt-Controllers verbunden ist, fordert der Zähler 0 in Intervallen von 1 ms Unterbrechungen des Programms.
Der Interrupt-Controller enthält vier **ICW**s (**I**nitial **C**ode **W**ord) und drei **OCW**s (**O**perating **C**ode **W**ord), mit deren Hilfe ioinit die Betriebsart des Bausteins bestimmt.
So macht ioinit über ICW1 den IR0-Eingang **flankensensitiv** und teilt dem Controller mit, daß er als **einziger** im System arbeitet, also nicht kaskadiert ist. Über ICW2 bestimmt ioinit die **Vektor-Nummer** für Interrupts an IR0, die dem Controller über ICW4 mitteilt, daß er mit einem **x86**/Pen-

tium-Prozessor im **vollverschachtelten** Modus zusammenarbeitet, außerdem **nicht** gepuffert ist und daß das In-Service-Bit **automatisch** mit Interrupt Acknowledge (INTA*; * = Low-aktiv) gelöscht wird. Zusätzlich erlaubt ioinit über OCW1 nur Programmunterbrechungen an IR0, d. h. nur **Timer**-Interrupts.
Das vollständige C-Programm der Betriebssystem-Funktion **ioinit**
zeigt das folgende Bild 45:

```
#include<i86.h>

                /* Adressen des Timers 8254 */

unsigned short int zaehler0   =  0x40;    /* Zaehler 0              */
unsigned short int steuerport =  0x43;    /* Steuerport             */

                /* Adressen des Interrupt-Controllers 8259 */

unsigned short int icw1_reg   =  0x20;    /* ICW1-Register          */
unsigned short int icw2_reg   =  0x21;    /* ICW2-Register          */
unsigned short int icw4_reg   =  0x21;    /* ICW4-Register          */
unsigned short int ocw1_reg   =  0x21;    /* OCW1-Register          */

                /* Timer-Initialisierung */

unsigned char steuerbyte_zaehler0 = 0x36; /* Zuerst Low-Byte,       */
                                          /* dann High-Byte laden   */
                                          /* Rechteckgenerator      */
                                          /* binaeres Zaehlen       */

                /* Interrupt-Controller-Initialisierung */

unsigned char icw1            = 0x13;     /* ICW1:                  */
                                          /* flankengetriggert      */
                                          /* einzeln                */
                                          /* ICW4 notwendig         */
unsigned char icw2            = 0x20;     /* ICW2:                  */
                                          /* Interrupt-Vektor an IR0 */
unsigned char icw4            = 0x03;     /* ICW4:                  */
                                          /* voll verschachtelt     */
                                          /* nicht gepuffert        */
                                          /* automatisches EOI      */
                                          /* 80x86-Prozessor        */
unsigned char ocw1            = 0xfe;     /* OCW1:                  */
                                          /* Masken setzen: nur     */
                                          /* Timer-Interrupt erlaubt */
short int teiler = 1145;                  /* Teilerfaktor:          */
                                          /* Periodendauer 1ms      */
char teiler_low,teiler_high;              /* Low- und Highbyte      */
                                          /* des Teilerfaktors      */
```

```
void ioinit(void)
{
                                        /* Betriebsart Zaehler 0   */
                                        /* festlegen               */
 outbyte(steuerport,steuerbyte_zaehler0);
                                        /* Low- und High-Byte des  */
                                        /* Teilerfaktors           */
                                        /* ermitteln               */
 teiler_low  = (char) teiler & 0xff;
 teiler_high = (char) (teiler >> 8) & 0xff;
 outbyte(zaehler0,teiler_low);          /* Low-Byte des            */
                                        /* Teilerfaktors laden     */
 outbyte(zaehler0,teiler_high);         /* High-Byte des           */
                                        /* Teilerfaktors laden     */

 outbyte(icw1_reg,icw1);                /* ICW1 laden              */
 outbyte(icw2_reg,icw2);                /* ICW2 laden              */
 outbyte(icw4_reg,icw4);                /* ICW4 laden              */
 outbyte(ocw1_reg,ocw1);                /* OCW1 laden              */
}
```

Bild 45: Betriebssystem-Funktion ioinit (Datei ioinit.c)

Dem Programm ist die **Header**-Datei **i86.h** «einverleibt». Neben anderem ist dort die Funktion outbyte definiert:

- `void outbyte (unsigned short port,unsigned char value)`

Sie ermöglicht einem C-Programm **Schreib**-Zugriffe auf I/O-Ports, und ioinit benutzt sie, um den Timer und den Interrupt-Controller zu initialisieren. So generiert der 32-Bit-C-Compiler ic386/486 für den Funktionsaufruf

- `outbyte(port,value);`

den folgenden Assembler-Code:

```
• MOV AL,value
  MOV DX,port
  OUT DX,AL
```

1.8.2 Anwender-Tasks anmelden und System starten

Nachdem init die Privilegstufen eingerichtet und ioinit die Zeitgeber-Hardware initialisiert hat, setzt die Assembler-Prozedur task_ur_proc ihre Arbeit fort und veranlaßt mit dem **FAR**-Jump-Befehl (16-Bit-Selektor, 32-Bit-Offset)

- `jmp start_task`

über den TSS-Deskriptor der Start_Task einen Taskwechsel zur Start_Task in der Privileg-Ebene 3. Siehe Bild 46:

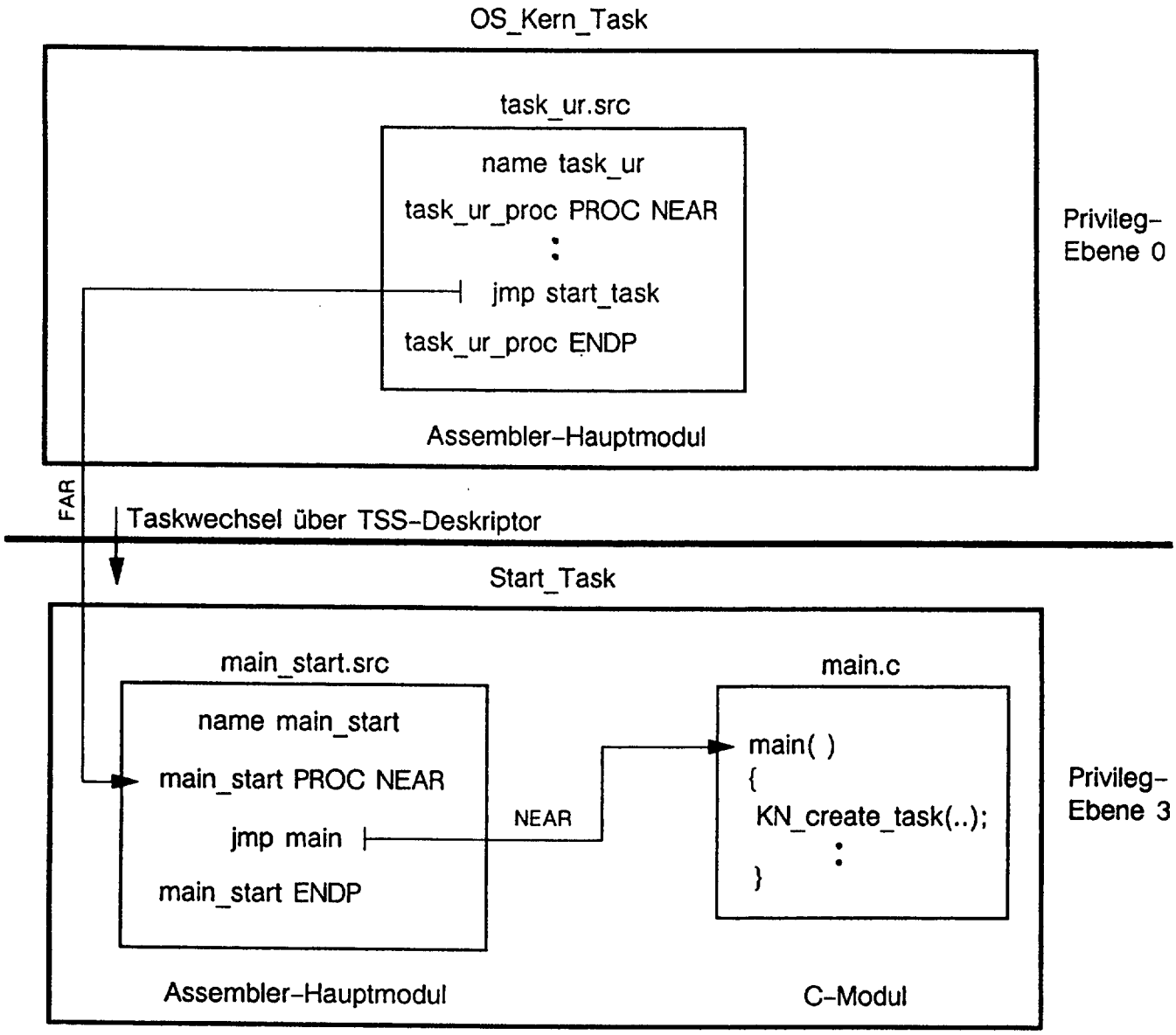


Bild 46: Wechsel zur Start_Task

Die Start_Task führt den Code von **zwei** Modulen aus, und zwar

- zuerst die Prozedur **main_start** ihres Assembler-**Haupt**moduls und
- dann die Funktion **main** des C-Anwendermoduls

Die Programme dieser beiden Module haben folgenden Aufbau (Bilder 47 und 48):

```
            NAME main_start

EXTRN main:near
PUBLIC main_start

DATA SEGMENT RW PUBLIC
     DW 100 dup (?)
DATA ENDS

STACK STACKSEG 100

CODE32 SEGMENT ER PUBLIC

main_start PROC  NEAR

      jmp main              ;Aufruf des Main-Moduls

main_start ENDP

CODE32 ENDS
       END main_start,DS:DATA,SS:STACK
```

Bild 47: Assembler-Hauptmodul der Start_Task (Datei main_sta.src)

```
#define ready 1

extern void far KN_create_task(char,char,char);
extern void far KN_start_scheduler(void);

char task_id,task_status,task_prioritaet;

void main(void)

{
 /* Vier Anwender-Tasks 0..3 anmelden */

 task_id=0,task_status=ready,task_prioritaet=2;
 KN_create_task(task_id,task_status,task_prioritaet);  /*Call-Gate*/

 task_id=1,task_status=ready,task_prioritaet=2;
 KN_create_task(task_id,task_status,task_prioritaet);  /*Call-Gate*/

 task_id=2,task_status=ready,task_prioritaet=2;
 KN_create_task(task_id,task_status,task_prioritaet);  /*Call-Gate*/

 task_id=3,task_status=ready,task_prioritaet=2;
 KN_create_task(task_id,task_status,task_prioritaet);  /*Call-Gate*/

 /* Übergabe der Multitasking-Steuerung an den Scheduler */

 KN_start_scheduler();                                 /*Task-Gate*/
}
```

Bild 48: Anwendermodul der Start_Task (Datei main.c)

Über die Funktion **main** der Start_Task kann der Benutzer eine **beliebige** Anzahl betriebsbereiter Tasks dem System **bekannt** machen.
Diese Aktion führt zu einem Zustandsübergang, der sich formal folgendermaßen beschreiben läßt:

- `create (Taskname); unbekannt -> bereit`

Das im Bild 6 dargestellte Task-Zustandsdiagramm kann somit ergänzt werden:

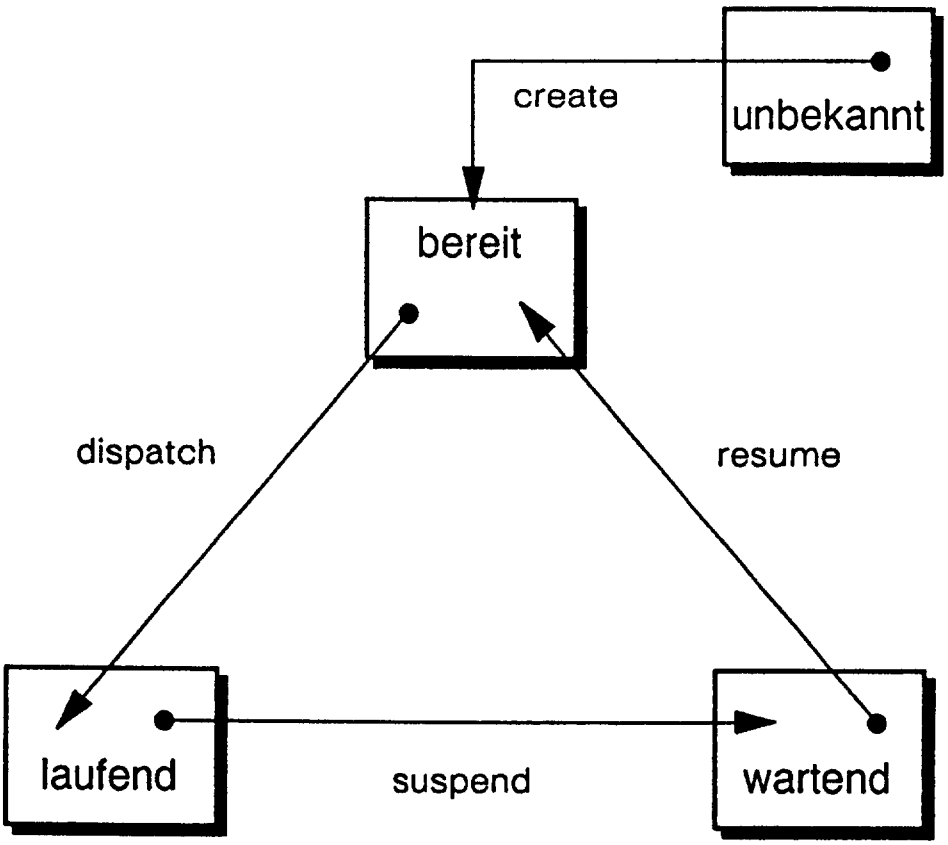


Bild 49: Task-Zustandsdiagramm; erste Ergänzung

Die Funktion main der Start-Task realisiert den neuen Übergang, indem sie über den Call-Gate-Deskriptor

- `KN_crate_task(char,char,char);`

die Betriebssystem-Funktion **task_init** mit folgenden drei Parametern aufruft:

- **task_id** (**Task-Id**entifizierer bzw. Tasknummer)
- **task_status** und
- **task_prioritaet**

Mit jedem Aufruf initialisiert task_init einen Task-Control-Block und kettet ihn bei der spezifizierten Prioritätsstufe ein. Auf diese Weise werden dem Betriebssystem eine beliebige Anzahl betriebsbereiter Anwender-Tasks bekannt gemacht.
Im obigen Beispiel (siehe Bild 48) ruft main die Betriebssystem-Funktion task_init **4 mal** auf. Und so werden vier Anwender-Tasks angelegt(task_id = 0..3 und task_status = ready), die alle die **gleiche** Priorität haben (task_prioritaet = 2). Siehe Bild 50.

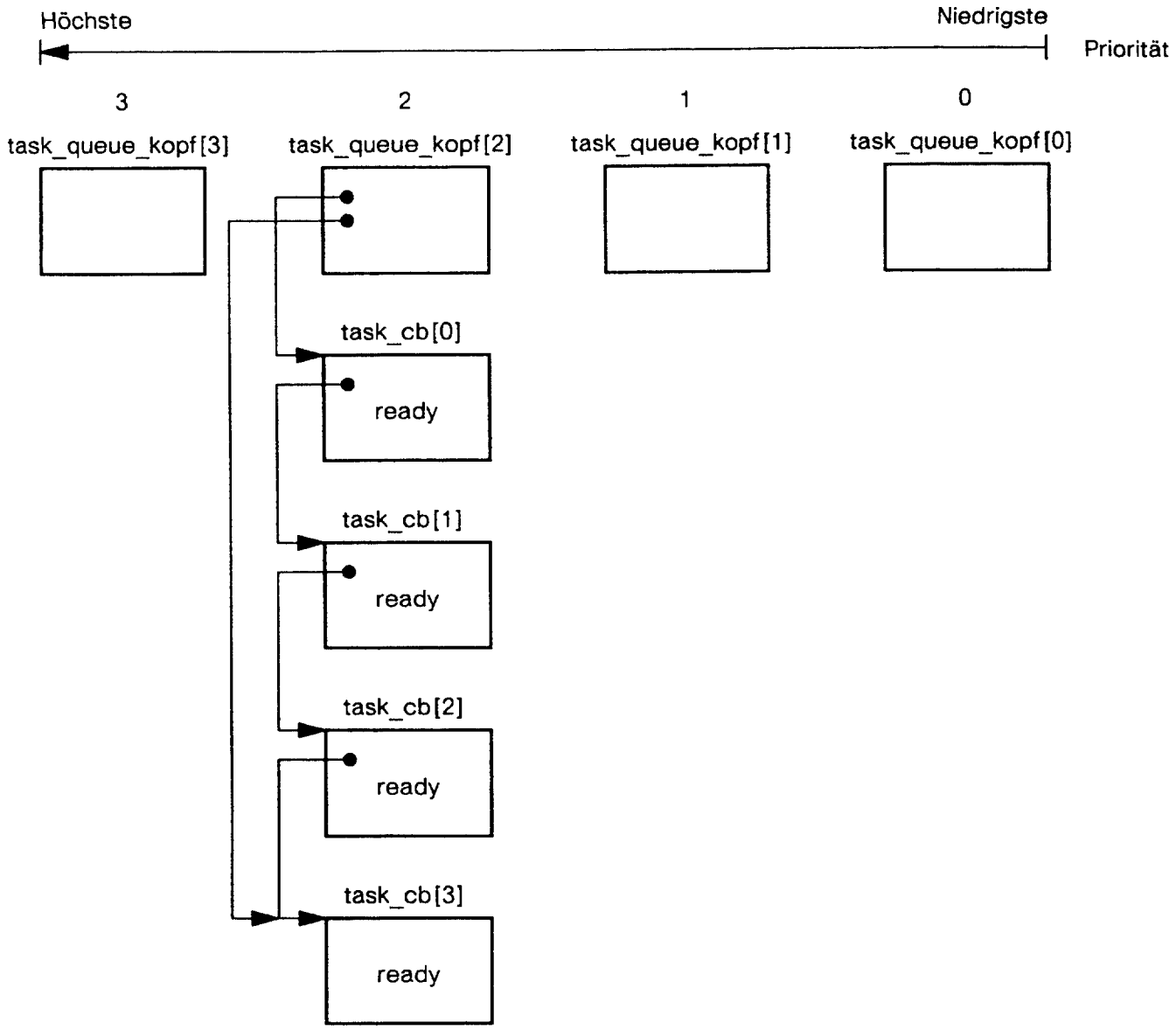


Bild 50: Anlegen von vier betriebsbereiten Tasks

Mit jedem System-Call (über einen Call-Gate-Deskriptor) ist ein Privilegwechsel von der Ebene 3 in die Ebene 0 verbunden. Dies hat zur Folge, daß die Start-Task und auch die Anwender-Tasks die Segmente **CODE32**, **DATA** und **STACK** der OS_Kern_Task benutzen. Daher müssen die Selektoren **CS**, **DS**, **SS** und möglicherweise auch **ES**, **FS** und **GS** umgeladen werden. Für den Anwender-Programmierer in der Privileg-Ebene 3 ist jeder System-Call ein sogenannter «**entfernter**» Funktionsaufruf. Aus diesem Grund werden vom Prozessor automatisch auch die Stack-Segmente gewechselt, und die Parameter vom Stack der Ebene 3 in den Stack der Ebene 0 kopiert. Die Folge ist, daß der Assembler-Adapter noch weitere Aufgaben hat. So muß er die kopierten Parameter im Ebene-0-Stack so verschieben, daß die Betriebssystem-Funktion **task_init** sie an den richtigen Stellen im Stack vorfindet, und von dort in gewohnter Weise entnehmen kann.
Der Assembler-Adapter gehört zur OS_Kern_Task und enthält die Prozedur **adapt00**, deren Adresse sich im Call-Gate-Deskriptor **KN_create_task** befindet. Der Aufruf von task_init geschieht daher indirekt über den Adapter adapt00, der zunächst die Selektoren umlädt, die kopierten Parameter verschiebt und schließlich die Betriebssystem-Funktion aufruft. Die Korre-

spondenzen zwischen der Funktion **main** der Start_Task, dem **Call-Gate-Deskriptor**, dem **Assembler-Adapter** und der Betriebssystem-Funktion **task_init** sind im folgenden Bild 51 illustriert:

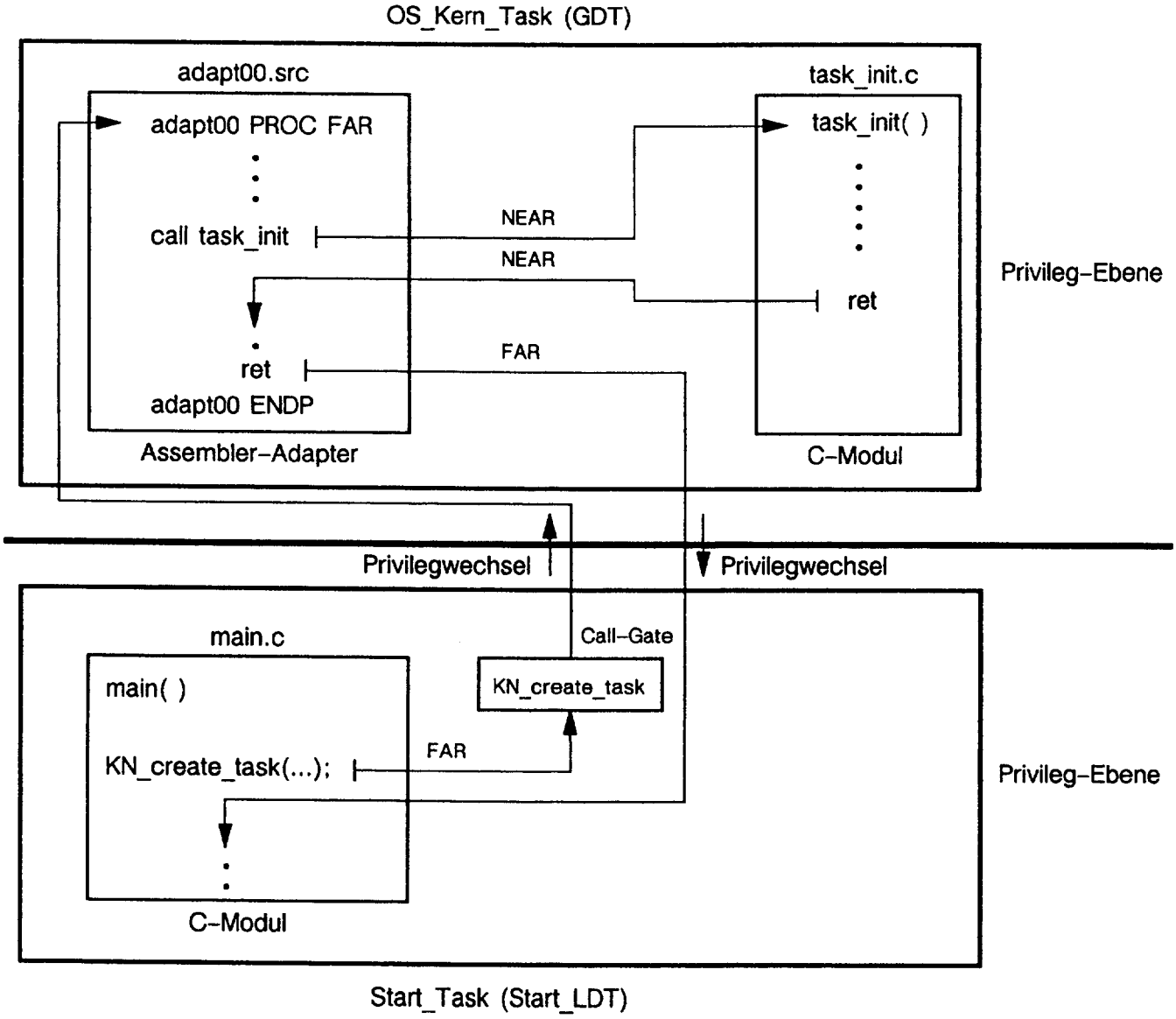


Bild 51: Aufruf der Betriebssystem-Funktion task_init

Wenn mit KN_create_task(task_id,task_status,task_prioritaet) ein System-Call veranlaßt wird, dann werden zunächst die Parameter von links nach rechts (fixedparams) an den Stack der Ebene 3 übergeben. Dabei ergibt sich folgendes Stack-Layout:

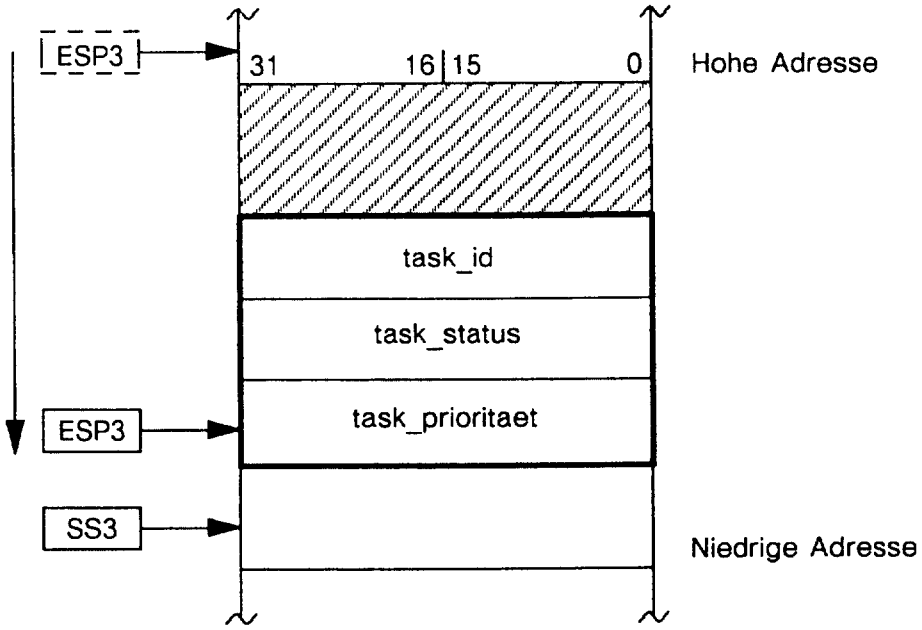


Bild 52: Ebene-3-Stack **vor** dem Privileg-Wechsel

Danach wird der Privileg-Wechsel durchgeführt, und der Prozessor speichert als erstes den Zeiger SS3:ESP3 im Stack der Ebene 0. Er stellt damit eine Verbindung zum Stack der Ebene 3 her. Anschließend kopiert er die Anzahl der Parameter vom alten Stack in den neuen Stack. Dies wird vom DWORD COUNT-Feld im Call-Gate-Deskriptor bestimmt. Schließlich speichert er noch den Zeiger CS3:EIP3, der die Rücksprungadresse zum aufrufenden CODE32-Segment der Start_Task darstellt. Damit ergibt sich das folgende Stack-Layout:

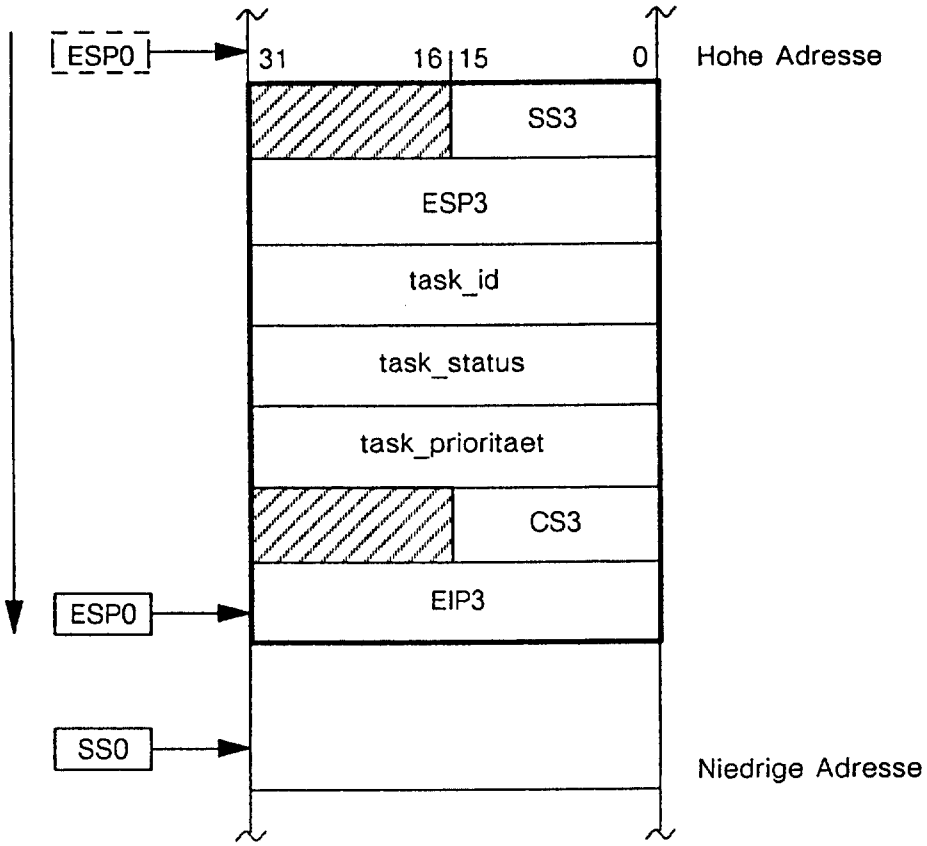


Bild 53: Ebene-0-Stack **nach** dem Privileg-Wechsel

Ausgehend von dieser Stack-Situation, führt jetzt die Adapter-**FAR**-Prozedur adapt00 die folgenden Assembler-Befehle aus:

```
NAME adapt00

PUBLIC adapt00
EXTRN  task_init:NEAR

DATA SEGMENT RW PUBLIC
     DD  ?
DATA ENDS

CODE32 SEGMENT ER PUBLIC

adapt00 PROC FAR WC(3)
        push   ebp            ;Prolog
        mov    ebp,esp
        push   ds             ;DS-Selektor der Start-Task retten
        mov    eax,[ebp+20]   ;Parameter: task_id
        push   eax
        mov    eax,[ebp+16]   ;Parameter: task_status
        push   eax
        mov    eax,[ebp+12]   ;Parameter: task_prioritaet
        push   eax
        mov    ax,data        ;DS-Selektor/Deskriptor der OS-Kern-Task
        mov    ds,ax          ;laden
        call   task_init      ;Aufruf der Betriebssystem-Funktion task_init
        pop    ds             ;DS-Selektor/Deskriptor der Start-Task laden
        pop    ebp            ;Epilog
        ret    12             ;Zurueck zur Start-Task und Parameter entfernen
adapt00 ENDP

CODE32 ENDS
       END
```

Bild 54: Adapter adapt00 für Betriebssystemfunktion task_init (Datei adapt00.src)

Nach dem Prolog **rettet** adapt00 mit **push ds** den DS-Selektor der Start_Task (wenn notwendig, können auch ES, FS und GS gerettet werden) und kopiert anschließend mit

- `mov eax,[ebp+20]`
  `push eax`
- `mov eax,[ebp+16]`
  `push eax`
- `mov eax,[ebp+12]`
  `push eax`

die Parameter task_id, task_status und task_prioritaet an Stack-Positionen jenseits der Rücksprungadresse CS3:EIP3. Das resultierende Stack-Layout der Ebene 0 zeigt Bild 55:

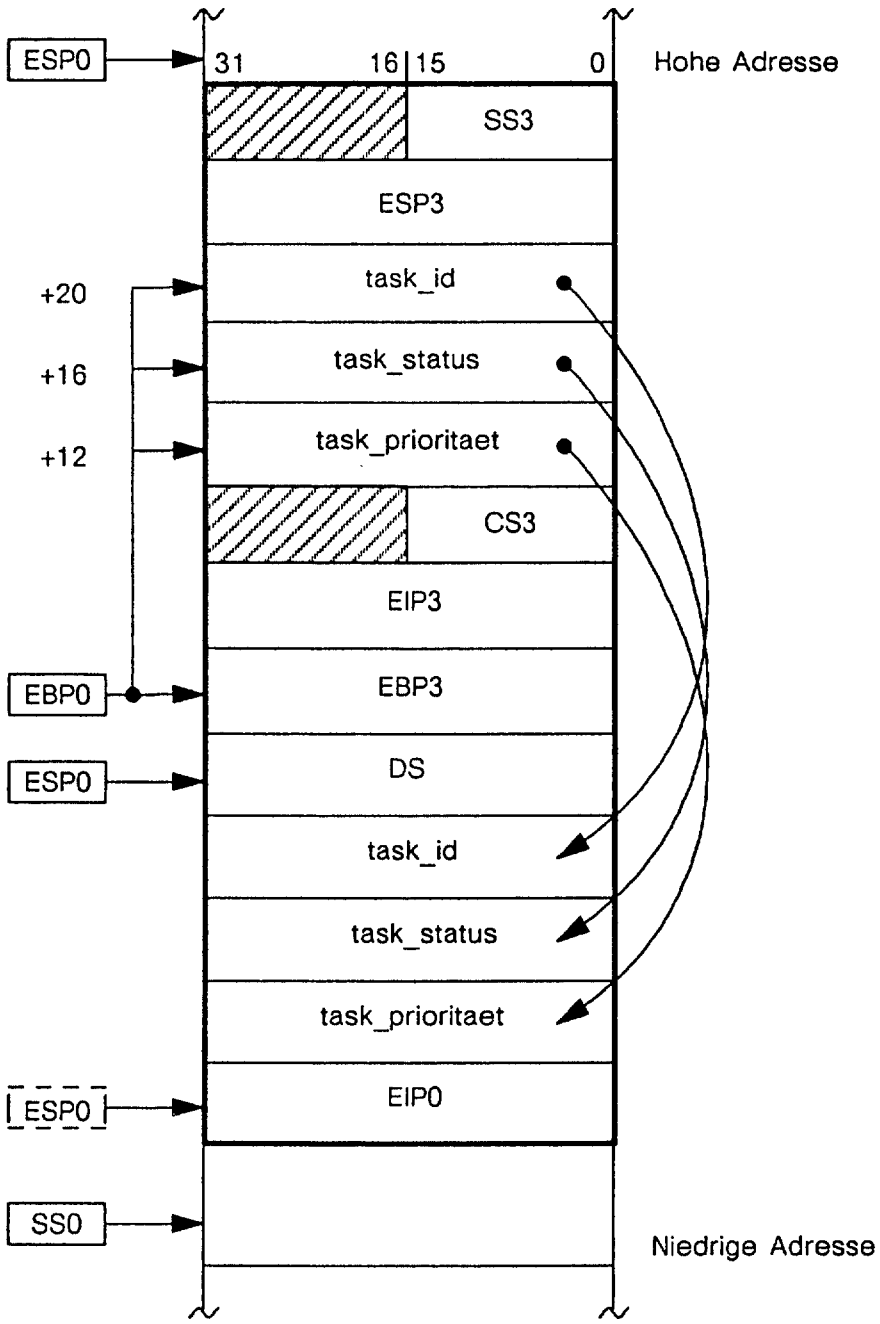


Bild 55: Resultierendes Stack-Layout der Ebene 0

Bevor adapt00 die Betriebssystem-Funktion task_init aufruft, lädt sie mit

```
• mov  ax,data
  mov  ds,ax
```

das DS-Register um. Danach zeigt DS auf das **DATA**-Segment der **OS_Kern_Task**, die nun bei Bedarf problemlos auf ihre statischen Variablen DS-relativ zugreifen kann. Nach dem Aufruf von task_init findet die Funktion die Parameter an den richtigen Stack-Positionen und kann sie von dort in gewohnter Weise abholen und verarbeiten.
Am Ende führt task_init den Befehl **ret 12** aus (wird vom C-Compiler generiert) und «entfernt» damit die vom Assembler-Adapter bereitgestellten Parameter aus dem Stack der Ebene 0. Zusätzlich sorgt ret 12 für die Rückkehr zum Adapter, und zwar zum Befehl **pop ds** (NEAR-Return).

Dieser Befehl lädt das DS-Register um, so daß danach DS wieder auf das **DATA**-Segment der **Start_Task** zeigt. Der nun folgende Befehl **pop ebp** im Epilog lädt den aktuellen Base-Pointer der Privileg-Ebene 3 (ebp3) zurück in das EBP-Register, und ein weiterer **ret 12** «entfernt» die vom **Prozessor** kopierten Parameter aus dem Ebene-0-Stack.
Zusätzlich veranlaßt dieser Befehl die Rückkehr zur Start_Task in der Privileg-Ebene 3 (FAR-Return). Siehe hierzu Bild 51. Nachdem die Start_Task dem Betriebssytem die gewünschte Anzahl betriebsbereiter Tasks bekannt gemacht hat, übergibt sie die Steuerung des Multitasking-Systems an den **Scheduler**. Sie benutzt zu diesem Zweck den **Task-Gate**-Deskriptor

- `KN_start_scheduler;`

der einen Taskwechsel von der Start_Task in die OS_Kern_Task zum task_ur_PROC-Befehl **jmp scheduler** veranlaßt. Siehe Bild 56:

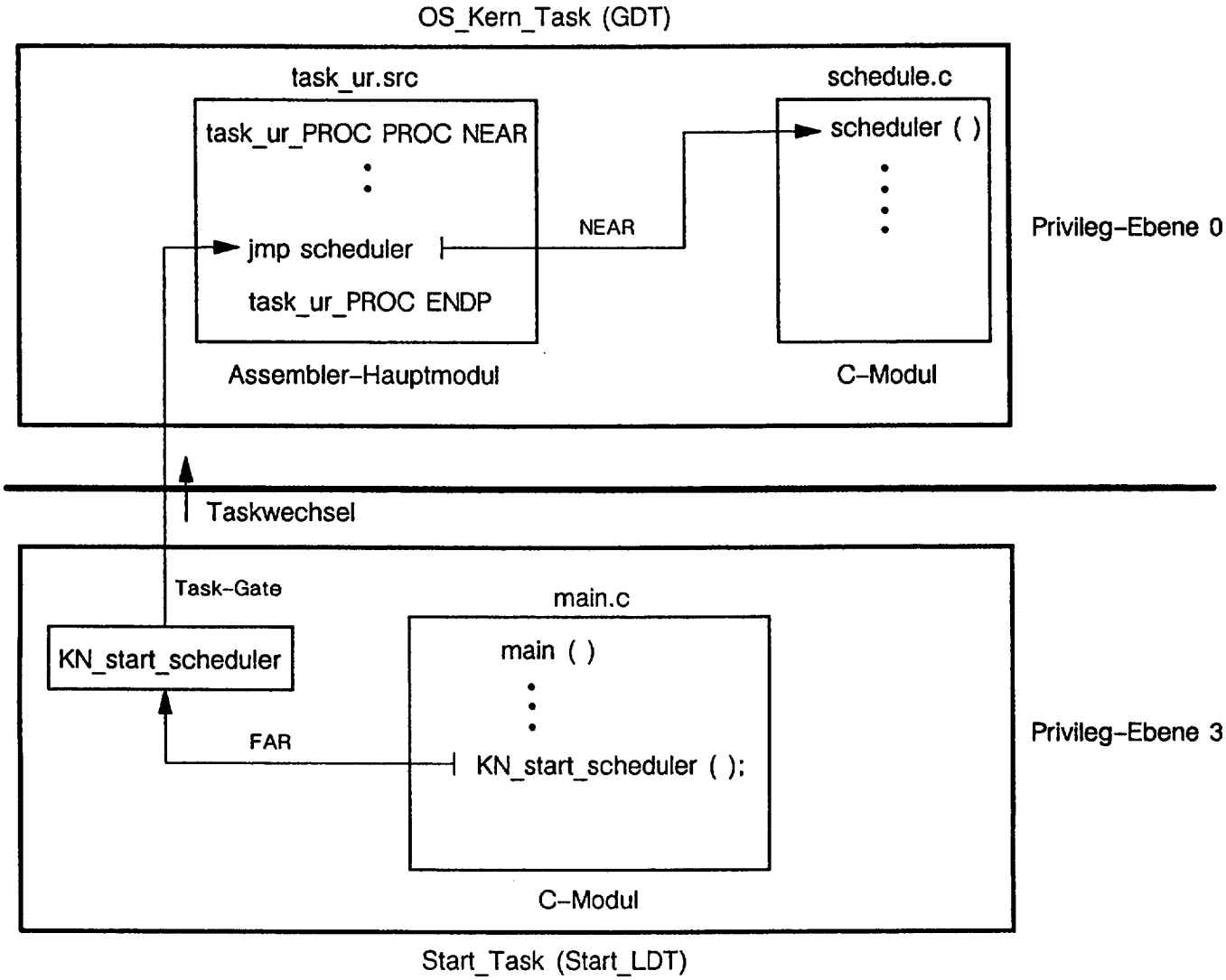


Bild 56: Übergabe der Multitasking-Steuerung an den Scheduler

Ist der Taskwechsel realisiert, hat der Benutzer keinerlei Einfluß mehr auf den Ablauf des Systems. Denn der Befehl jmp scheduler ruft den Scheduler auf, der von jetzt ab zusammen mit dem Dispatcher die Ablaufsteuerung des Multitasking-Systems übernimmt.

Damit ist die Arbeit der Betriebssystem-Prozedur task_ur_Proc beendet, und sie wird im weiteren Geschehen nicht mehr benutzt; hier nun ihr vollständiger Assembler-Code:

```
      NAME task_ur
EXTRN init:NEAR,ioinit:NEAR,start_task:FAR,scheduler:NEAR
PUBLIC task_ur_PROC

DATA SEGMENT RW PUBLIC
     DD  ?
DATA ENDS

STACK STACKSEG 1200H

CODE32 SEGMENT ER PUBLIC

task_ur_PROC PROC  NEAR

            call init           ;OS-Kern-Datenstrukturen initialisieren
            call ioinit         ;I/O-Bausteine initialisieren
            jmp start_task      ;Wechsel in die Start-Task ueber
                                ;einen TSS-Deskriptor
            jmp scheduler       ;Aufruf des Schedulers

task_ur_PROC ENDP

CODE32 ENDS
      END task_ur_PROC,DS:DATA,SS:STACK
```

Bild 57: Betriebssystem-Funktion task_ur_PROC (Datei task_ur.src)

1.9 Taskwechsel-Schnittstellen

1.9.1 Der Dispatcher

Die x86/Pentium-Architektur unterstützt den Entwurf des Betriebssystem-Kerns (engl.: Kernel), der alle wichtigen Funktionen des Betriebssystems enthält.

So ist der Kern z. B. dafür zuständig, die verfügbare CPU-Zeit nach einer **gerechten** Strategie auf die **Anwender-Tasks** zu verteilen. Wenn die Anwender-Tasks in einen Dialog treten wollen, dann muß der Kern zusätzlich Dienste zur Synchronisation und Kommunikation zur Verfügung stellen usw.

Um zu erreichen, daß diese Dienste von **allen** Anwender-Tasks in Anspruch genommen werden können, müssen sie sich im **globalen** Adreßraum aufhalten, der von der GDT (**G**lobale **D**eskriptor **T**abelle) definiert wird.

Es muß außerdem sichergestellt sein, daß diese Dienste nicht «wild» benutzt werden. Sie halten sich daher in der **Privileg-Ebene 0** auf und können dort nur über entsprechende Call-Gate-Deskriptoren aufgerufen werden. Die Folge ist, daß jede Anwender-Task in **zwei** Bereichen läuft, nämlich im

- **User**-Bereich der Privileg-Ebene 3 und im

- Betriebssystem-**Kern**-Bereich der Privileg-Ebene 0.

Eine solche Struktur ist typisch für sogenannte **monolithische** Betriebssysteme: Hier besteht der Betriebssystem-Kern aus einer Anzahl von Funktionen, die sich je nach Situation gegenseitig aufrufen können. Mit anderen Worten, der Kern repräsentiert sich als «große Unordnung».
Den Anwender-Tasks bleibt dies aber verborgen. Denn sie benutzen nur eine Teilmenge dieser Funktionen (Dienste), die sie mittels Call-Gate-Deskriptoren nur über **wohldefinierte** Schnittstellen erreichen. So gibt es auch in einem monolitischen System eine Minimalstruktur.
Ist auch der Kern als Task definiert, dann führt dies zu einem sogenannten **task-orientierten** Betriebssystem, wie es im folgenden Bild illustriert ist:

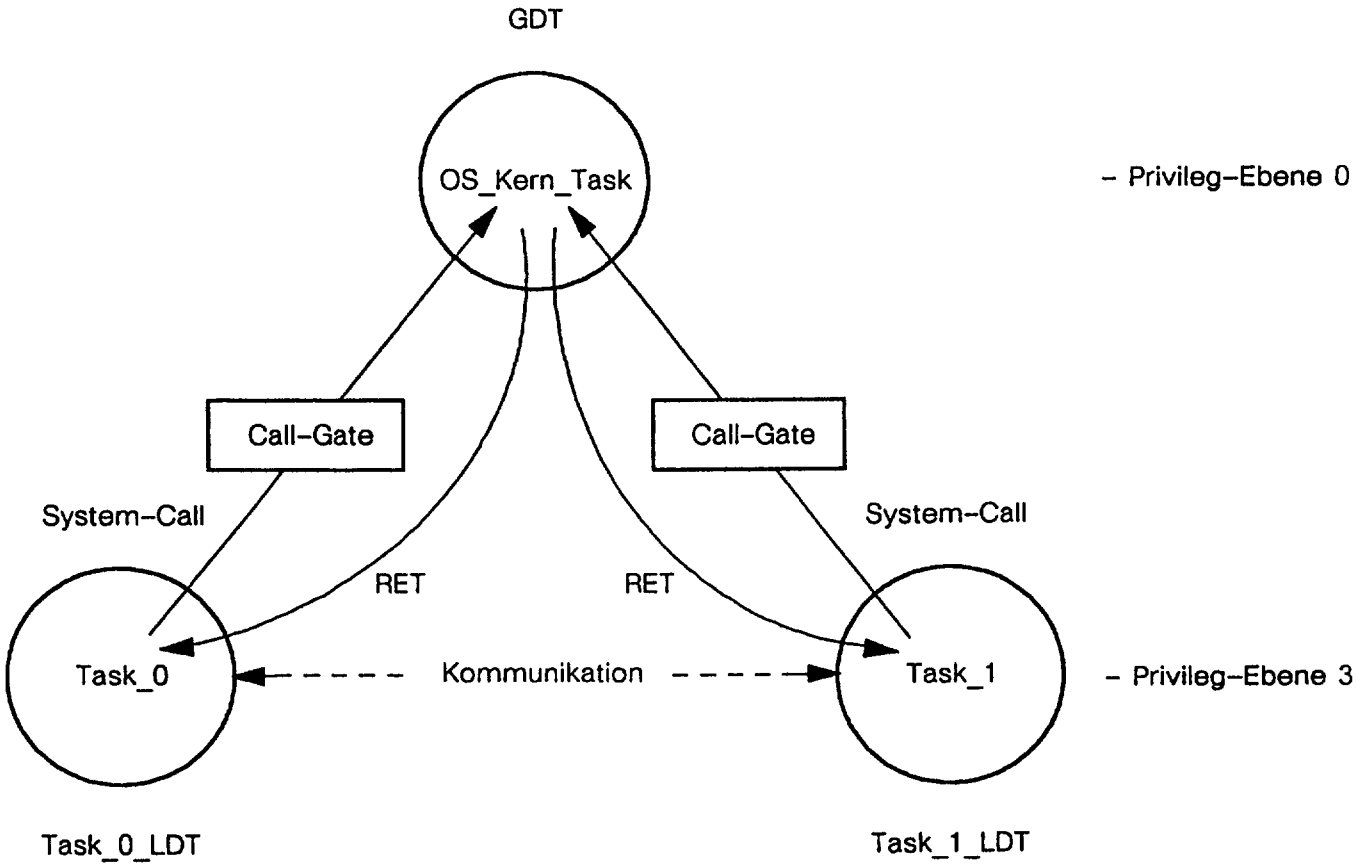


Bild 58: Task-orientiertes Betriebssystem

Anmerkung: Weil in der Regel eine Task in einer **Endlos**-Schleife läuft, wird sie oft als **Kreis** dargestellt. Eine solche Schleife kann in C folgendermaßen definiert werden:

```
• while (1)
  {
   task-code;
  }
```

Jede Task in einem Multitasking-System wird durch ihr **Task-Status-Segment** und ihren **Task-Control-Block** identifiziert. Hat der Scheduler eine arbeitsbereite Anwender-Task über ihren Task-Control-Block ausgewählt, dann aktiviert er den Dispatcher, der daraufhin die Anwender-Task (Task_3..Task_0) startet. Weil sich der Dispatcher als Teil der OS_Kern_Task in der Privileg-Ebene 0 aufhält und die Anwender-Tasks in der Privileg-Ebene 3, geschieht dies durch einen

Taskwechsel. Dabei **scheidet** die OS_Kern_Task **aus** und die ausgewählte Anwender-Task tritt an **ihre Stelle**. Wenn eine Anwender-Task terminiert, dann erfolgt umgekehrt ein Taskwechsel zurück zur OS_Kern_Task, wobei jetzt die betreffende Anwender-Task ausscheidet und die OS_Kern_Task an ihre Stelle tritt.
Der Scheduler hat daraufhin Gelegenheit, eine neue Task auszuwählen und das «Spiel» beginnt von vorne. Der Wechsel von einer Anwender-Task auf eine andere geschieht also immer über die OS_Kern_Task: Sie alleine erzeugt die Anwender-Tasks und wird daher oft auch als **parent**-Task (parent-Prozess) bezeichnet, während die erzeugten Anwender-Tasks **child**-Tasks (child-Prozesse) genannt werden. Im folgenden Bild ist das Schema dieser Taskwechsel-Operationen illustriert:

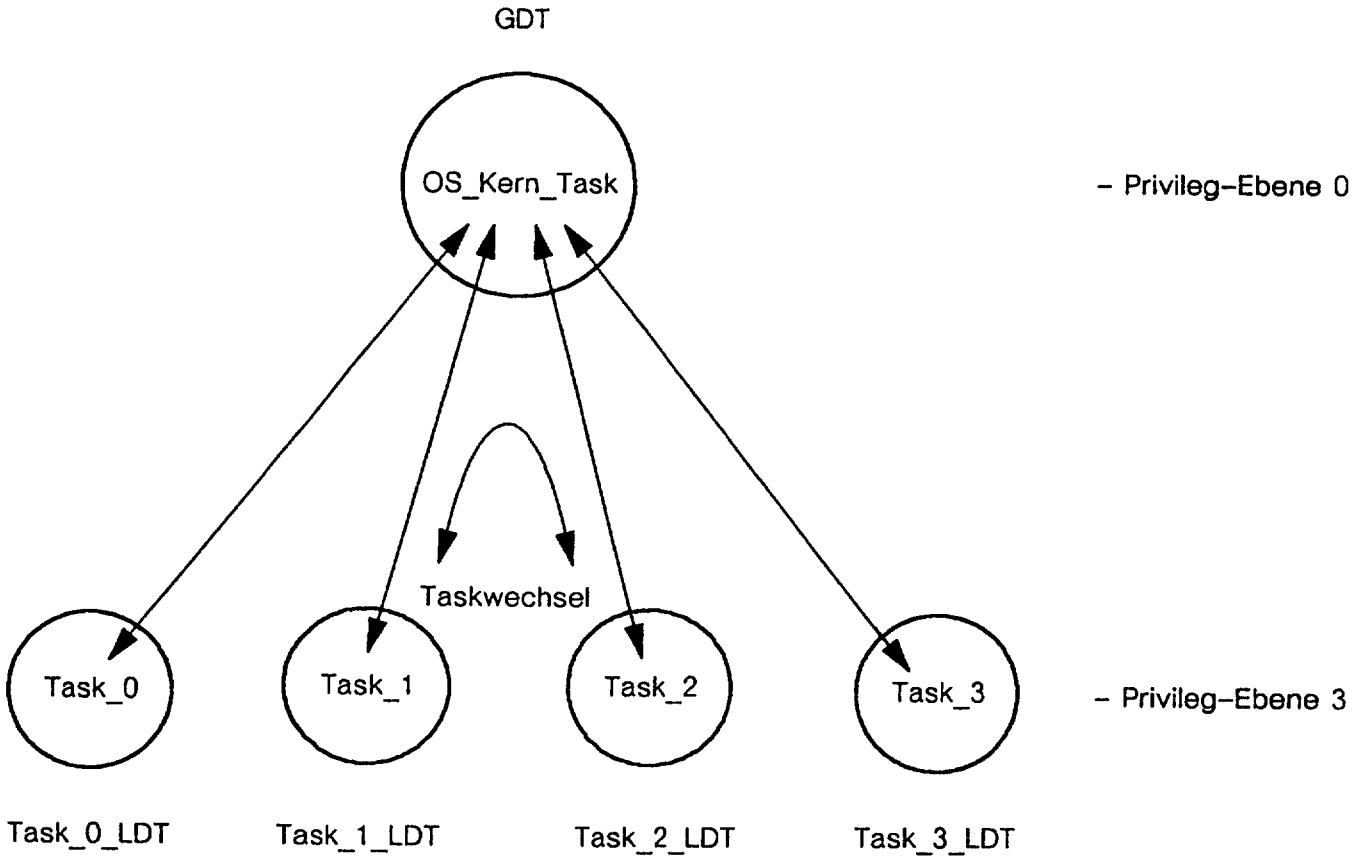


Bild 59: Umschalten von einer Anwender-Task auf eine andere

Die zentrale «Schaltstelle» für einen Wechsel von der OS_Kern_Task zu einer Anwender-Task ist der **Dispatcher**, der ausschließlich mit dem Scheduler korrespondiert. Hat der Scheduler eine betriebsbereite Anwender-Task ausgewählt, dann speichert er die **Nummer** dieser Task in der globalen Privileg-0-Variablen

- `aktuell_task_id` (siehe Bild 41).

Abhängig vom Wert dieser Variablen, wählt dann der Dispatcher einen der **TSS-Deskriptoren** aus und realisiert mit seiner Hilfe den Taskwechsel. Die Betriebssystem-Funktion dispatcher ist im folgenden Bild 60 zu sehen:

```
extern void far task_0(void);
extern void far task_1(void);
extern void far task_2(void);
extern void far task_3(void);

extern char aktuell_task_id;

void dispatcher(void)
{
    switch(aktuell_task_id)
  {
    case  0:
            task_0();       /*Wechsel nach Task 0 ueber TSS-Deskriptor*/
            break;
    case  1:
            task_1();       /*Wechsel nach Task 1 ueber TSS-Deskriptor*/
            break;
    case  2:
            task_2();       /*Wechsel nach Task 2 ueber TSS-Deskriptor*/
            break;
    case  3:
            task_3();       /*Wechsel nach Task 3 ueber TSS-Deskriptor*/
            break;

    default: break;
   }
}
```

Bild 60: Betriebssystem-Funktion **dispatcher** (Datei dispatch.c)

Die TSS-Deskriptoren sind zusammen mit den Task-Status-Segmenten im **Build**-Programm **extern** spezifiziert und werden vom Dispatcher daher über die **FAR**-Calls (16-Bit-Selektor, 32-Bit-Offset)

- `task_0(); task_1(); task_2(); und task_3();`

erreicht.

Auf welche Weise nun die x86/Pentium-Architektur den Taskwechsel unterstützt, illustriert das folgende Bild 61. Es sei aktuell_task_id = 0, und so führt der Dispatcher den FAR-Call task_0(); aus, worauf ein Taskwechsel von der OS_Kern_Task zur Task_0 erfolgt.

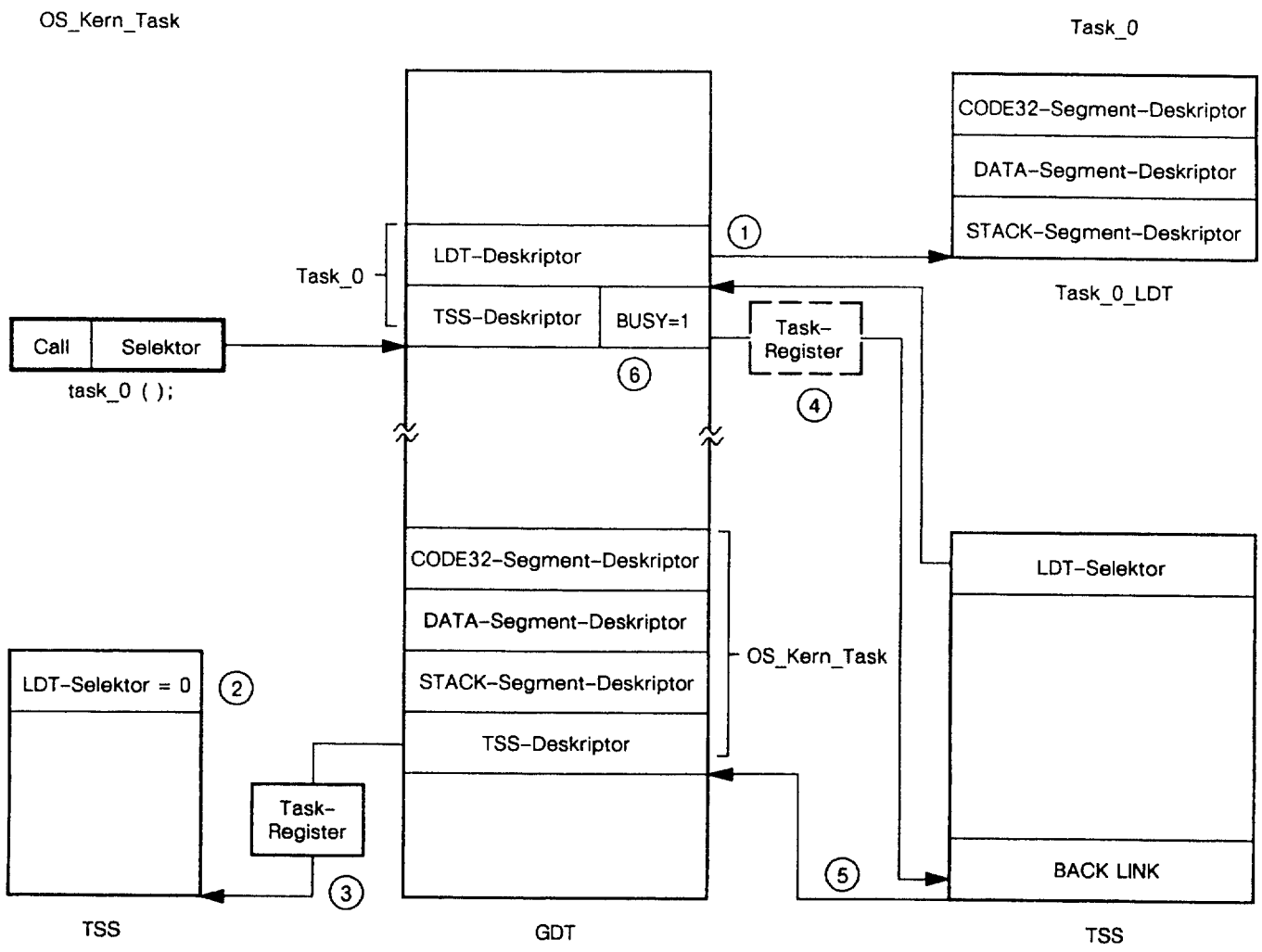


Bild 61: Taskwechsel von der OS_Kern_Task zur Task_0

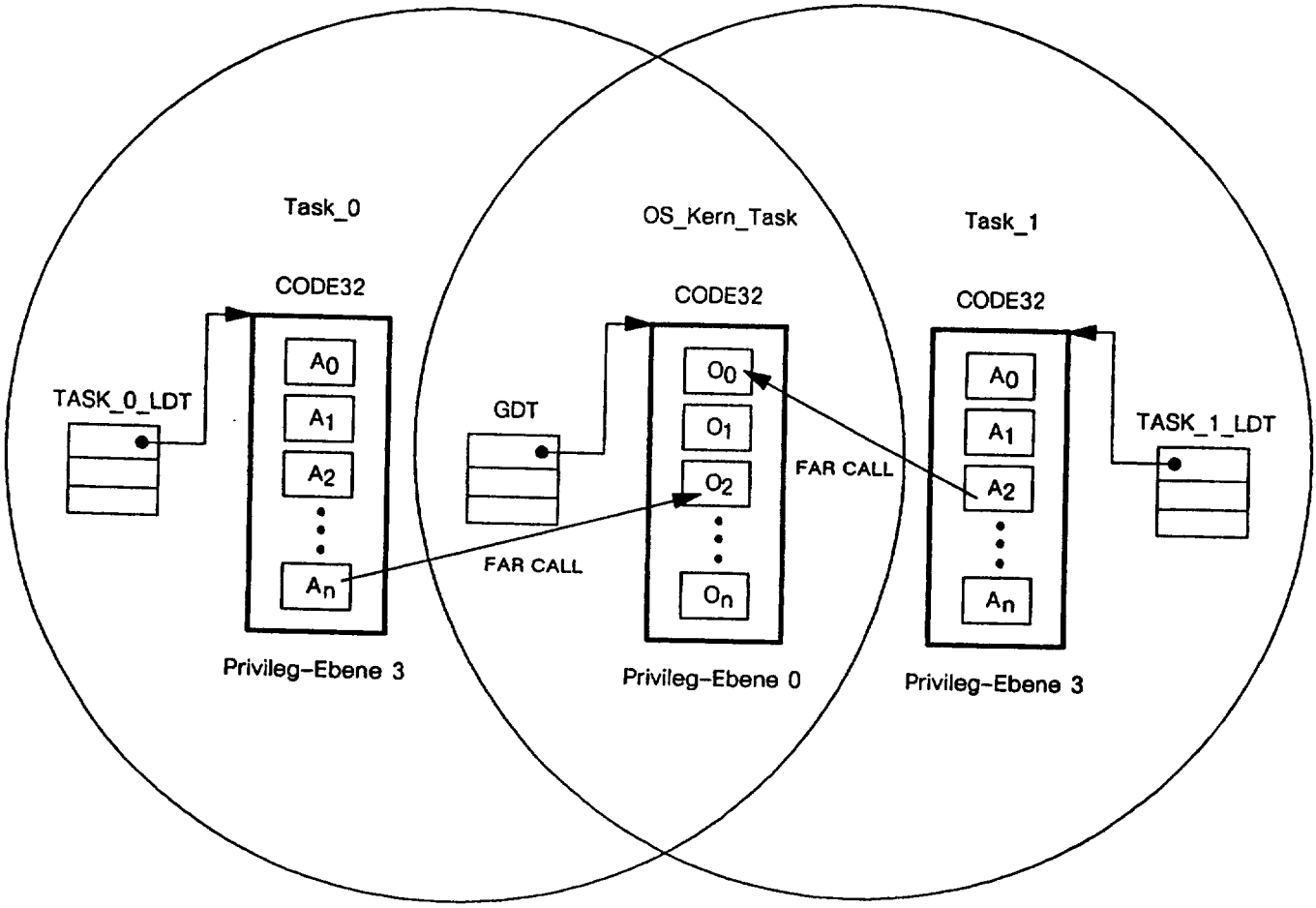
Beide Tasks werden durch ihre Task-Status-Segmente (TSS) beschrieben. Der Adreßraum der OS_Kern_Task ist durch die globale Deskriptor-Tabelle (GDT) definiert und enthält die Deskriptoren der **OS_Kern_Task-Segmente** CODE32, DATA und STACK. Der Adreßraum der Task_0 ist durch die lokale Deskriptor-Tabelle (LDT) mit dem Namen Task_0_LDT definiert. Sie enthält die Deskriptoren der **Task_0-Segmente** ebenfalls mit den Namen CODE32, DATA und STACK. Sowohl die TSS-Deskriptoren der beiden Tasks, als auch der Task_0_LDT-Deskriptor befinden sich in der GDT. Dabei wird der LDT-Deskriptor der Task_0 durch das LDT-Selektor-Feld des zugehörigen Task-Status-Segments ausgewählt, siehe (1) in Bild 61. Weil der Adreßraum der OS_Kern_Task von der GDT bestimmt wird, enthält das LDT-Feld im zugehörigen Task-Status-Segment den Wert **0**, siehe (2).

Zunächst sei die OS_Kern_Task **aktiv**. Ihr TSS-Deskriptor befindet sich deshalb im Task-Status-Segment der CPU, siehe (3). Wenn jetzt der Selektor im Funktionsaufruf **task_();** den TSS-Deskriptor der Task_0 auswählt, dann wird das Task-Status-**Register** (4) der CPU **umgeladen**. Es bekommt den TSS-Deskriptor der Task_0 und zeigt damit auf das Task-Status-Segment der **neuen** Task. Die CPU realisiert daraufhin den Taskwechsel, indem sie alle CPU-Register im Task-Status-Segment der **ausscheidenden** OS_Kern_Task zwischenspeichert und danach den Inhalt des Task-Status-Segments der **eintretenden** Task_0 in die CPU-Register überträgt. Zusätzlich initialisiert die CPU das **BACK LINK**-Feld im Task-Status-Segment der Task_0 mit einem Selektor, der zum

Task-Status-Segment der ausgeschiedenen OS_Kern_Task zeigt (5). Diese Verbindung nutzt der Prozessor, wenn er umgekehrt mit einem **IRETD**-Befehl von der Task_0 in die OS_Kern_Task zurückschaltet.
Durch den Wechsel von der OS_Kern_Task zur Task_0 hat die Task_0 den Zustand **bereit** (ready) verlassen und ist übergegangen in den Zustand **laufend** (running bzw. busy). Der Prozessor kennzeichnet diesen Zustand, indem er das **Busy**-Bit im Access-Byte ihres TSS-Deskriptors setzt (Busy-Bit = 1), siehe (6).

1.9.2 Die Betriebssystem-Prozedur task_wechsel

Alle Funktionen $\mathbf{A_x}$ der individuellen Anwender-Tasks sind im **CODE32**-Segment der betreffenden Task zusammengefaßt. Das gleiche gilt für die Funktionen $\mathbf{O_i}$ des Betriebssystemkerns. Sie sind im **CODE32**-Segment der OS_Kern_Task enthalten. Wenn eine Anwender-Funktion A_x eine der **gemeinsam** benutzbaren Betriebssystemkern-Funktionen O_i aufruft, dann befindet sich die zugehörige Anwender-Task **nach** dem Privilegwechsel nicht mehr im User-Bereich der Privileg-Ebene 3, sondern im Betriebssystemkern-Bereich der Privileg-Ebene 0. Sie hat sich die Betriebssystemkern-Funktion gewissermaßen «einverleibt». Im folgenden Bild 62 ist dies am Beispiel der beiden Anwender-Tasks Task_0 und Task_1 illustriert:



Anwender-Funktionen der Task_0: A_j (j = 0..n)
Anwender-Funktionen der Task_1: A_k (k = 0..n)
Betriebssystemkern-Funktionen der OS_Kern_Task: O_i (i = 0..n)

Bild 62: Anwender-Tasks führen Betriebssystem-Funktionen aus

Manche der Betriebssytemkern-Funktionen entscheiden nun darüber, ob die augenblicklich aktive Anwender-Task suspendiert wird und damit pausieren muß oder nicht.

Wird sie suspendiert, dann sorgt die Betriebssystemkern-Funktion als erstes dafür, daß der **Task-Control-Block** der betreffenden Anwender-Task aus der Task-Queue **ausgekettet** wird. Zu diesem Zweck aktiviert sie die Betriebssystem-Funktion **ausketten**. Danach erfolgt dann über den Assembler-Befehl **IRETD** ein Taskwechsel zurück zur OS_Kern_Task. Das folgende Bild 63 zeigt hierfür ein Beispiel:

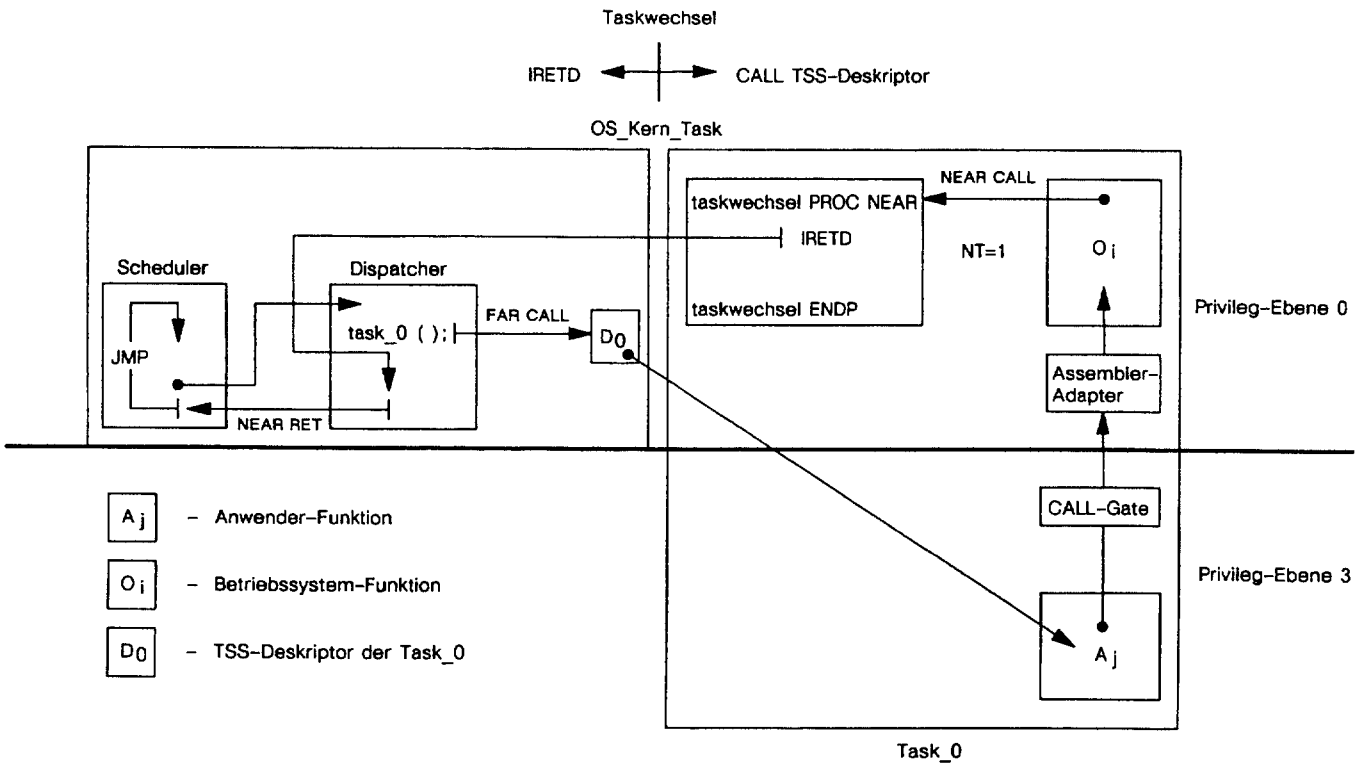


Bild 63: Taskwechsel-Operationen zwischen OS_Kern_Task und Task_0

Nachdem der Dispatcher über den TSS-Deskriptor der Task_0 einen Taskwechsel veranlaßt hat, setzt der Prozessor das **NT**-Bit (Nested Task) in den **EFLAGs** und markiert damit den BACK LINK-Selektor im Task-Status-Segment der Task_0 als **gültig**. Im Bild ist nun zu sehen, daß die Anwender-Funktion A_j in der Privileg-Ebene 3 über einen Call-Gate-Deskriptor und einen passenden Assembler-Adapter die Betriebssystem-Funktion O_i aufruft. Die Task_0 befindet sich jetzt in der Privileg-Ebene 0 und führt dort O_i aus. Wenn sich O_i für einen Taskwechsel entscheidet, dann ruft sie die Assembler-NEAR-Prozedur **task_wechsel** auf. Diese Betriebssystem-Prozedur ist die zentrale «**Schaltstelle**» für den Wechsel von einer **beliebigen** Anwender-Task (im Beispiel Task_0) zur OS_Kern_Task.

Den Assembler-Code der Prozedur task_wechsel zeigt Bild 64:

```
        NAME task_wechsel

PUBLIC task_wechsel

CODE32 SEGMENT ER PUBLIC

task_wechsel PROC NEAR

             iretd            ;Taskwechsel
             ret              ;Zurueck zum Aufrufer

task_wechsel ENDP

CODE32 ENDS
      END
```

Bild 64: Betriebssystem-Prozedur task_wechsel (Datei taskwech.src)

Weil **NT = l** ist, schaltet **IRETD** von der Anwender-Task zurück zur OS_Kern_Task und löscht gleichzeitig das **Busy-Bit** (Busy = 0) im TSS-Deskriptor der ausscheidenden Task. Die betreffende Anwender-Task **verläßt** den Zustand laufend, und die OS_Kem_Task tritt an ihre Stelle. Sie befindet sich nach dem Taskwechsel beim ersten Befehl hinter task_0(); (oder task_l(); oder task_2(); oder task_3();) im Dispatcher-Code (siehe Bild 63). Von dort erfolgt die Rückkehr zum Scheduler, der daraufhin seine Arbeit aufnimmt und in den Task-Queue-Köpfen nach einer neuen betriebsbereiten Anwender-Task sucht.
Die Taskwechsel finden also an **zwei** zentralen «Schaltstellen» statt:

- Die eine ist der **Dispatcher**: Sie sorgt für den Wechsel von der OS_Kem_Task zu einer Anwender-Task.

- Die andere ist die Betriebssystem-Prozedur **task_wechsel**. Sie sorgt umgekehrt für den Wechsel von einer Anwender-Task zur OS_Kem_Task. Siehe Bild 65:

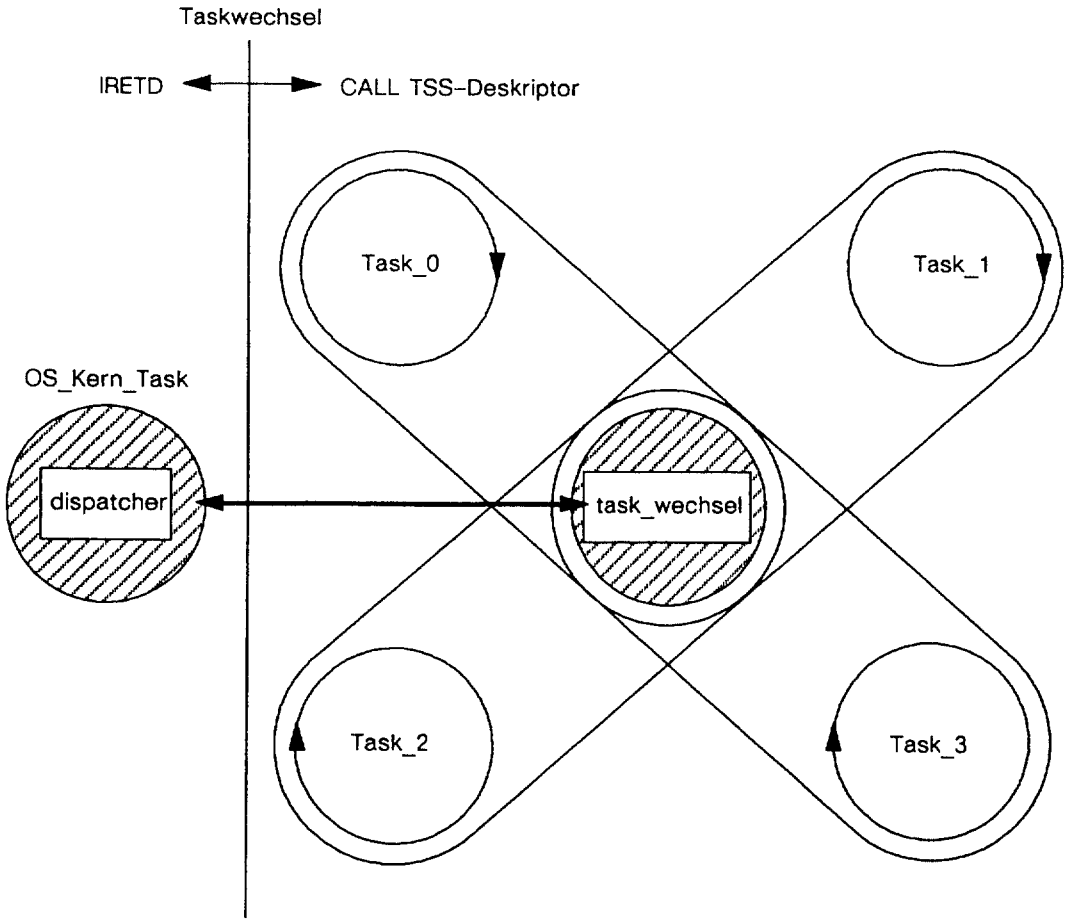


Bild 65: Taskwechsel-Schaltstellen

Hat eine Betriebssystem-Funktion eine Anwender-Task suspendiert, dann entscheidet eine andere darüber, ob sie wieder aktiviert wird oder nicht. Wenn ja, dann sorgt die betreffende Betriebssystem-Funktion als erstes dafür, daß der zugehörige Task-Control-Block in die Task-Queue **eingekettet** wird. Sie ruft zu diesem Zweck die Betriebssystem-Funktion **einketten** auf.
Die Anwender-Task bewirbt sich jetzt wieder zusammen mit den anderen Tasks beim Scheduler um den Prozessor. Wird sie ausgewählt, dann erfolgt über den Dispatcher der Taskwechsel, und zwar zum **ersten** Befehl hinter **IRETD** in der Betriebssystem-Prozedur task_wechsel. Die betreffende Anwender-Task befindet sich also wieder in der Privileg-Ebene 0 und führt dort den Befehl **ret** aus. Siehe Bild 66. Er veranlaßt die Rückkehr zur Betriebssystem-Funktion O_i, siehe (1). Von dort erfolgt die Rückkehr zum Assembler-Adapter, siehe (2) und von dort die Rückkehr zur Anwender-Funktion A_j in der Privileg-Ebene 3, siehe (3).

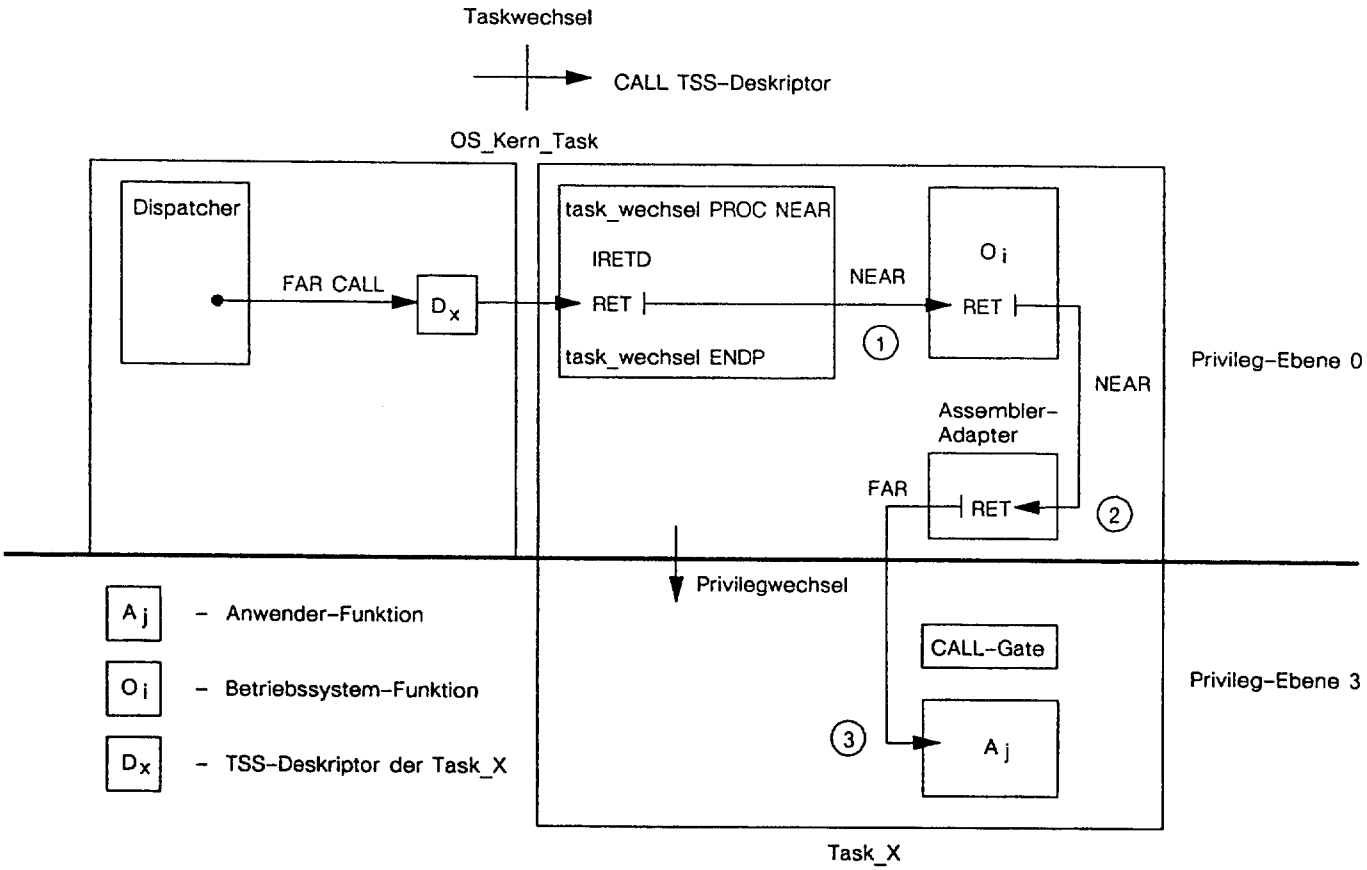


Bild 66: Rückkehr zur Anwender-Funktion in der Privileg-Ebene 3

1.10 Stackbereiche der Anwender-Tasks in der Privileg-Ebene 0

Immer dann, wenn eine Anwender-Task über einen Call-Gate-Deskriptor eine Betriebssystem-Funktion O_i aufruft, bewirkt der Privilegwechsel, daß automatisch auch die Stacksegmente gewechselt werden.

In der Privileg-Ebene 0 benutzen **alle** Anwender-Tasks den Stack der OS_Kern_Task. Den Beginn der Stack-Arbeitsbereiche (Top of Stack) in der Ebene 0 bestimmen die SS0:ESP0-Felder der individuellen Task-Status-Segmente. Diese Felder initialisiert der **System-Builder** in allen Task-Status-Segmenten mit **identischen** Zeigerwerten. Warum das so ist, zeigen im folgenden Beispiel die Task-Definitionen von vier Anwender-Tasks Task_0..Task_3 (Bild 67):

```
Task_0  (DPL=0,PRESENT,
         CODE=start_0_proc,
         DATA=Anw_task_0.DATA,
         STACKS=(Anw_task_0.STACK,
                 OS_Kern.STACK),
         LDT=TASK_0_LDT,                 --LDT-Selektor der Task_0
         IOPRIVILEGE=0,                  --IOPL=0
         INTENABLED),                    --Interrupt zugelassen

Task_1  (DPL=0,PRESENT,
         CODE=start_1_proc,
         DATA=Anw_task_1.DATA,
         STACKS=(Anw_task_1.STACK,
                 OS_Kern.STACK),
         LDT=TASK_1_LDT,                 --LDT-Selektor der Task_1
         IOPRIVILEGE=0,                  --IOPL=0
         INTENABLED),                    --Interrupt zugelassen

Task_2  (DPL=0,PRESENT,
         CODE=start_2_proc,
         DATA=Anw_task_2.DATA,
         STACKS=(Anw_task_2.STACK,
                 OS_Kern.STACK),
         LDT=TASK_2_LDT,                 --LDT-Selektor der Task_2
         IOPRIVILEGE=0,                  --IOPL=0
         INTENABLED),                    --Interrupt zugelassen

Task_3  (DPL=0,PRESENT,
         CODE=start_3_proc,
         DATA=Anw_task_3.DATA,
         STACKS=(Anw_task_3.STACK,
                 OS_Kern.STACK),
         LDT=TASK_3_LDT,                 --LDT-Selektor der Task_3
         IOPRIVILEGE=0,                  --IOPL=0
         INTENABLED);                    --Interrupt zugelassen
```

Bild 67: Definition von vier Anwender-Tasks mit dem System-Builder

In jeder Anwender-Task werden wegen

- `STACKS = (Anw_task_x.STACK,`
  `OS_Kern.STACK),`

zwei Stacks definiert, und zwar jeweils ein Anwender-Stack in der **Privileg-Ebene 3** und wegen **OS_Kern.STACK** für **alle** Anwender-Tasks ein **gemeinsamer** Stack in der **Privileg-Ebene 0**. Mit anderen Worten, die SS0:ESP0-Zeiger in den vier Anwender-Task-Status-Segmenten sind **identisch**.

Findet ein Privilegwechsel statt, dann lädt der Prozessor die Register SS und ESP mit den Werten der SS0:ESP0-Felder. Das Problem liegt auf der Hand, denn alle Anwender-Tasks beginnen nach dem Privilegwechsel bei der **gleichen** Stack-Anfangsposition. Dies ist im folgenden Bild 68 verdeutlicht:

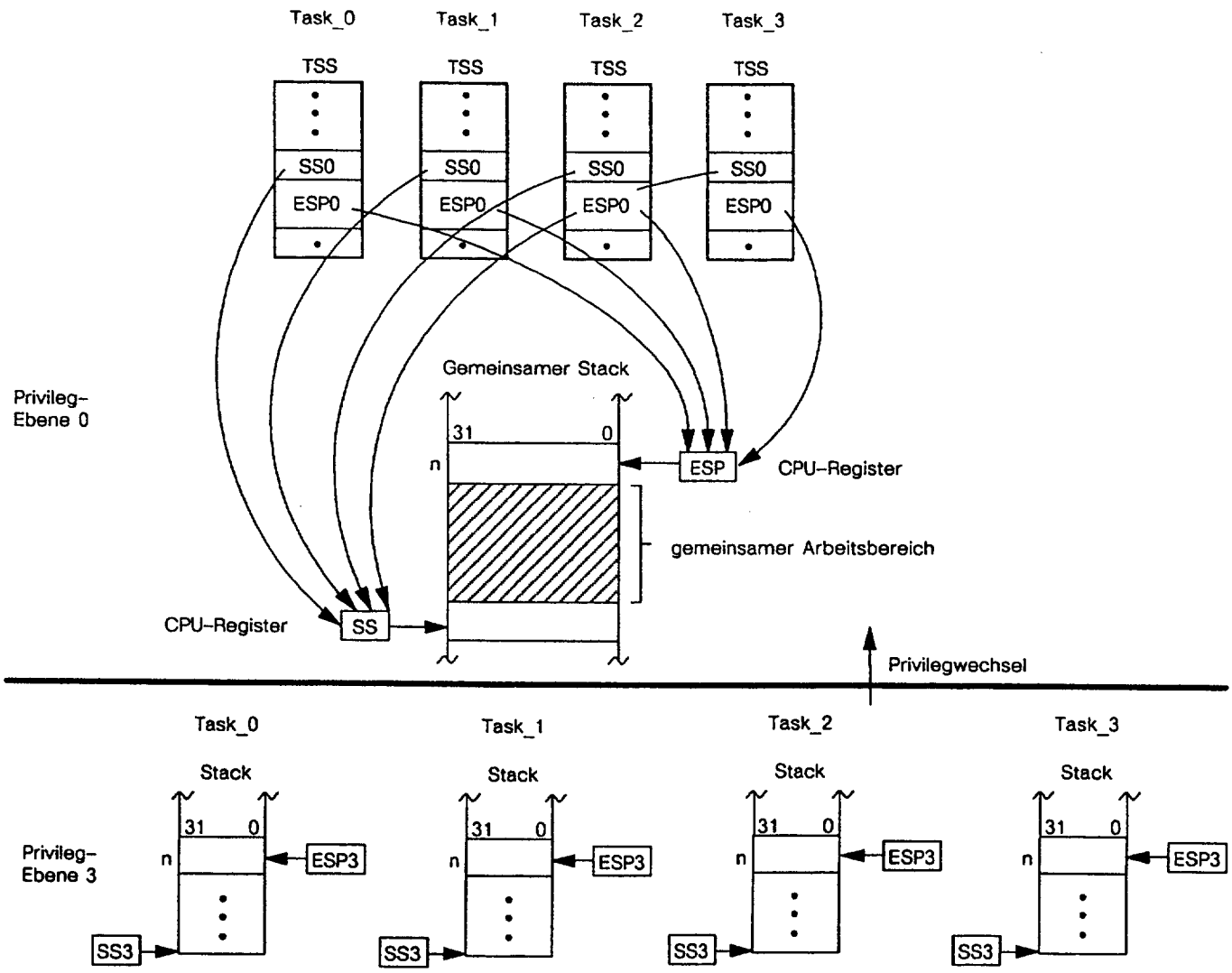


Bild 68: Gemeinsamer Stack-Arbeitsbereich aller Tasks in der Privileg-Ebene 0

Scheidet eine Anwender-Task aus, dann enthält der gemeinsame Stack-Arbeitsbereich in der Ebene 0 noch immer aktuelle Daten der betreffenden Task, nämlich die Adresse **SS3:ESP3** des Anwender-Stacks der Ebene 3, eventuell **Parameter** und die Rücksprung-Adresse **CS3:EIP3**. Wenn bei dieser Situation eine **neue** Anwender-Task eine Betriebssystemkern-Funktion aufruft und daher einen Privilegwechsel durchführt, dann beginnt sie in der Privileg-Ebene 0 bei der **gleichen** Stack-**Anfangs**position SS0:ESP0 wie die vorher ausgeschiedene Task. Der Prozessor speichert fatalerweise ab dieser Position die üblichen Daten, also die Adresse SS3:ESP3 des Anwender-Stacks der **neuen** Task, eventuell Parameter und die Rücksprung-Adresse CS3:EIP3. Die Folge ist, daß die Stack-Daten der ausgeschiedenen Task «verloren» sind. Das Problem kann aber vermieden werden, wenn für jede Anwender-Task im gemeinsamen Ebene-0-Stack ein **eigener** lokaler Arbeitsbereich reserviert ist. Und dies wird erreicht, wenn die Anwender-Tasks an **unterschiedlichen Anfangs**positionen SS0:ESP0 beginnen. Im Bild 69 ist dies illustriert:

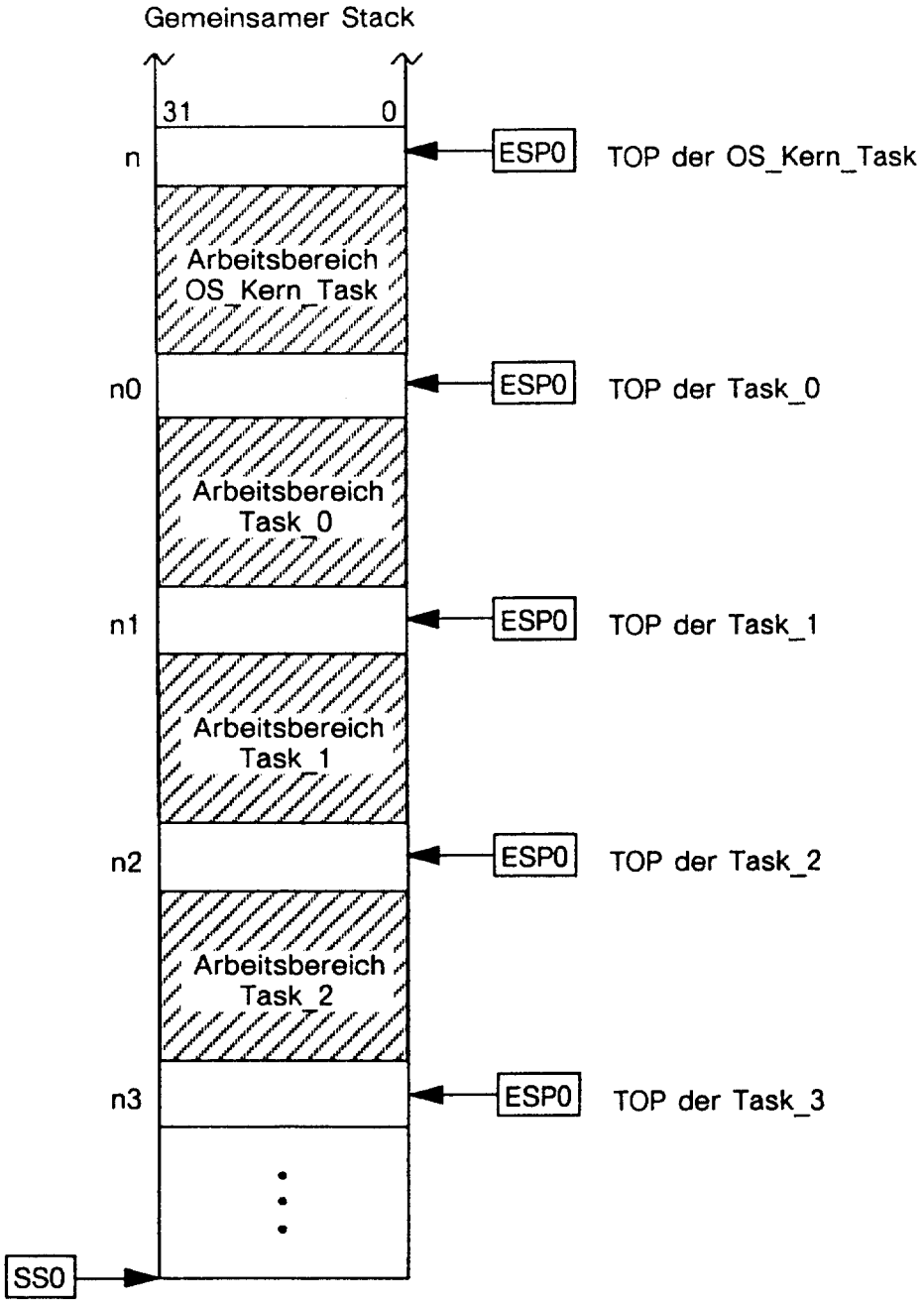


Bild 69: Individuelle Arbeitsbereiche der Tasks im gemeinsamen Stack der Privileg-Ebene 0

In einem Multitasking-System mit z. B. vier Anwender-Tasks ist der gemeinsame Stack in insgesamt fünf Arbeitsbereiche eingeteilt:

- ein Bereich für die OS_Kern_Task und
- vier Bereiche für die Anwender-Tasks.

Auf welche Weise kann nun eine solche Aufteilung realisiert werden? Wir wissen, wenn eine Anwender-Task einen Privilegwechsel durchführt, dann werden die Anfangspositionen des gemeinsamen Stack von den SS0:ESP0-Feldern der betreffenden Task-Status-Segmente bestimmt.
So müssen diese Felder, die zunächst der System Builder definiert, mit passenden Werten überschrieben werden, wobei SS0 unberührt bleiben kann.

Für diese Aufgabe bietet sich die Betriebssystem-Funktion **task_init** an, denn sie wird ja benutzt, um neue Anwender-Tasks einzuführen. Das vollständige C-Programm von task_init zeigt Bild 70:

```
#define task_max       4
#define prioritaet_max 4
#include<struct.h>

extern task_queue_kopf_struct task_queue_kopf[prioritaet_max];
extern task_cb_struct task_cb[task_max];

static union
{
 unsigned int far *zeiger_auf_tss;
 struct {
        unsigned int ESP0_offset;
        unsigned short tss_alias_deskriptor_offset;
       }s1;
}u1;

void task_init(char task_id,char task_status,char task_prioritaet)
{
 task_queue_kopf_struct *task_queue_kopf_adr;
 task_cb_struct *neu_task_cb_adr;

 task_cb[task_id].task_nummer     = task_id;
 task_cb[task_id].task_status     = task_status;
 task_cb[task_id].task_prioritaet = task_prioritaet;

 task_queue_kopf_adr = &task_queue_kopf[task_cb[task_id].task_prioritaet];
 neu_task_cb_adr = &task_cb[task_id];
 einketten(task_queue_kopf_adr,neu_task_cb_adr);

 u1.s1.ESP0_offset=0x00000004;                  /*Offset des ESP0-Registers in*/
                                                /*den Task-Status-Segmenten*/
 switch(task_id)
 {
  case 0:
  u1.s1.tss_alias_deskriptor_offset=0x0080;     /*Offset des TSS-Alias-Daten-*/
                                                /*Segments der Task 0*/
  *(u1.zeiger_auf_tss)=0x01000;                 /*ESP0-Register im TSS der*/
  break;                                        /*Task 0=1000H*/

  case 1:
  u1.s1.tss_alias_deskriptor_offset=0x0088;     /*Offset des TSS-Alias-Daten-*/
                                                /*Segments der Task 1*/
  *(u1.zeiger_auf_tss)=0x0800;                  /*ESP0-Register im TSS der*/
  break;                                        /*Task 1=800Hex*/
```

```
case 2:
  u1.s1.tss_alias_deskriptor_offset=0x0090;     /*Offset des TSS-Alias-Daten-*/
                                                /*Segments der Task 2*/
  *(u1.zeiger_auf_tss)=0x0600;                  /*ESP0-Register im TSS der*/
  break;                                        /*Task 2=600Hex*/

  case 3:
  u1.s1.tss_alias_deskriptor_offset=0x0098;     /*Offset des TSS-Alias-Daten-*/
                                                /*Segments der Task 3*/
  *(u1.zeiger_auf_tss)=0x0400;                  /*ESP0-Register im TSS der*/
  break;                                        /*Task 3=400Hex*/
  default: break;
 }
}
```

Bild 70: Vollständige Betriebssystem-Funktion task_init (Datei task_ini.c)

Damit task_init auf die ESP0-Felder der Task-Status-Segmente zugreifen kann, muß in der globalen Deskriptor-Tabelle (GDT) für jedes Task-Status-Segment ein **Alias**-Deskriptor existieren, der diese sogenannten **System**-Segmente als schreib- und lesbare **Daten**-Segmente bzw. **Alias**-Segmente definiert. Mit Hilfe des System Builders können solche Alias-Deskriptoren vereinbart werden. Siehe Bild 71:

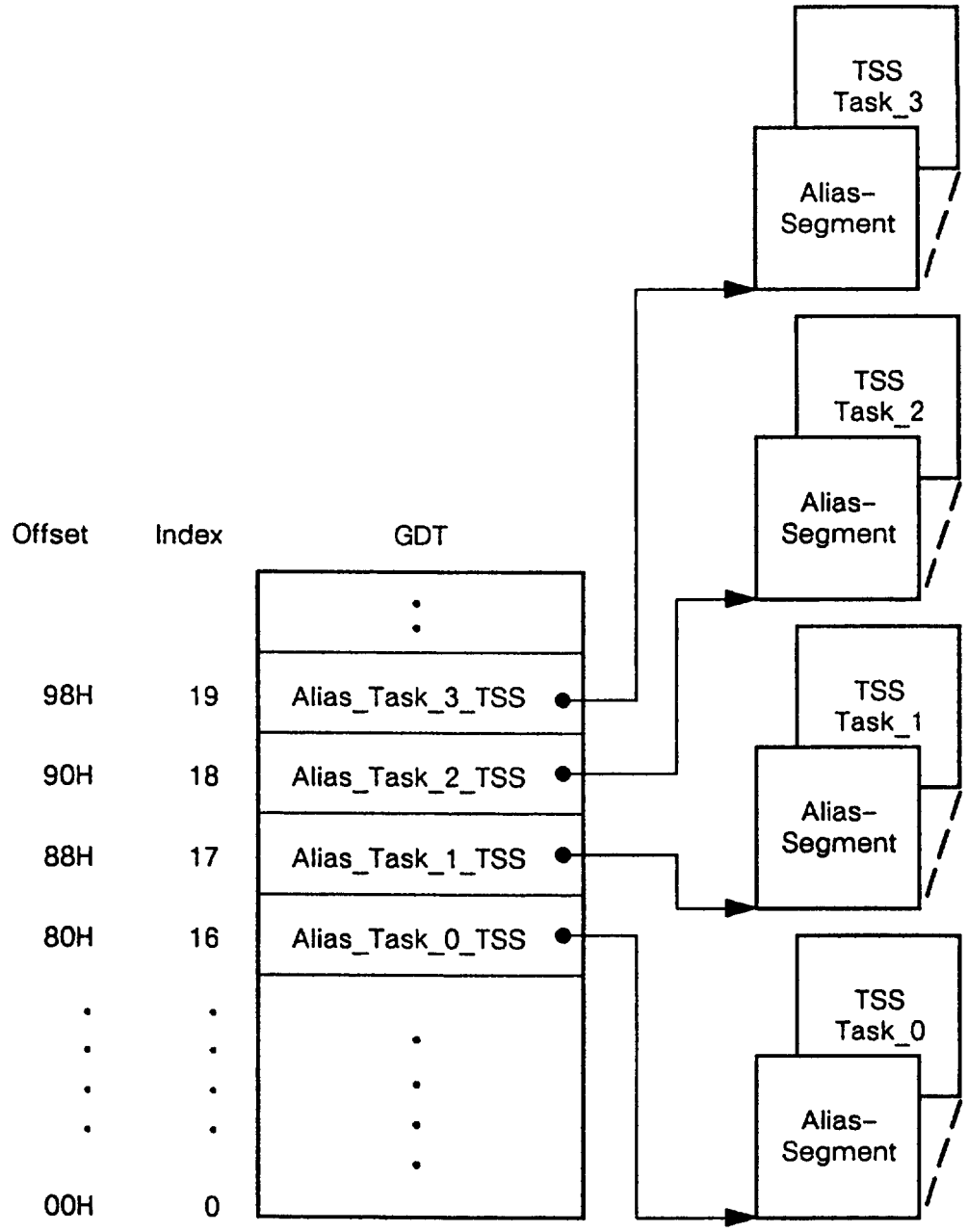


Bild 71: Definition der Anwender-Task-Status-Segmente als Alias-Segmente

Sie definieren, wie beabsichtigt, die vier Anwender-Task-Status-Segmente als Daten-bzw. Alias-Segmente und liegen in der **GDT** z. B. bei Index 16..19 bzw. Offset 80H..98H.
Grundlage für den Zugriff auf die ESP0-Felder ist die **union**-Variable **u1**, die folgendermaßen definiert ist:

```
static union
{
 unsigned int far *zeiger_auf_tss;
 struct {
        unsigned int ESP0_offset;
        unsigned short tss_alias_deskriptor_offset;
       }s1;
}u1;
```

Sie besteht aus zwei Komponenten, und zwar

- dem **far**-Zeiger (16-Bit-Selektor, 32-Bit-Offset) vom Typ **int** (32 Bits) und
- der Strukturvariablen **s1**.

Dabei ist s1 aus der **int**-Variablen ESP0_offset (32 Bits) und der **short int**-Variablen tss_alias_deskriptor_offset (16 Bits) zusammengesetzt. Das Layout von u1 ist im folgenden Bild 72 illustriert:

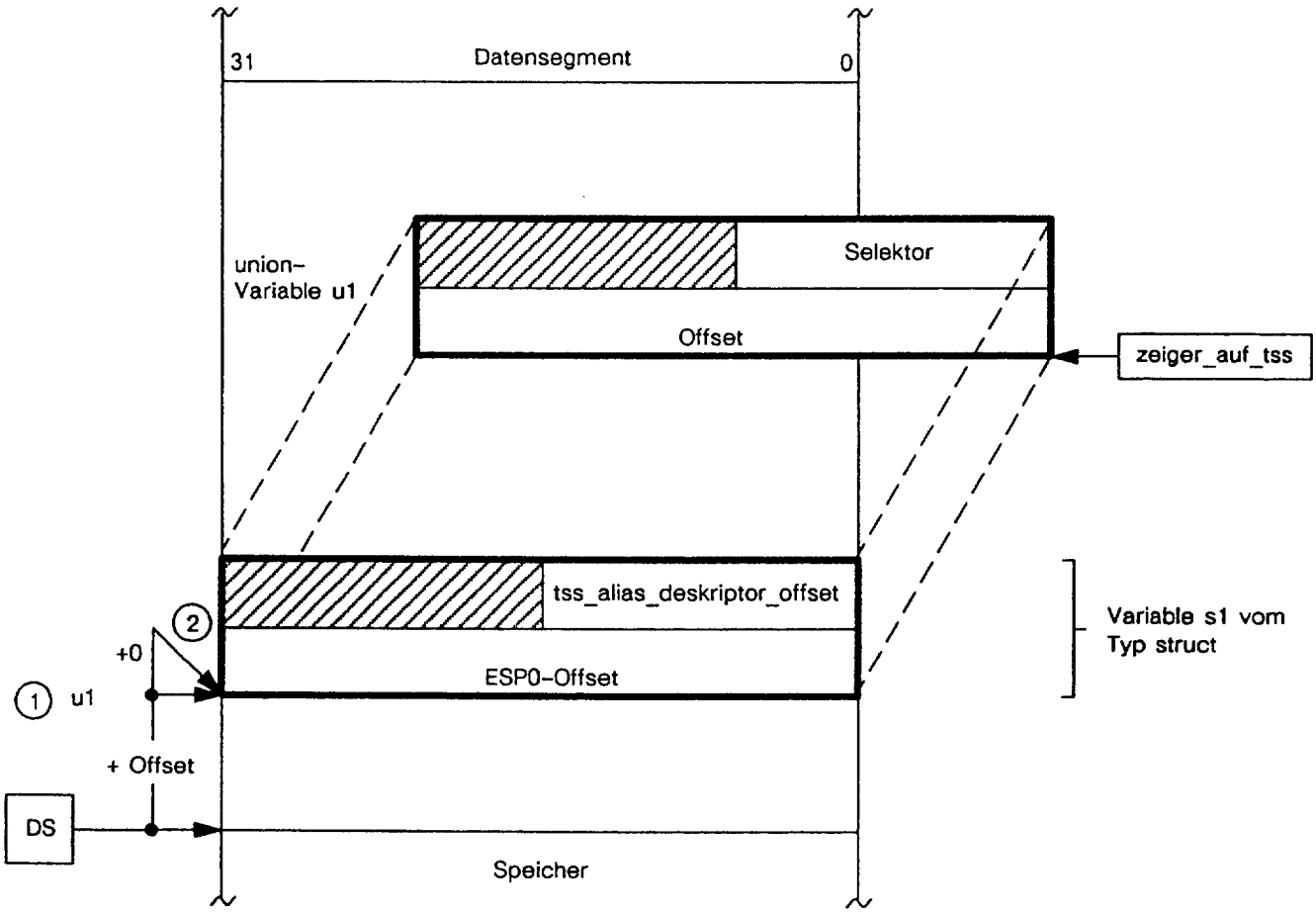


Bild 72: Layout der union-Variablen u1

Wie zu erkennen ist, liegt u1 bei einem bestimmten Offset relativ zur Basisadresse des Datensegments, siehe (1). Die Strukturvariable s1 und der far-Zeiger zeiger_auf_tss sind **überlagert** und liegen daher beide beim Offset +0 relativ zur Anfangsadresse der union-Variablen u1, siehe (2).

Auf welche Weise nun auf die beiden Elemente s1 und zeiger_auf_tss in u1 zugegriffen werden kann, ist im Bild 73 analysiert:

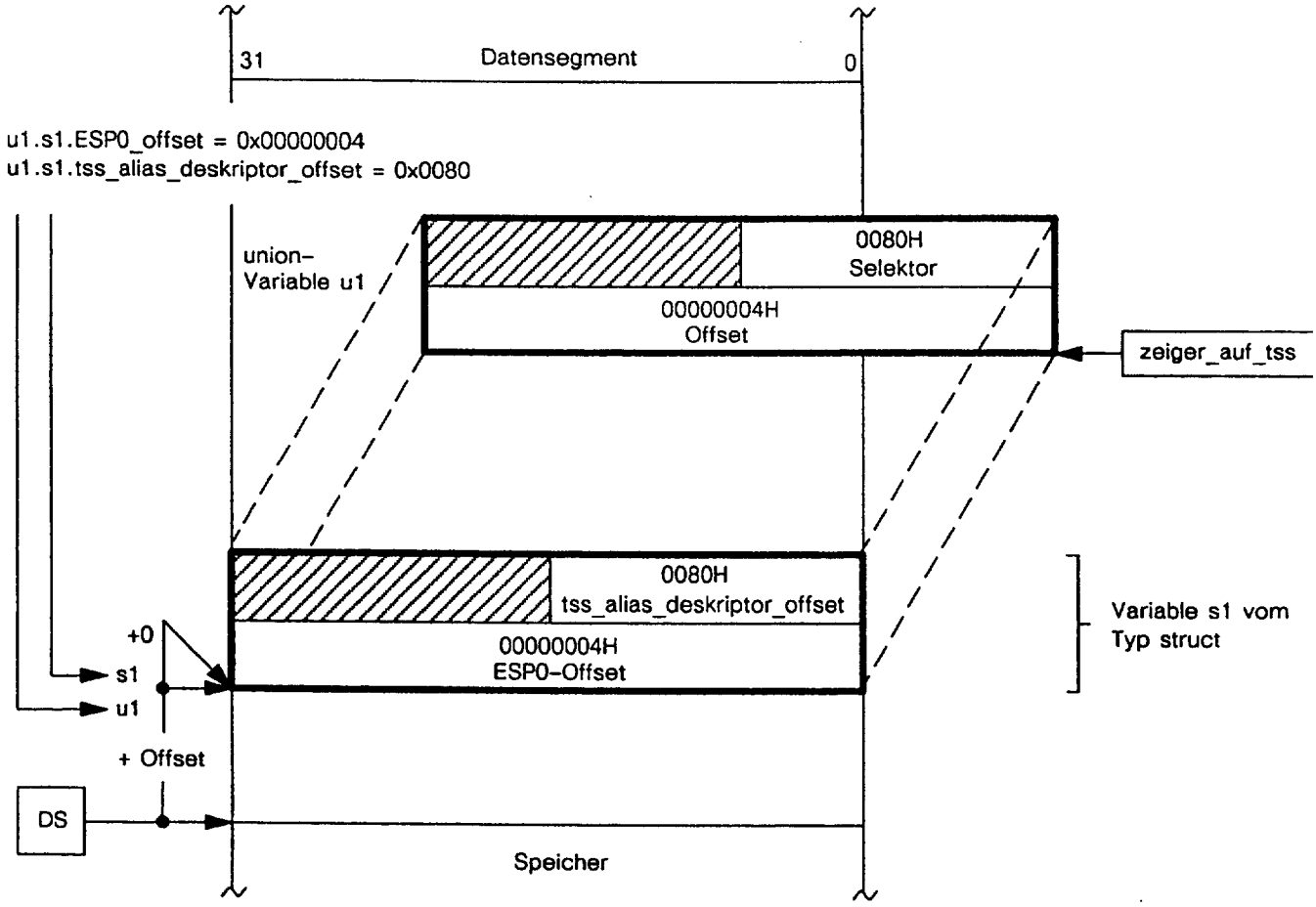


Bild 73: Zugriff auf die Elemente von u1

In den Anweisungen

- `u1.s1.ESP0_offset = 0x00000004;`
  und
- `u1.s1.tss_alias_deskriptor_offset = 0x0080;`

selektiert **u1.** die union-Variable u1 im Datensegment; **s1.** wählt innerhalb von u1 die Strukturvariable s1 aus; und ESP0_offset bzw. tss_alias_deskriptor_offset selektiert die Elemente von s1. Weil s1 mit dem far-Zeiger zeiger_auf_tss überlagert ist, werden sowohl die Elemente von s1 als auch die beiden Komponenten Selektor:Offset des far-Zeigers initialisiert. Die Selektor-Komponente bekommt den Wert 0080H und damit den Offset des Alias-Deskriptors **Alias_Task_0_TSS** (siehe Bild 71). Die Offset-Komponente bekommt den Wert 00000004H und damit den Offset des **ESP0**-Felds im Task-Status-Segment (Alias-Segment) der Task_0 (siehe Bild 8).
Mit Hilfe des far-Zeigers zeiger_auf_tss kann nun auf einfache Weise das ESP0-Feld erreicht und geändert werden. Dies geschieht mit der Anweisung

- `*(u1.zeiger_auf_tss) = 0x01000;`

Diese Anweisung wird im folgenden Bild 74 analysiert:

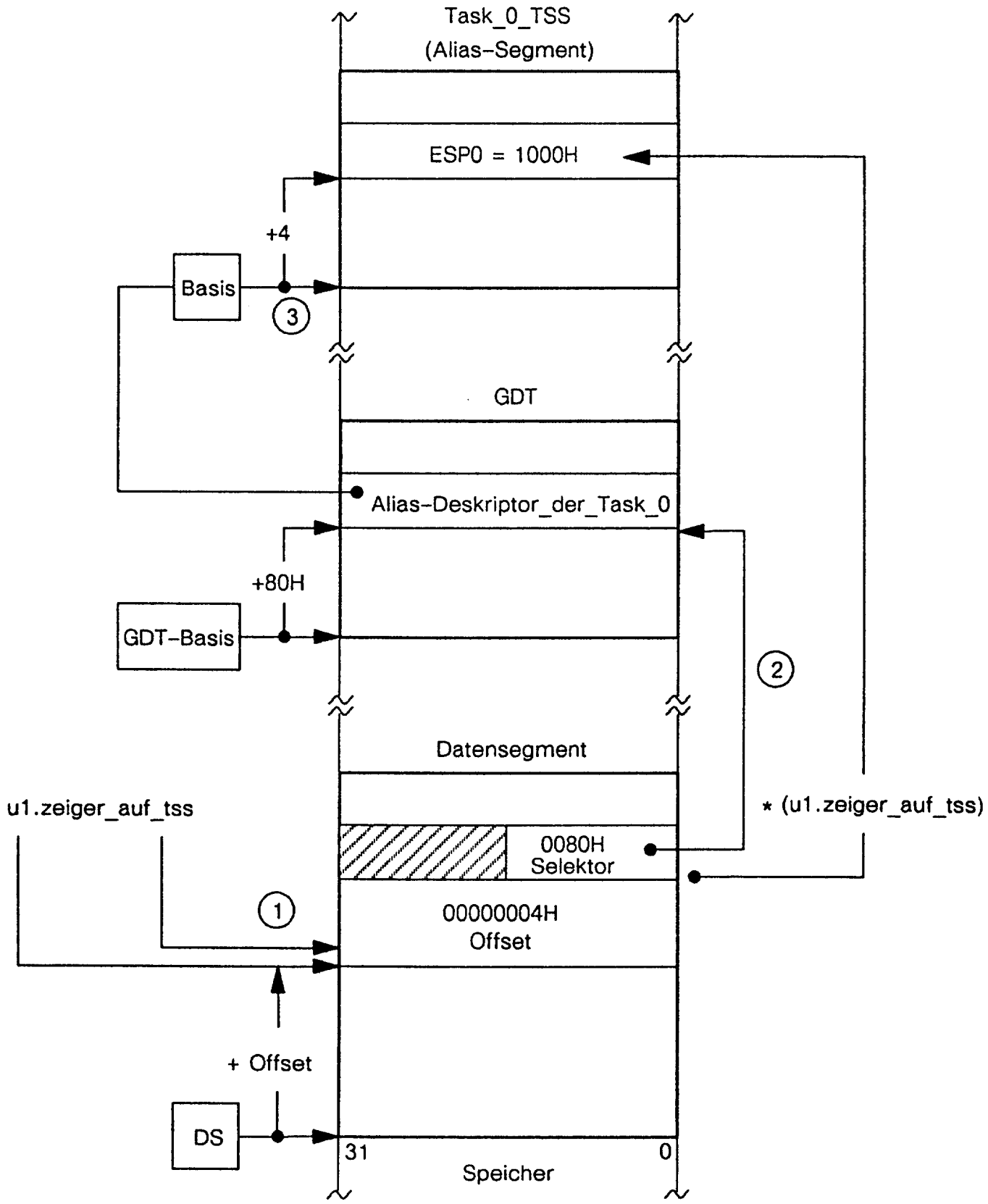


Bild 74: Initialisierung des ESP0-Felds

Zunächst selektiert u1.zeiger_auf_tss den far-Zeiger zeiger_auf_tss in u1, siehe (1). Der Einfachheit halber wurde im Bild 74 auf die Darstellung von s1 verzichtet. Danach wählt die Selektor-Komponente dieses Zeigers (Wert = 0080H) in der GDT den Alias-Deskriptor des Task_0-Status-

Segments aus, siehe (2). Dieser Deskriptor bestimmt die Basisadresse des zugehörigen Alias-Segments, siehe (3).
Relativ zu dieser Basisadresse adressiert nun die Offset-Komponente des far-Zeigers (Wert = 00000004H) das ESP0-Feld. Wegen *(u1.zeiger_auf_tss) = 0x01000 bekommt dann dieses Feld den Wert **1000H**. Die Anfangsposition SS0:ESP0 der Task_0 ist somit bestimmt. Abhängig von **task_id** werden die ESP0-Felder der **restlichen** Task-Status-Segmente auf die gleiche Weise initialisiert. Dabei liegt es in der Verantwortlichkeit des Beriebssystem-Entwicklers, die ESP0-Werte so zu wählen, daß die verfügbaren Stack-Arbeitsbereiche der Anwender-Tasks in der Privileg-Ebene 0 genügend groß sind.

1.11 Round-Robin-Scheduling gleich wichtiger Tasks

Angenommen, alle Anwender-Tasks sind **gleich wichtig** und haben deswegen die **gleiche** Priorität. So ergibt sich z. B. eine Task-Queue-Konfiguration, wie sie das folgende Bild 75 zeigt:

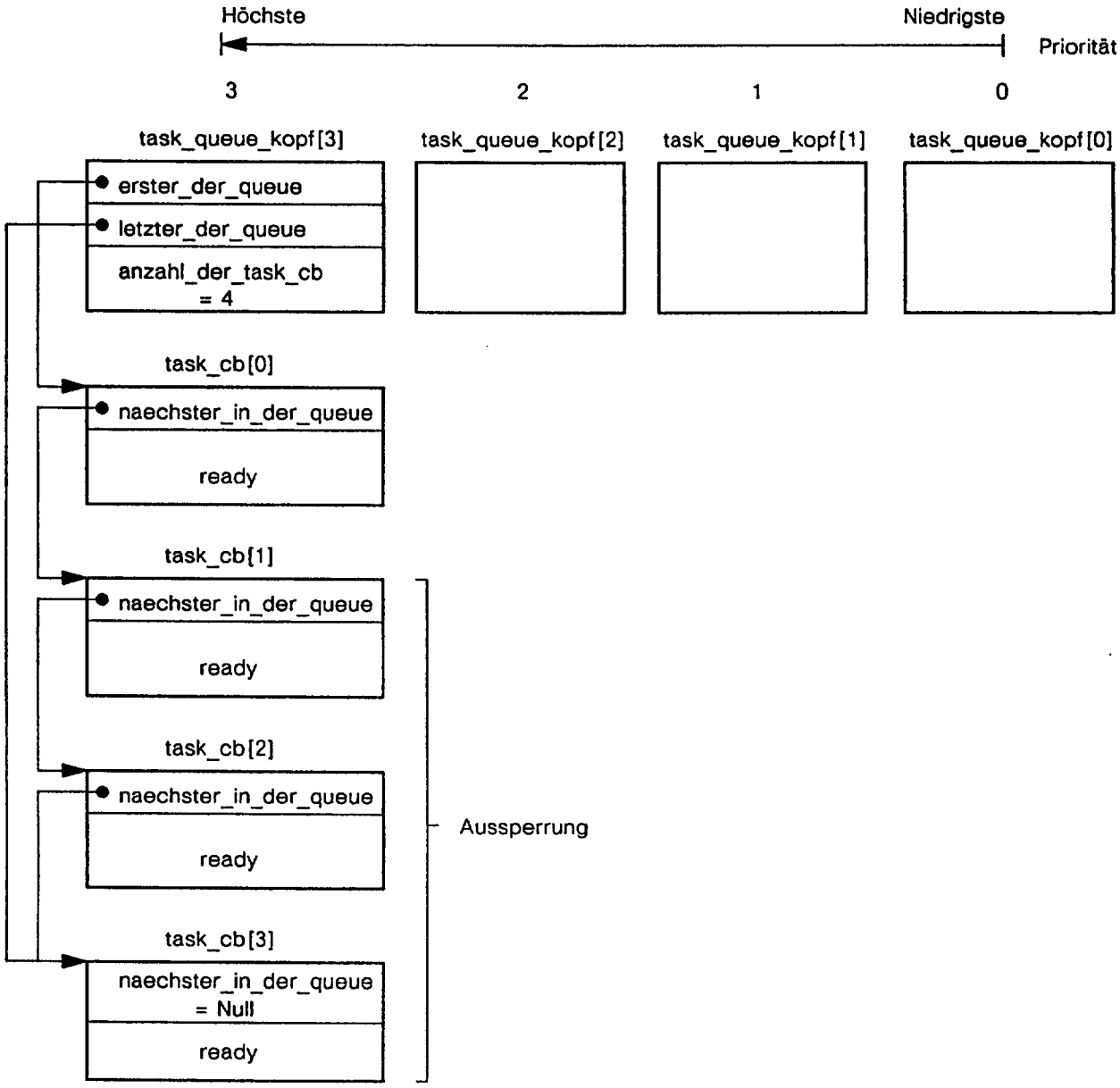


Bild 75: Vier Anwender-Tasks mit gleicher Wichtigkeit

Hier haben alle Anwender-Tasks die **höchste** Priorität, nämlich 3. Die Task-Queue dieser Prioritätsstufe ist komplett aufgefüllt, während die Task-Queues der verbleibenden Prioritätsstufen **leer** sind. Wenn wir weiter annehmen, daß jede Anwender-Task in einer **Endlosschleife** läuft und freiwillig niemals terminiert, dann kann jede Task den Prozessor für sich vereinnahmen.
In der obigen Task-Queue-Konfiguration ist der Task-Control-Block task_cb[0] der Task_0 als **erstes** eingekettet. Der Scheduler wählt sie aus und der Dispatcher «verbindet» sie mit dem Prozessor. Solange sie im Besitz des Prozessors ist, bleibt ihr Task-Control-Block **eingekettet**. So haben die restlichen Control-Blöcke keine Chance, an die Spitze der Queue zu gelangen. Die zugehörigen Tasks sind **ausgesperrt,** denn ihr Werben um den Prozessor kann vom Scheduler nicht festgestellt werden. Oft wird dies auch als individueller «Hungertod» bezeichnet. Was ist also zu tun?

Das Problem ist lösbar, wenn man jeder Task eine bestimmte **Zeitspanne** (engl.: time-slice) zuweist. Jede Task darf nur **innerhalb** dieser Zeitspanne, auch **Quantum** genannt, laufen. Ist sie am Ende der Zeitspanne immer noch aktiv, dann wird ihr die CPU **entzogen** und die nächste Task in der Task-Queue gestartet. Den Algorithmus dieses sogenannten **Round-Robin-Scheduling** verdeutlicht das folgende Bild 76:

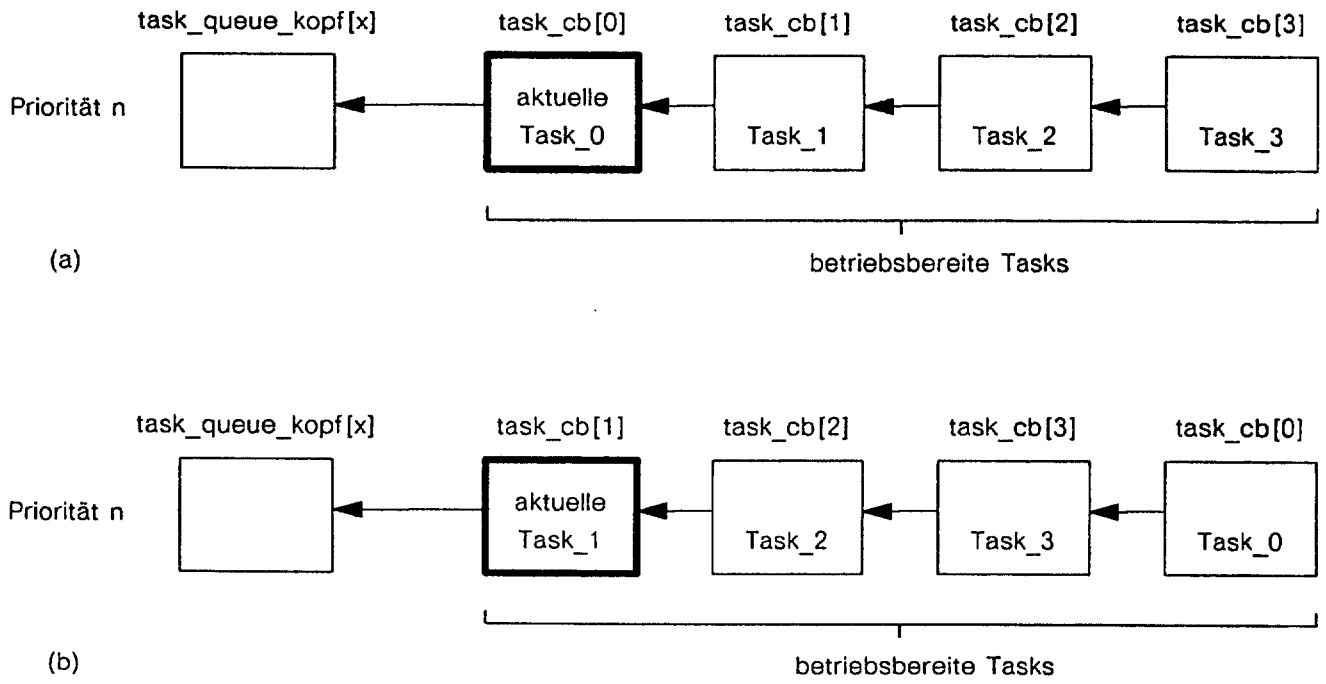


Bild 76: Round-Robin-Scheduling

Im Bild 76(a) liegt der Task-Control-Block der Task_0 an der Spitze der Queue. Sie ist die aktuelle Task, die der Scheduler ausgewählt hat. Wenn ihr Quantum abgelaufen ist, wird ihr Task-Control-Block **ausgekettet** und am Ende der Queue wieder **eingekettet**.
Das Bild 76(b) zeigt, daß danach der Task-Control-Block der Task_1 an der Spitze der Queue liegt. So ist jetzt Task_1 die aktuelle Task und anschließend folgen Task_2, Task_3 und dann wieder Task_0 usw.

In einem Zeitdiagramm läßt sich die Strategie des Round-Robin-Scheduling folgendermaßen darstellen:

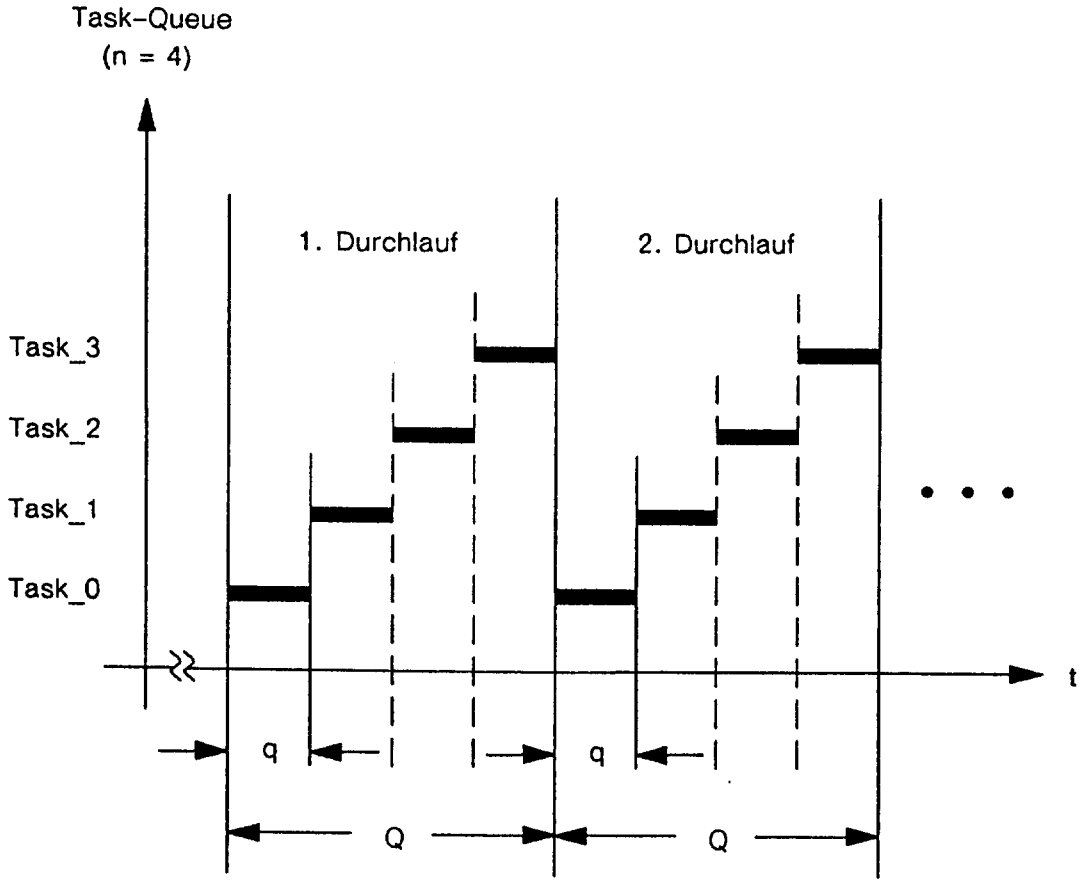


Bild 77: Round-Robin-Scheduling im Zeitdiagramm

Jede Anwender-Task nimmt den Prozessor die Zeitspanne **q** in Anspruch. Es startet als erstes Task_0 und dann der Reihe nach Task_1, Task_2 und Task_3. Sind alle Tasks durchgelaufen, dann haben sie **zusammen** die Zeit **Q** verbraucht. Wenn Q konstant ist, dann kann q als **q = Q/n** berechnet werden, wobei n die Anzahl der Anwender-Tasks darstellt.
Nach jedem Q folgt ein weiterer Durchlauf, und so nehmen die Tasks immer wieder ihre Arbeit auf. Sie laufen scheinbar **parallel** ab, wenn auch mit «gebremster Geschwindigkeit».

1.11.1 Time-Slice-Realisierung

Die einzig interessante Größe der Round-Robin-Strategie ist die Zeitspanne q. Ist sie zu klein, verringert sich die Effizienz des Prozessors, weil der Betriebssystem-Kern zuviele Taskwechsel veranlassen muß.
Ist sie zu groß, werden die Anwender-Tasks zu selten aktiviert, und es verschlechtern sich ihre Reaktions-bzw. Anwortzeiten. Eine Zeitspanne q von etwa **100 ms** ein praxisüblicher Kompromiß. Aus dem Zeitdiagramm im Bild 77 läßt sich erkennen, daß bei q = 100 ms jede Task nur **3·q = 300 ms** warten muß, bis sie wieder an der Reihe ist.
Grundlage der time-slice-Realisierung ist die **Zeitgeber-Hardware**, die im Zeitintervall von 1ms periodische Unterbrechungen fordert (siehe Bild 44). Diese Clock-Ticks unterbrechen die Anwender-Tasks in der Privileg-Ebene 3 und rufen einen **Interrupt-Handler** in der Privileg-Ebene 0 auf. Er ist Mitglied des Betriebssystemkerns und wird benutzt, um die Betriebssystem-Funktion

time_slice aufzurufen. Diese Funktion repräsentiert den time-slice-Manager des Systems. Er zählt die Clock-Ticks und ermittelt daraus für jede Anwender-Task die Zeitspanne q:

- Ist q erreicht, sorgt er für den notwendigen Taskwechsel.
- Ist q **nicht** erreicht, veranlaßt er die Rückkehr zum Interrupt-Handler, und von dort erfolgt die Rückkehr zur unterbrochenen Anwender-Task.

Den Aufruf des time_slice-Managers und die Rückkehr zur unterbrochenen Anwender-Task illustriert Bild 78:

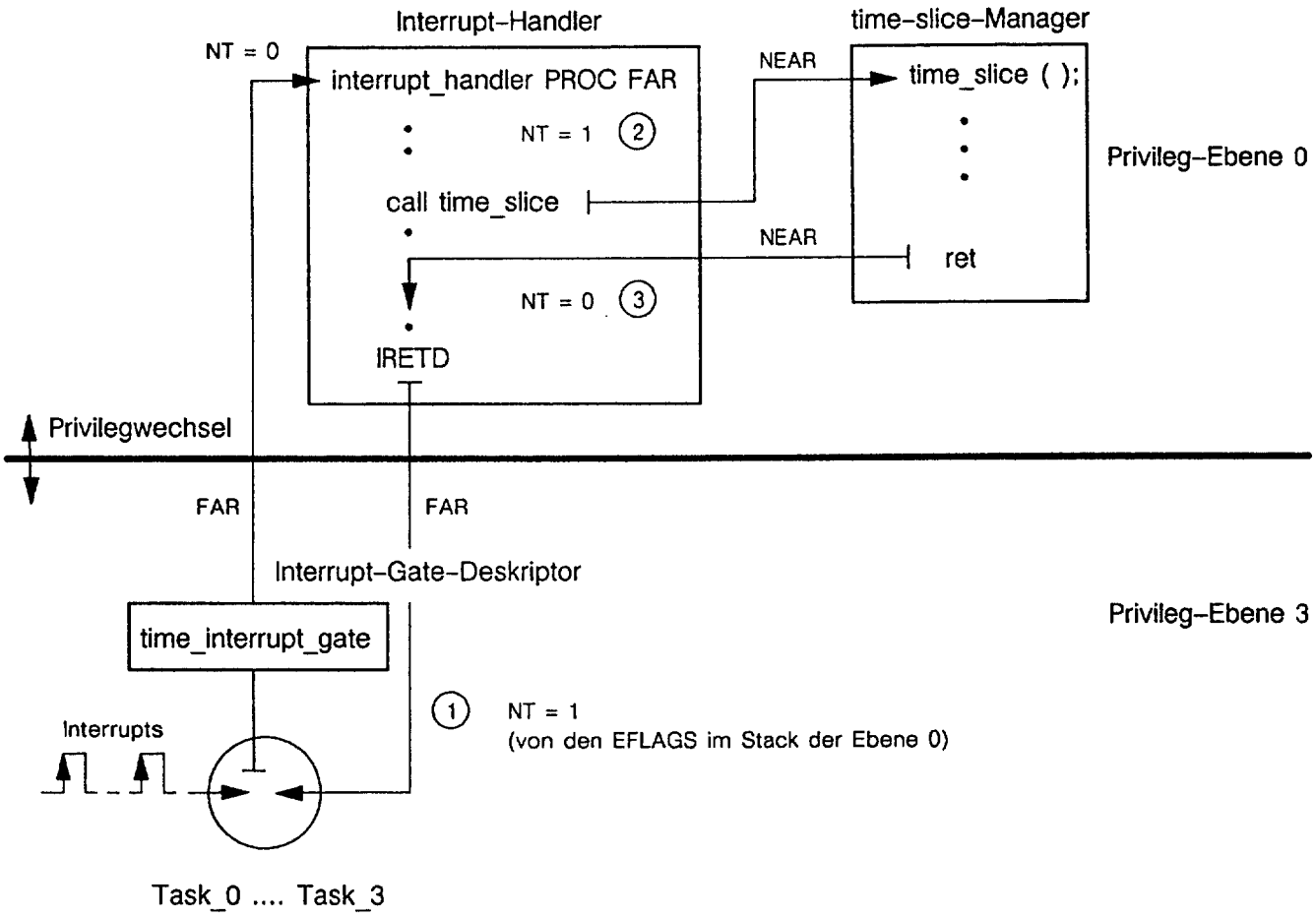


Bild 78: Aufruf des time-slice-Managers und Rückkehr

Die Zeitgeber-Hardware liefert zwar **ständig** Clock-Ticks, sie kann aber damit nur diejenige Anwender-Task unterbrechen, die im Augenblick im Besitz des Prozessors ist. Mit jedem Clock-Tick findet über den Interrupt-Gate-Deskriptor **timer_interrupt_gate** ein Privilegwechsel zum Interrupt-Handler in der Privileg-Ebene 0 statt. Er ist als Assembler-Prozedur realisiert und rettet als erstes alle Register und Selektoren der unterbrochenen Anwender-Task. Danach lädt er den DS-Selektor der OS_Kern_Task und ruft anschließend den time-slice-Manager auf. Die Rückkehr zur unterbrochenen Anwender-Task veranlaßt der Befehl **IRETD**. Bevor dies aber geschieht, schreibt der Interrupt-Handler alle Register und Selektoren der unterbrochenen Task wieder zurück in die CPU.

Der Interrupt-Handler hat aber noch die andere **wichtige** Aufgabe, nach einem Interrupt die Ablaufkontrolle an die unterbrochene Task zurückzugeben.
Wenn ein Hardware-Interrupt oder Interrupt-Befehl einen Interrupt-Gate- oder Trap-Gate-Deskriptor benutzt, dann **löscht** der Prozessor das **NT**-Bit (Nested Task) im EFLAG-Register. Führt der Interrupt-Handler bei **NT = 0** den **IRETD**-Befehl aus, dann erfolgt die Rückkehr zum unterbrochenen CODE32-Segment **innerhalb** der **gleichen** Task. Siehe Bild 79:

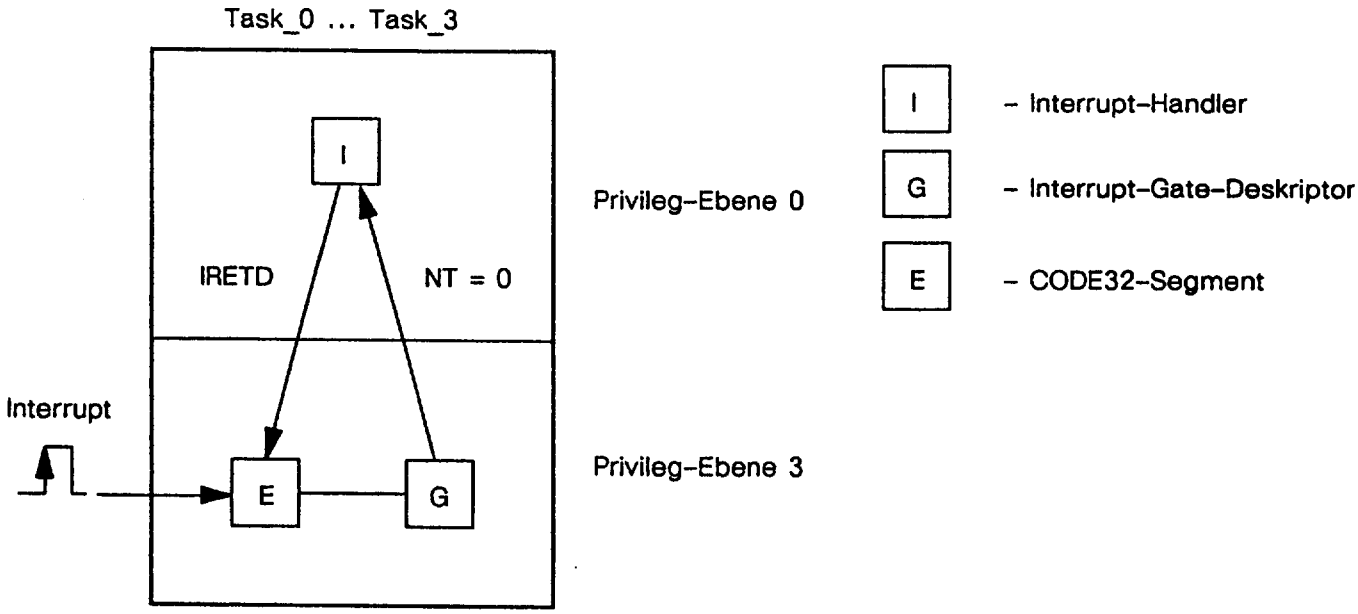


Bild 79: Rückkehr innerhalb der gleichen Task nach einem Interrupt

In einem Multitasking-System liegt nun folgende Situation vor. Veranlaßt der Dispatcher einen Taskwechsel, dann scheidet die OS_Kern_Task aus, und die neue Anwender-Task tritt an ihre Stelle. Dabei bleibt das NT-Bit der ausscheidenen OS_Kern_Task **erhalten** (NT = NT) und das NT-Bit der eintretenden Anwender-Task wird **gesetzt** (NT = 1). Wegen NT = 1 ist das BACK LINK-Feld im Task-Status-Segment der augenblicklich aktiven Anwender-Task als **gültig** markiert. Siehe Bild 80.

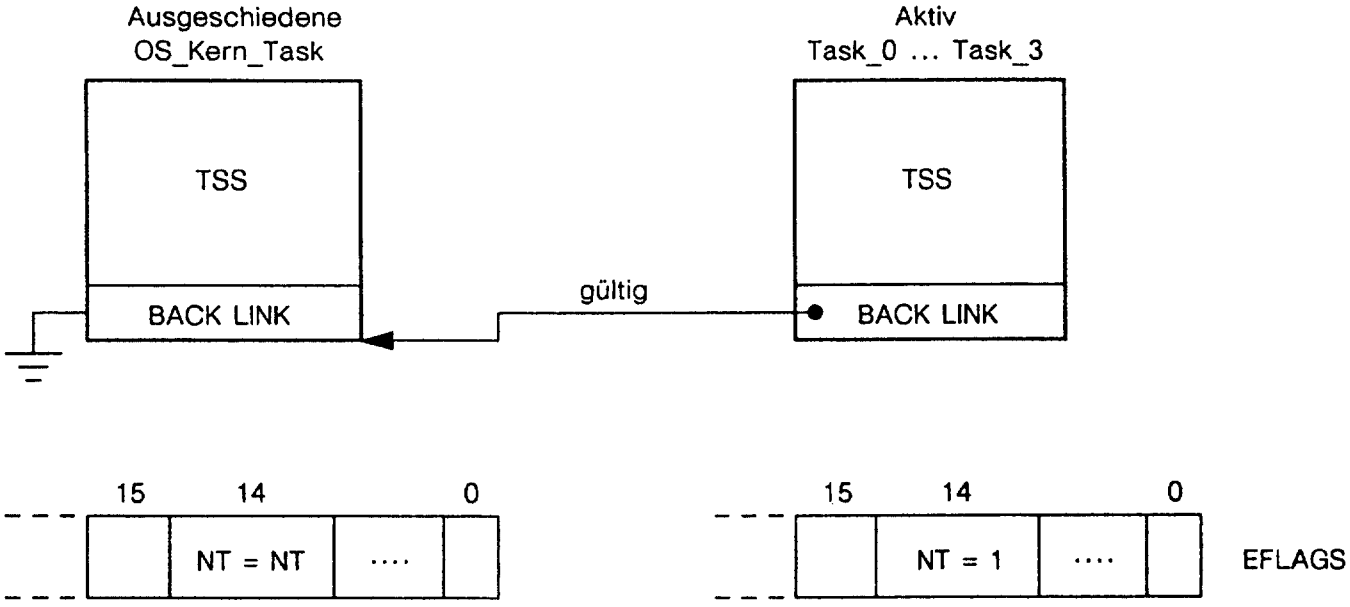


Bild 80: Verhalten von NT beim Taskwechsel

Wird bei dieser Situation der Interrupt-Handler aufgerufen, dann läßt sich das Problem erahnen. Der Prozessor **löscht** das NT-Bit der aktiven Anwender-Task (NT = 0) und erklärt damit das BACK LINK-Feld im zugehörigen Task-Status-Segment als **ungültig**. Die Verbindung zur OS_Kern_Task ist aufgehoben.
Solange das Ende der Zeitspanne q **nicht** erreicht ist und deswegen der time-slice-Manager auch **keinen** Taskwechsel veranlaßt, ist dies zunächst folgenlos. Bevor nämlich das NT-Bit gelöscht wird, speichert der Prozessor die EFLAGs mit NT = 1 im Stack der Ebene 0. Von dort schreibt IRETD die EFLAGs (mit NT = 1) wieder zurück in die CPU und stellt die Verbindung zur OS_Kern_Task wieder her. Es sieht so aus, als sei nichts geschehen. Siehe (1) in Bild 78. Die Situation wird erst dann fatal, wenn das Ende der Zeitspanne q **erreicht** ist. Denn jetzt muß ein Taskwechsel zurück zur OS_Kern_Task erfolgen, der aber wegen NT = 0 (BACK LINK-Feld ungültig) **nicht** realisiert werden kann. So kommt es zu einer Ausnahme-Situation, die ohne besondere Maßnahmen zum «Absturz» des Systems führt. Um dies zu verhindern, ist der Interrupt-Handler mit einigen zusätzlichen Leistungsmerkmalen ausgestattet, wie die Kommentare nach seinen Assembler-Code im Bild 81 zeigen:

```
        NAME int_hand


PUBLIC interrupt_handler
EXTRN  time_slice:NEAR

DATA SEGMENT RW PUBLIC
     DD  ?
DATA ENDS

CODE32 SEGMENT ER PUBLIC

interrupt_handler PROC FAR WC(0)
     pushad                    ;Alle Register und alle Selektoren der
     push ds                   ;unterbrochenen Task retten
     push es
     push fs
     push gs

     pushfd                    ;Taskwechsel ermoeglichen
     pop  eax
     or   ax,0100000000000000B ;NT-Bit=1 setzen (wegen Task-Verkettung)
     push eax
     popfd

     mov  ax,data              ;DS-Selektor der OS-Kern-Task laden
     mov  ds,ax
     call time_slice

     pushfd                    ;Rueckkehr zur unterbrochenen
     pop  eax                  ;Anwender-Task ermoeglichen
     and  ax,1011111111111111B ;NT-Bit=0 setzen (alter Zustand)
     push eax
     popfd

     pop  gs                   ;Alle Register und alle Selektoren
     pop  fs                   ;der unterbrochenen Task zurueckladen
     pop  es
     pop  ds
     popad
     iretd                     ;Zurueck zur unterbrochenen Task
interrupt_handler ENDP

CODE32 ENDS
      END
```

Bild 81: Betriebssytem-Prozedur interrupt_handler (Datei int_hand.src)

Vor dem Aufruf der Betriebssystem-Funktion time_slice **setzt** der Interrupt-Handler mit **or ax,0100000000000000B** das NT-Bit (NT=1) und gibt damit dieser Funktion die Möglichkeit, einen **Taskwechsel** zu veranlassen. Siehe (2) in Bild 78. Findet kein Taskwechsel statt, nimmt der

Interrupt-Handler seine Arbeit wieder auf und **löscht** als erstes mit **and ax,1011111111111111B** das NT-Bit (NT=0). Er stellt durch diese Maßnahme den alten Zustand her und ermöglicht so die Rückkehr zur unterbrochenen Anwender-Task. Siehe (3) in Bild 78.

Betrachten wir als nächstes die Betriebssystem-Funktion time_slice:

```
#define task_max      4

extern char aktuell_task_id;

static short int interrupt_zaehler_task0=0,interrupt_zaehler_task1=0;
static short int interrupt_zaehler_task2=0,interrupt_zaehler_task3=0;
static short int zeitspanne=400/task_max;

void time_slice()
{
 switch (aktuell_task_id)
 {
  case 0: interrupt_zaehler_task0 ++;
          if (interrupt_zaehler_task0 == zeitspanne)
             {
              interrupt_zaehler_task0 = 0;
              round_robin();
              task_wechsel();
             }
  break;

  case 1: interrupt_zaehler_task1 ++;
          if (interrupt_zaehler_task1 == zeitspanne)
             {
              interrupt_zaehler_task1 = 0;
              round_robin();
              task_wechsel();
             }
  break;
```

```
  case 2: interrupt_zaehler_task2 ++;
          if (interrupt_zaehler_task2 == zeitspanne)
             {
              interrupt_zaehler_task2 = 0;
              round_robin();
              task_wechsel();
             }
  break;

  case 3: interrupt_zaehler_task3 ++;
          if (interrupt_zaehler_task3 == zeitspanne)
             {
              interrupt_zaehler_task3 = 0;
              round_robin();
              task_wechsel();
             }
  break;

  default: break;
 }
}
```

Bild 82: Betriebssystem-Funktion time_slice (Datei timslice.c)

In einem Multitasking-System ist es allen Tasks gestattet, einen **gemeinsamen** Speicherbereich oder eine gemeinsame Datei zu lesen und zu schreiben. Dabei ist es notwendig, diese sogenannten **Betriebsmittel** vor dem **gleichzeitigen** Zugriff durch zwei oder mehrere Task zu schützen.
Wenn sich mehrere Tasks um das gleiche Betriebsmittel bewerben, dann ist zu jedem Zeitpunkt immer nur **einer** Task der Zugriff auf dieses Betriebsmittel erlaubt. Die anderen Tasks müssen warten und sind deshalb suspendiert.
So kommt es oft vor, daß eine Task während ihrer Zeitspanne q den Zustand «ready» verläßt und in den Zustand «warten_auf_ein_betriebsmittel» übergeht. Erst wenn das Betriebsmittel frei geworden ist, wird die betreffende Task wieder «geweckt». Sie geht dann zurück in den Zustand «ready», setzt ihre Arbeit fort und beansprucht weiter ihre Zeitspanne q. Wenn Tasks fortlaufend suspendiert und «aufgeweckt» werden, sind sie während ihrer Zeitspanne immer nur kurzzeitig im Besitz der CPU. So läßt sich q als Summe

- $q = edz_1 + edz_2 + \ldots edz_n$

darstellen. Dabei bedeuten **edz** die Dienstzeiten, während derer eine Task die CPU tatsächlich besaß.
Ist q schließlich abgelaufen, dann erfolgt ein **Round-Robin**-Abbruch.
Im folgenden Bild 83 ist der beschriebene Sachverhalt dargestellt:

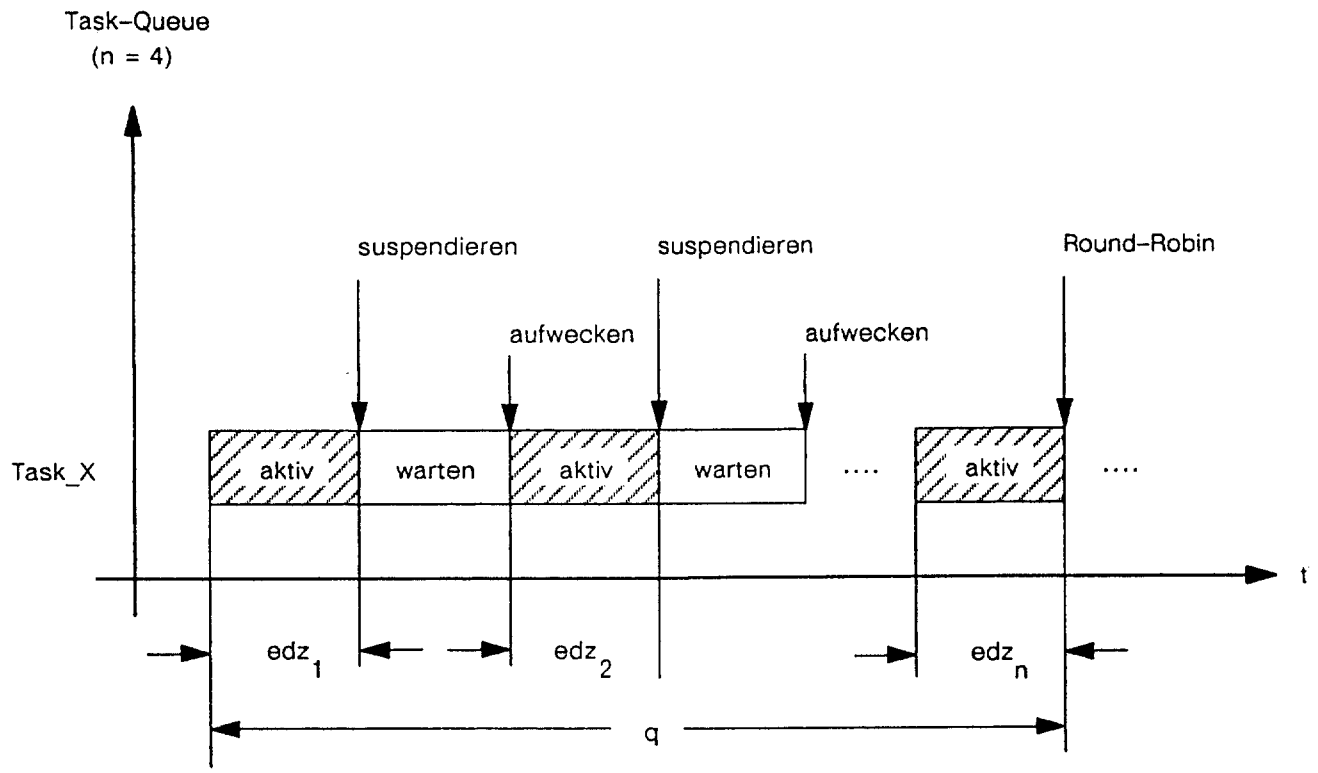


Bild 83: Während der Zeitspanne q sind Tasks immer nur kurzzeitig in edz aktiv

Die Tatsache, daß Anwender-Tasks in der Zeitspanne q nur zeitweilig aktiv sind, wird in der Betriebssystem-Funktion **time_slice** beachtet. Weil eine Anwender-Task nur dann unterbrochen werden kann, wenn sie im Besitz der CPU ist, prüft time_slice die globale Ebene-0-Variable **aktuell_task_id** und stellt fest, um welche Task es sich im Augenblick handelt. So ist jeder Anwender-Task ein individueller **Interruptzähler** zugeordnet, nämlich z.B **interrupt_zaehler_task0**... **interrupt_zaehler_task3**. Abhängig von aktuell_task_id inkrementiert die Betriebssystem-Funktion mit jedem Clock-Tick, also im **Zeittakt** von **1ms**, einen dieser Interruptzähler.
Besteht das System aus n = 4 Anwender-Tasks (task_max), und ist Q = 400 ms konstant, dann ist die Zeitspanne der Anwender-Task abgelaufen, wenn ihr Interruptzähler beispielsweise den Wert q = Q/n bzw. **zeitspanne = 400/task_max = 100 (ms)** enthält. Der Interruptzähler wird daraufhin für einen neuen Zählvorgang gelöscht, und anschließend die Betriebssystem-Funktion **round_robin** aufgerufen. Sie kettet den zuletzt aktuellen Task-Control-Block am Anfang der Task-Queue aus und hängt ihn am Ende der Queue wieder ein. Nachdem dies geschehen ist, ruft time_slice die Betriebssystem-Prozedur **task_wechsel** auf und veranlaßt damit den Taskwechsel zurück zur OS_Kern_Task.
Dort wählt dann der Scheduler, wie üblich, eine **neue** Task aus. Bei dem beschriebenen Verfahren wird eine Anwender-Task, ohne ihr Einverständnis, einfach aus dem System «verdrängt». Daher wird diese Verdrängungsmethode oft auch als **preemptiv**-Scheduling bezeichnet.
Den Round-Robin-Algorithmus realisiert die folgende Betriebssystem-Funktion:

```
#define prioritaet_max  4
#define task_max        4
#include<struct.h>

extern task_queue_kopf_struct task_queue_kopf[prioritaet_max];
extern task_cb_struct task_cb[task_max];
extern char aktuell_task_id;

void round_robin(void)
{
 task_queue_kopf_struct *task_queue_kopf_adr;
 task_cb_struct *neu_task_cb_adr;

 ausketten(&task_queue_kopf[task_cb[aktuell_task_id].task_prioritaet]);

 task_queue_kopf_adr =
            &task_queue_kopf[task_cb[aktuell_task_id].task_prioritaet];
 neu_task_cb_adr = &task_cb[aktuell_task_id];
 einketten (task_queue_kopf_adr,neu_task_cb_adr);
}
```

Bild 84: Betriebssystem-Funktion round_robin (Datei round_ro.c)

Sie ruft als erstes die Betriebssystem-Funktion **ausketten** auf und übergibt ihr als Parameter die Adresse des richtigen Task-Queue-Kopfs, nämlich

- `&task_queue_kopf[task_cb[aktuell_task_id].task_prioritaet]`

Beispiel: Wenn beispielsweise aktuell_task_id = 0 ist und task_prioritaet = 3, dann ergibt sich aus task_cb[0].task_prioritaet = 3 die Adresse des Task-Queue-Kopfs an der Prioritätsstufe 3, also &task_queue_kopf[3].

Weil die Betriebssystem-Funktion **ausketten** nach dem Prinzip FIFO arbeitet, entfernt sie den ersten Task-Control-Block aus der betreffenden Task-Queue (siehe Punkt 1.4.1.2 Ausketten nach FIFO). Dieser Control-Block muß anschließend am Ende der Queue wieder eingekettet werden. So ruft round_robin die Betriebssystem-Funktion **einketten** auf und übergibt ihr als Parameter zwei Adressen, nämlich die Adresse des betreffenden Task-Queue-Kopfs **task_queue_kopf_adr** und die Adresse des einzukettenden Task-Control-Blocks **neu_task_cb_adr** (siehe 1.4.1.1 Einketten nach FIFO).

1.11.1.1 Vermeiden von «Geisterinterrupts»

Manchmal kann ein Clock-Tick, der beim Interrupt-Controller 8259 eine Unterbrechung der aktiven Anwender-Task anfordert, einen sogenannten «Geisterinterrupt» auslösen. Verursacht wird ein solcher Interrupt bei folgender Situation:

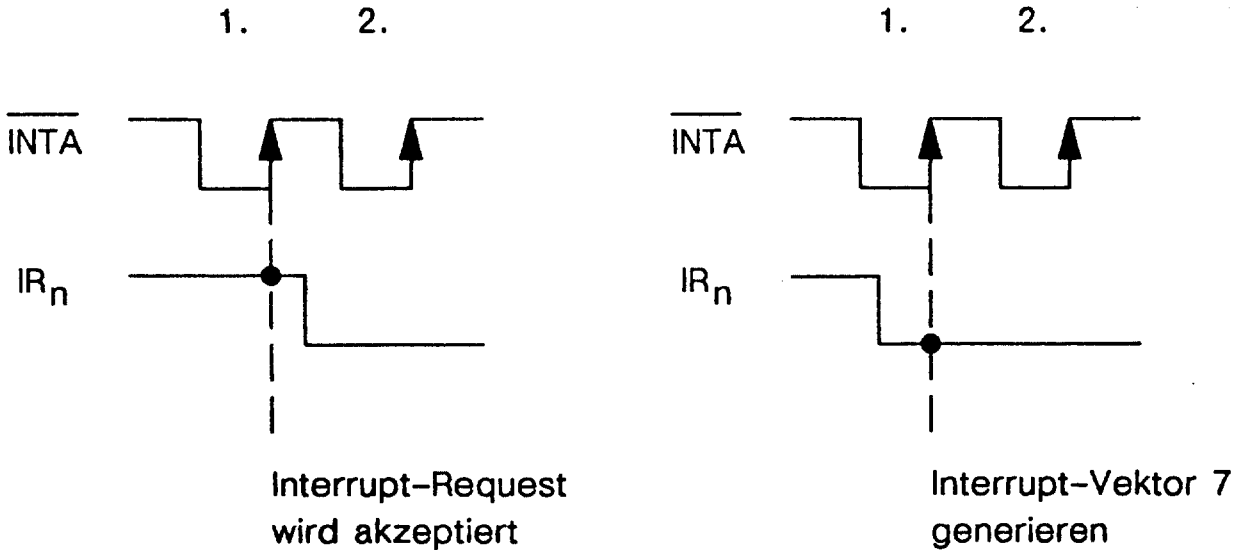


Bild 85a: Bedingung erfüllt

Bild 85b: Ausnahme-Situation

Der Interrupt-Controller quittiert jede Interruptanforderung an seinen Interrupt-Eingängen $\mathbf{IR_n}$ mit **zwei** Interrupt-Acknowledge-Impulsen (INTA*, * bedeutet Low-aktiv). Dabei muß die Anforderung mindestens bis zur **positiven** Flanke des 1. INTA*-Impulses stabil sein. Wenn diese Bedingung erfüllt ist, dann wird der betreffende Interrupt-Request akzeptiert. Siehe Bild 85a. Wenn aber die Anforderung **vor** dem 1. INTA*-Impuls inaktiv wird, dann stellt dies eine Ausnahme-Situation dar und der Interrupt-Controller generiert den **Interrupt-Vektor 7**. Siehe Bild 85b. Es kommt zu einem «Geisterinterrupt», der aber von einem entsprechenden Fehler(Exception)-Handler leicht «aufgefangen» werden kann.

In der Betriebssytem-Funktion **ioinit** ist für eine Interrupt-Anforderung an **IR0** mittels ICW2-Register der Interrupt-Vektor **20H** vereinbart (Siehe Bild 45). Diesen Vektor liefert der Interrupt-Controller immer dann, wenn die Unterbrechung der Anwender-Task durch einen Clock-Tick erfolgt.

Wird dagegen die Anwender-Task aufgrund der oben beschriebenen Ausnahme-Situation unterbrochen, dann liefert der Controller den Interrupt-Vektor 20H+7 = **27H**.

So enthält die **I**nterrupt-**D**eskriptor-**T**abelle (IDT) neben dem Interrupt-Gate-Deskriptor **timer_interrupt_gate** (Offset 20H), der zum Timer-Interrupt-Handler zeigt, auch den Interrupt-Gate-Deskriptor **fehler_interrupt_gate** (Offset 27H), der zum Fehler-Handler zeigt. Siehe Bild 86:

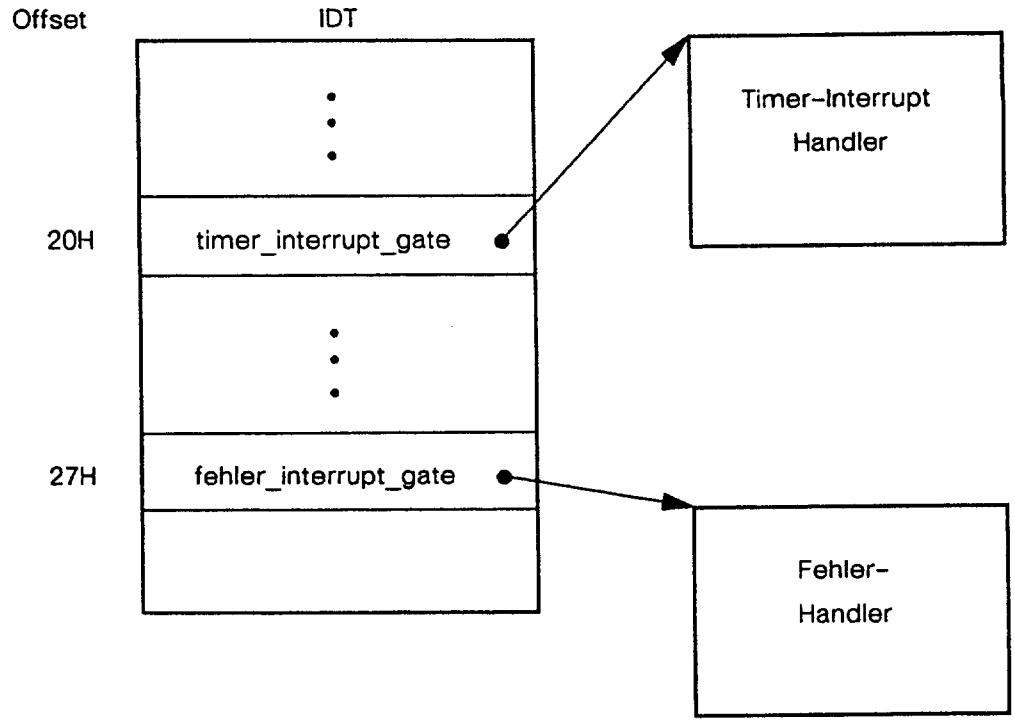


Bild 86: Inhalt der Interrupt-Deskriptor-Tabelle

Der Fehler-Handler gehört zur OS_Kern_Task und enthält lediglich den Befehl **IRETD**. Bei welchem Befehl auch immer die Anwender-Task aufgrund der Ausnahme-Situation unterbrochen worden ist, IRETD sorgt für die Rückkehr zum nächsten Befehl, und die Anwender-Task kann ihre Arbeit fortsetzen. So ist der Fehler-Handler als eine kleine Assembler-Prozedur realisiert, die im folgenden Bild 87 zu sehen ist:

```
          NAME fehler_handler

PUBLIC fehler_handler

CODE32 SEGMENT ER PUBLIC

fehler_handler PROC FAR WC(0)

       iretd                    ;Zurueck zur unterbrochenen
                                ;Anwender-Task

fehler_handler ENDP

CODE32 ENDS
       END
```

Bild 87: Betriebssystem-Prozedur fehler_handler (Datei fehl_han.src)

1.12 Mehrstufiges Herabsetzen der Priorität (engl.: multilevel feedback, Mlf)

Auf den ersten Blick scheint Round-Robin ein fairer Algorithmus zu sein. Denn er verteilt die CPU-Zeit Q **gleichmäßig** auf die Anwender-Tasks und bewahrt sie so vor der **Aussperrung**. Der Nachteil ist aber, daß Round-Robin nur innerhalb einer Gruppe von Tasks **gleicher** Priorität funktioniert. Wenn nämlich ein Multitasking-System aus mehreren solchen Gruppen bzw. **Prioritätsklassen** besteht, dann ist der Round-Robin-Algorithmus alleine nicht mehr geeignet, die CPU-Zeit gerecht auf die Tasks zu verteilen. Betrachten wir hierzu das folgende Bild 88:

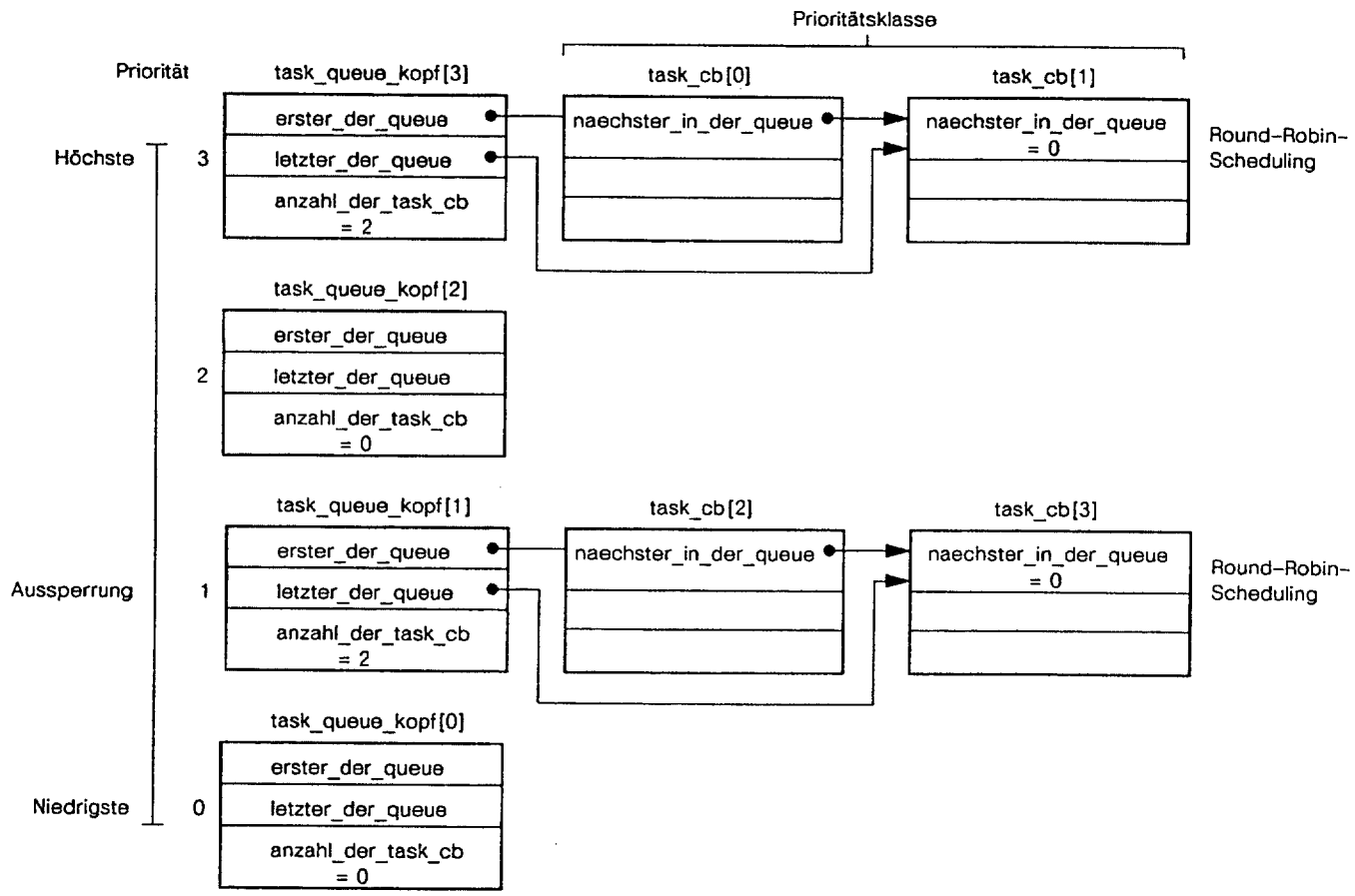


Bild 88: Multitasking-System mit mehreren Prioritätsklassen

Hier hat das Multitasking vier Prioritätsstufen und enthält **zwei** Prioritätsklassen. Die eine Klasse von Anwender-Tasks hat die höchste Priorität 3 und die andere die Priorität 1.

Innerhalb einer jeden Prioritätsklasse kann das Round-Robin-Scheduling benutzt werden und zwischen den Prioritätsklassen das Prioritäts-Scheduling.
Was ist aber, wenn die Anwender-Tasks an der Prioritätsstufe 3 auf ewig dort bleiben? Round-Robin kettet ihre Task-Control-Blöcke reihum immer wieder aus und ein und der Scheduler kann die Task-Control-Blöcke an der Prioritätsstufe 1 niemals feststellen. So vereinnahmen die Tasks an der Prioritätsstufe 3 den Prozessor für sich, während die Tasks an der Prioritätsstufe 1 ausgesperrt bleiben und «verhungern».
Ein Ansatz zur Lösung dieses Problems ist folgender: Jede Anwender-Task, die **neu** im System ankommt, erhält die Prioritätsstufe **n-1**. Wobei **n** die **höchste** Prioritätsstufe ist, die ausschließlich

für sogenannte **Interrupt-Tasks** reserviert bleibt. So beginnt jede «normale» Anwender-Task eine Prioritätsstufe niedriger und verbringt dort die maximale Verweilzeit $\mathbf{T_n}$. Hat sie T_n ausgeschöpft, wird ihre Priorität **dekrementiert** und sie verbringt die Zeit $\mathbf{2T_n}$ auf der Prioritätsstufe **n-2**, anschließend $\mathbf{4T_n}$ auf der Prioritätsstufe **n-3** usw. So läßt sich für jede Prioritätsstufe P die Zeit T_P folgendermaßen berechnen:

$$T_P = 2^{(n-1)-P} T_n$$

Sobald die Task die niedrigste Prioritätsstufe 0 erreicht hat, bleibt sie dort für $\mathbf{2^{n-1}T_n}$ Zeiteinheiten. Ist auch diese Zeit verbraucht, dann schaltet sich Round-Robin ein und gestattet der Task unendlich lange auf der niedrigsten Prioritätsstufe zu bleiben.

Das beschriebene Verfahren führt zum Aufbau von sogenannten **Multilevel-feedback-Task-Queues**, deren Struktur im folgenden Bild 89 illustriert ist:

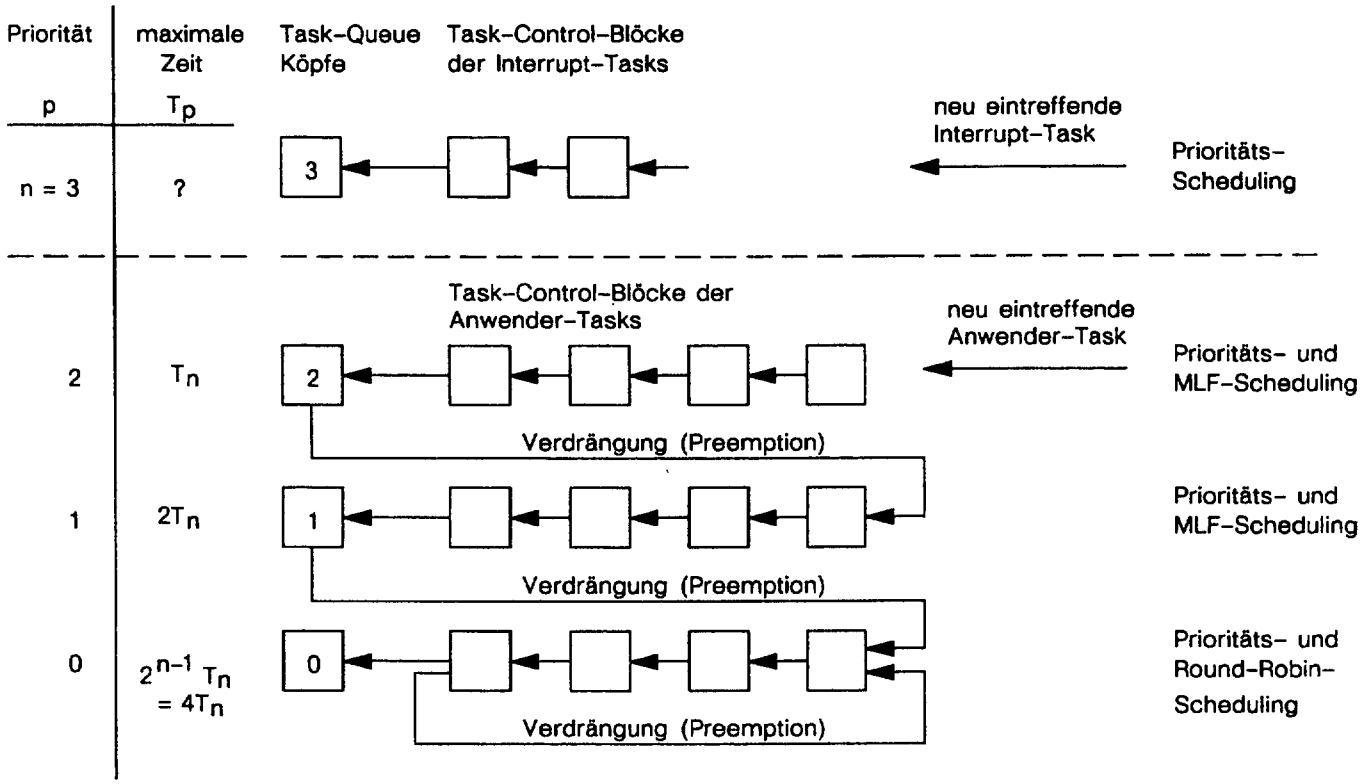


Bild 89: Struktur von Multilevel-feedback-Task-Queues

Eine suspendierte Task, die auf einen Interrupt wartet, wird beim Eintreffen ihres Interrupts in die Task-Queue der höchsten Prioritätsstufe n = 3 eingereiht. Weil sie von äußerster Wichtigkeit ist, darf sie ohne ihr Einverständnis niemals aus dem System verdrängt werden. Sie unterliegt daher ausschließlich dem Prioritäts-Scheduling.

Alle anderen Tasks werden in die Task-Queue der nächst-niedrigeren Prioritätsstufe n-1 eingereiht. Ab dieser Stufe unterliegen sie dem MLF-Scheduling, und zwar solange, bis sie die niedrigste Prioritätsstufe n-3 = 0 erreicht haben. Dort löst dann Round-Robin das MLF-Scheduling ab. So bewahrt die **Kombination** mehrerer Scheduling-Methoden (Priorität, Round-Robin, Mlf) die Tasks vor Aussperrungen.

Der Mechanismus arbeitet außerdem sachgerecht:

- Er bevorzugt kurze Interrupt-Tasks und unterstützt so das Echtzeitverhalten des Systems.

- Er weist «normalen» Tasks auf höheren Prioritätsstufen kürzere Verweilzeiten zu, und auf niedrigeren Prioritätsstufen zum Ausgleich dagegen längere.

Auf welche Weise nun das **Mlf**- und das **Round-Robin**-Scheduling zusammenarbeiten, zeigen die folgenden Betriebssystem-Funktionen. Betrachten wir als erstes die modifizierte Betriebssystem-Funktion **time_slice**. Siehe Bild 90:

```
#define task_max      4
#include<struct.h>

extern task_cb_struct task_cb[task_max];
extern char aktuell_task_id;

static short int zeitspanne=100/task_max;

void time_slice()
{
 switch(task_cb[aktuell_task_id].task_prioritaet)
 {
  case 2:  switch (aktuell_task_id)  /* Prioritaet=2 */
           {
           case 0: task0_mlf(zeitspanne);
           break;

           case 1: task1_mlf(zeitspanne);
           break;

           case 2: task2_mlf(zeitspanne);
           break;

           case 3: task3_mlf(zeitspanne);
           break;

           default: break;
          }
  break;

  case 1:  switch (aktuell_task_id)  /* Prioritaet=1 */
          {
           case 0: task0_mlf(2*zeitspanne);
           break;

           case 1: task1_mlf(2*zeitspanne);
           break;

           case 2: task2_mlf(2*zeitspanne);
           break;

           case 3: task3_mlf(2*zeitspanne);
           break;

           default: break;
          }
  break;
```

```
case 0: switch (aktuell_task_id)  /* Prioritaet=0 */
        {
         case 0: task0_round_robin(4*zeitspanne);
         break;

         case 1: task1_round_robin(4*zeitspanne);
         break;

         case 2: task2_round_robin(4*zeitspanne);
         break;

         case 3: task3_round_robin(4*zeitspanne);
         break;

         default: break;
        }

 break;

 default: break;
 }
}
```

Bild 90: Modifizierte Betriebssystem-Funktion time_slice (Datei timslice.c)

Sie wird, wie gehabt, im Zeittakt von 1ms vom Interrupt-Handler aufgerufen und stellt als erstes mit

- `switch(task_cb[aktuell_task_id].task_prioritaet)`

die **aktuelle** Prioritätsstufe fest.

- Befindet sich die Anwender-Task an der Prioritätsstufe 2 (case 2:) oder 1 (case 1:), dann aktiviert sie das Mlf-Scheduling.
- Befindet sich die Anwender-Task an der Prioritätsstufe 0 (case 0:), dann aktiviert sie das Round-Robin-Scheduling. Siehe Bild 89.

An jeder Prioritätsstufe können sich maximal vier Anwender-Tasks um den Prozessor bewerben. So ermittelt time_slice als nächstes mit

- `switch (aktuell_task_id)`

diejenige Task, die sich im Augenblick an der aktuellen Prioritätsstufe am Anfang der Task-Queue befindet. Für jede Anwender-Task stehen **zwei** Betriebssystem-Funktionen zur Verfügung, die entweder das Mlf- oder das Round-Robin-Scheduling einleiten.

- Befindet sich die Anwender-Task an den Prioritätsstufen 2 oder 1, dann ruft time_slice eine der Funktionen aus der **Mlf**-Gruppe auf und übergibt ihr als aktuellen Parameter entweder **zeitspanne** = t_n (Prioritätsstufe 2) oder **2*zeitspanne** = $2t_n$ (Prioritätsstufe 1).

- Befindet sich die Anwender-Task an der Prioritätsstufe 0, dann ruft time_slice eine Funktion aus der **Round-Robin**-Gruppe auf und übergibt ihr den aktuellen Parameter **4*zeitspanne** = $4t_n$.

In den folgenden Bildern 91 und 92 sind die beiden Funktionsgruppen zusammengefaßt:

```
static short int interrupt_zaehler_task0 = 0;

task0_mlf(short int t)
{
 interrupt_zaehler_task0 ++;
 if (interrupt_zaehler_task0 == t)
    {
     interrupt_zaehler_task0 = 0;
     mlf();
     task_wechsel();
    }
}
```

(Datei t0_mlf.c)

```
static short int interrupt_zaehler_task1 = 0;

task1_mlf(short int t)
{
 interrupt_zaehler_task1 ++;
 if (interrupt_zaehler_task1 == t)
    {
     interrupt_zaehler_task1 = 0;
     mlf();
     task_wechsel();
    }
}
```

(Datei t1_mlf.c)

```
static short int interrupt_zaehler_task2 = 0;

task2_mlf(short int t)
{
 interrupt_zaehler_task2 ++;
 if (interrupt_zaehler_task2 == t)
    {
     interrupt_zaehler_task2 = 0;
     mlf();
     task_wechsel();
    }
}
```

(Datei t2_mlf)

```
static short int interrupt_zaehler_task3 = 0;

task3_mlf(short int t)
{
 interrupt_zaehler_task3 ++;
 if (interrupt_zaehler_task3 == t)
    {
     interrupt_zaehler_task3 = 0;
     mlf();
     task_wechsel();
    }
}
```

(Datei t3_mlf)

Bild 91: Gruppe der Mlf-Betriebssystem-Funktionen

```
static short int interrupt_zaehler_task0 = 0;

task0_round_robin(short int t)
{
 interrupt_zaehler_task0 ++;
 if (interrupt_zaehler_task0 == t)
    {
     interrupt_zaehler_task0 = 0;
     round_robin();
     task_wechsel();
    }
}
```

(Datei t0_roro.c)

```
static short int interrupt_zaehler_task1 = 0;

task1_round_robin(short int t)
{
 interrupt_zaehler_task1 ++;
 if (interrupt_zaehler_task1 == t)
    {
     interrupt_zaehler_task1 = 0;
     round_robin();
     task_wechsel();
    }
}
```

(Datei t1_roro.c)

```
static short int interrupt_zaehler_task2 = 0;

task2_round_robin(short int t)
{
 interrupt_zaehler_task2 ++;
 if (interrupt_zaehler_task2 == t)
    {
     interrupt_zaehler_task2 = 0;
     round_robin();
     task_wechsel();
    }
}
```

(Datei t2_roro.c)

```
static short int interrupt_zaehler_task3 = 0;

task3_round_robin(short int t)
{
 interrupt_zaehler_task3 ++;
 if (interrupt_zaehler_task3 == t)
    {
     interrupt_zaehler_task3 = 0;
     round_robin();
     task_wechsel();
    }
}
```

(Datei t3_roro.c)

Bild 92: Gruppe der Round-Robin-Betriebssystem-Funktionen

In allen Funktionen wird der Interruptzähler der aktuellen Anwender-Task (interrupt_zaehler_task0...interrupt_zaehler_task3) **inkrementiert** und geprüft, ob er die vorgesehenen Zeiteinheiten an der derzeitigen Prioritätsstufe schon erreicht hat (t_n oder $2t_n$ oder $4t_n$). Wenn ja, wird der Interruptzähler gelöscht und an der Prioritätsstufe **1** oder **2** die Betriebssystem-Funktion **mlf** oder an der Prioritätsstufe **0** die Betriebssystem-Funktion **round_robin** aufgerufen. Am Ende veranlassen alle Funktionen einen Taskwechsel zurück zur OS_Kern_Task.

Die Betriebssystem-Funktion mlf unterscheidet sich nur geringfügig von der Betriebssystem-Funktion round_robin:

```
#define prioritaet_max 4
#define task_max       4
#include<struct.h>

extern task_queue_kopf_struct task_queue_kopf[prioritaet_max];
extern task_cb_struct task_cb[task_max];
extern char aktuell_task_id;

void mlf(void)
{
 task_queue_kopf_struct *task_queue_kopf_adr;
 task_cb_struct *neu_task_cb_adr;

 ausketten(&task_queue_kopf[task_cb[aktuell_task_id].task_prioritaet]);

 task_cb[aktuell_task_id].task_prioritaet --;

 task_queue_kopf_adr =
            &task_queue_kopf[task_cb[aktuell_task_id].task_prioritaet];
 neu_task_cb_adr = &task_cb[aktuell_task_id];
 einketten (task_queue_kopf_adr,neu_task_cb_adr);
}
```

Bild 93: Betriebssytem-Funktion mlf (Datei mlf.c)

Genauso wie round_robin, kettet sie als erstes den Task-Control-Block der aktuellen Anwender-Task aus der Task-Queue aus. Sie **dekrementiert** aber anschließend mit

- `task_cb[aktuell_task_id].task_prioritaet --;`

die **Priorität** der betreffenden Task, und kettet sie danach in die Task-Queue der nächst-niedrigeren Prioritätsstufe wieder ein. Wenn keine der Anwender-Tasks zwischendurch suspendiert wird, wandern sie der Reihe nach zur **niedrigsten** Prioritätsstufe **0** und bleiben dort unendlich lange. Wegen **zeitspanne = 100/task_max = 25**(ms) (siehe Bild 90), läuft dann auf dieser Prioritätsstufe nach der Round-Robin-Methode jede Anwender-Task mit einem Quantum von **4*zeitspanne = 100**(ms) ab.

Die Aktionen des Mlf- oder Round-Robin-Scheduling führen zu Task-Zustandsübergängen, die sich formal folgendermaßen ausdrücken lassen:

- **zeitspanne beendet** (Taskname); laufend -> bereit

Somit läßt sich das Zustandsdiagramm aus Bild 49 um diesen neuen Übergang ergänzen:

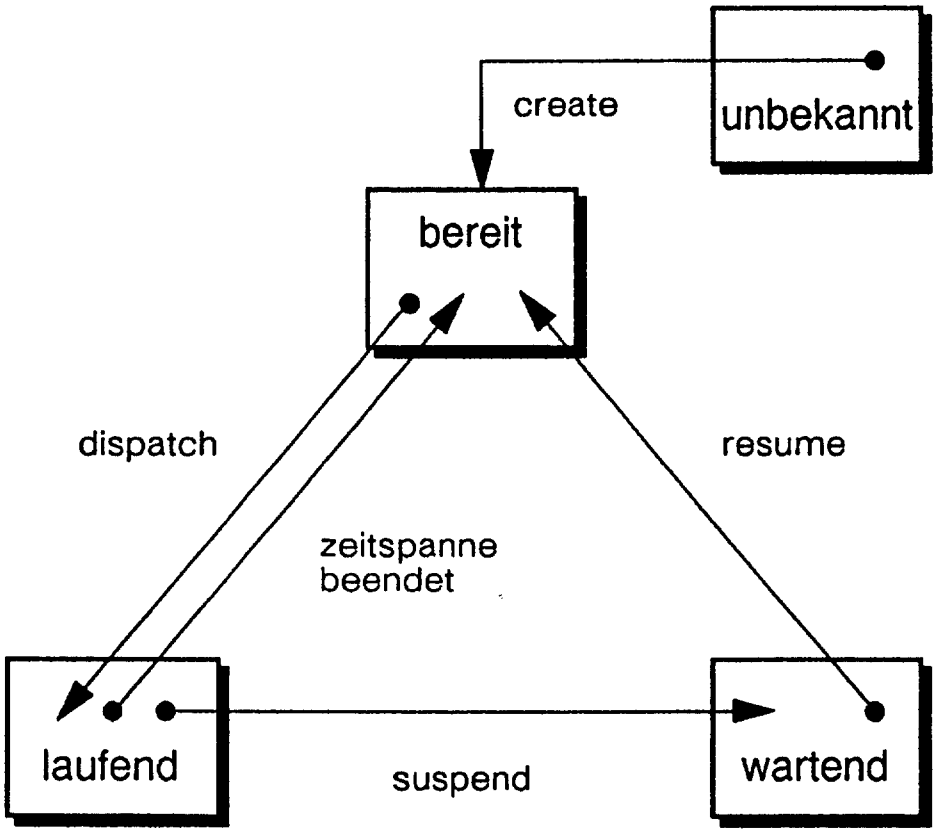


Bild 94: Task-Zustandsdiagramm; zweite Ergänzung

1.13 Eingabe/Ausgabe

Es gehört zu den Hauptaufgaben des Betriebssystems, den Zugriff auf I/O-Ports wie z. B. parallel I/O, seriell I/O, Interrupt-Controller, DMA-Controller usw. zu kontrollieren. Aus diesem Grund sind die Assembler-Befehle **IN**, **INS**, **OUT** und **OUTS**, über die der Prozessor die I/O-Ports erreicht, als sogenannte **I/O-sensitive** Befehle implementiert. Sie können nur ausgeführt werden, wenn die folgende Bedingung erfüllt ist:

- **Task CPL <= Port IOPL**
  (CPL = Current Privilege Level,
  IOPL = I/O-Privilege Level)

Diese Bedingung bedeutet, daß eine Task I/O-Befehle nur dann uneingeschränkt ausführen kann, wenn ihre Privileg-Ebene entweder numerisch **kleiner** ist als die Privileg-Ebene der I/O-Ports oder wenn beide Privileg-Ebenen **gleich** sind.
Die Privileg-Ebene der I/O-Ports wird durch das **IOPL**-Feld (Bits 13,12) in den **EFLAGs** bestimmt (00..11 binär). Siehe Bild 95:

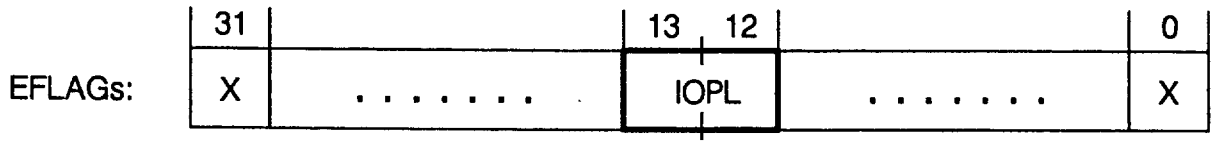


Bild 95: IOPL-Feld im EFLAG-Register

Ist IOPL = 00 binär, dann liegt die Kontrolle über die Ein-/Ausgabe-Operationen einzig und alleine bei der OS_Kern_Task in der Privileg-Ebene 0. Die Anwender-Tasks in der Privileg-Ebene 3 sind **nicht** in der Lage, die I/O-Ports direkt zu erreichen. Sie sind stattdessen gezwungen, entsprechende Betriebssystem-Funktionen aus der Privileg-Ebene 0 zu benutzen.
So gibt es z. B. die Betriebssytem-Funktion **read_io**, die einen **Input**-Port liest, und die Betriebssystem-Funktion **write_io**, die in einen **Output**-Port schreibt.

Nehmen wir an, daß in der Privileg-Ebene 0 das folgende System paralleler I/O-Ports existiert:

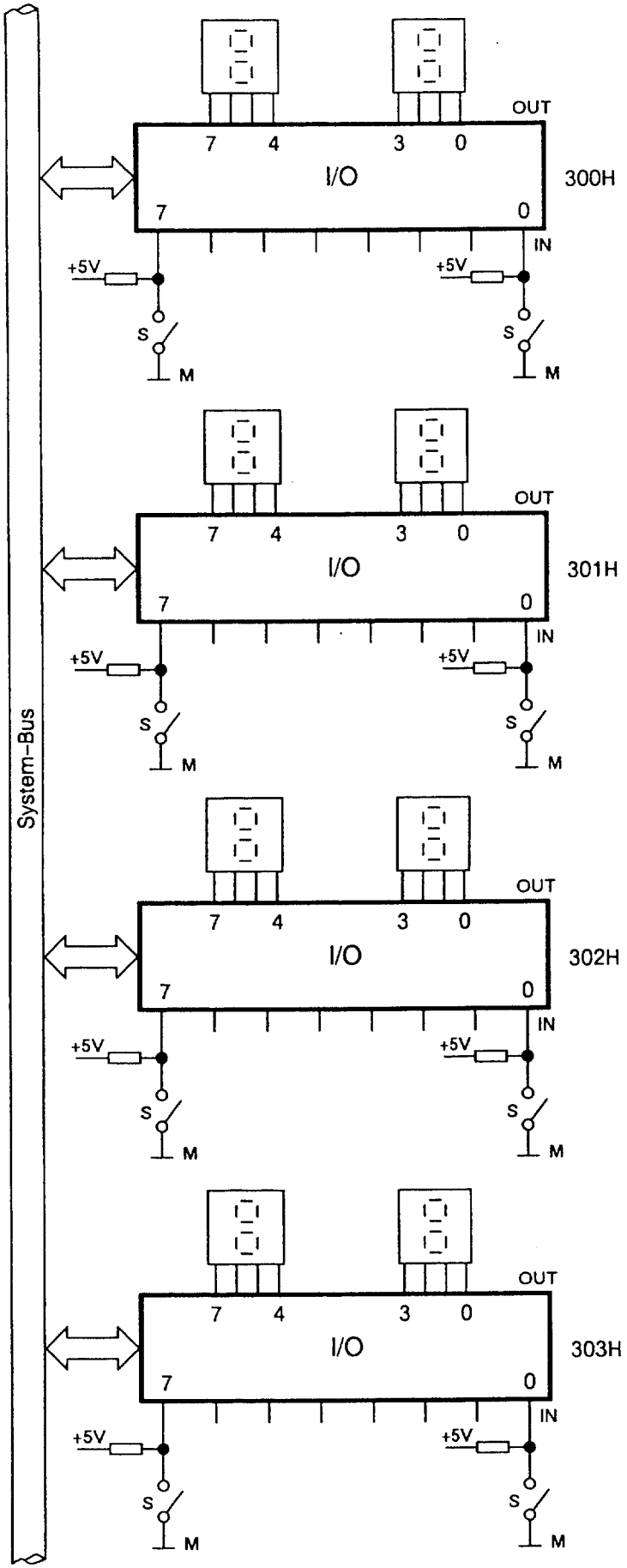


Bild 96: Hardware-Struktur eines parallelen I/O-Systems

Es besteht aus **vier** Ports mit den Adressen 300H..303H. Die Eingabekanäle sind jeweils mit acht **Schaltern** verbunden, und die Ausgabekanäle mit jeweils zwei **7-Segment-Anzeigen**. Die Betriebssystem-Funktion read_io **liest** die Schalterstellungen als 8-Bit-Binärmuster von einem der Eingabekanäle (S offen = «H», S geschlossen = «L») und speichert das Eingabedatum in der **char**-Variablen **puffer** der aktuellen Anwender-Task.
Die Betriebssystem-Funktion write_io liest diesen puffer und **schreibt** das Datum in einen der Ausgabekanäle, wo es an der 7-Segment-Anzeige hexadezimal angezeigt wird.

So verbinden die beiden Betriebssystem-Funktionen die Anwender-Tasks in der Privileg-Ebene 3 mit den I/O-Ports in der Privileg-Ebene 0. Siehe Bild 97:

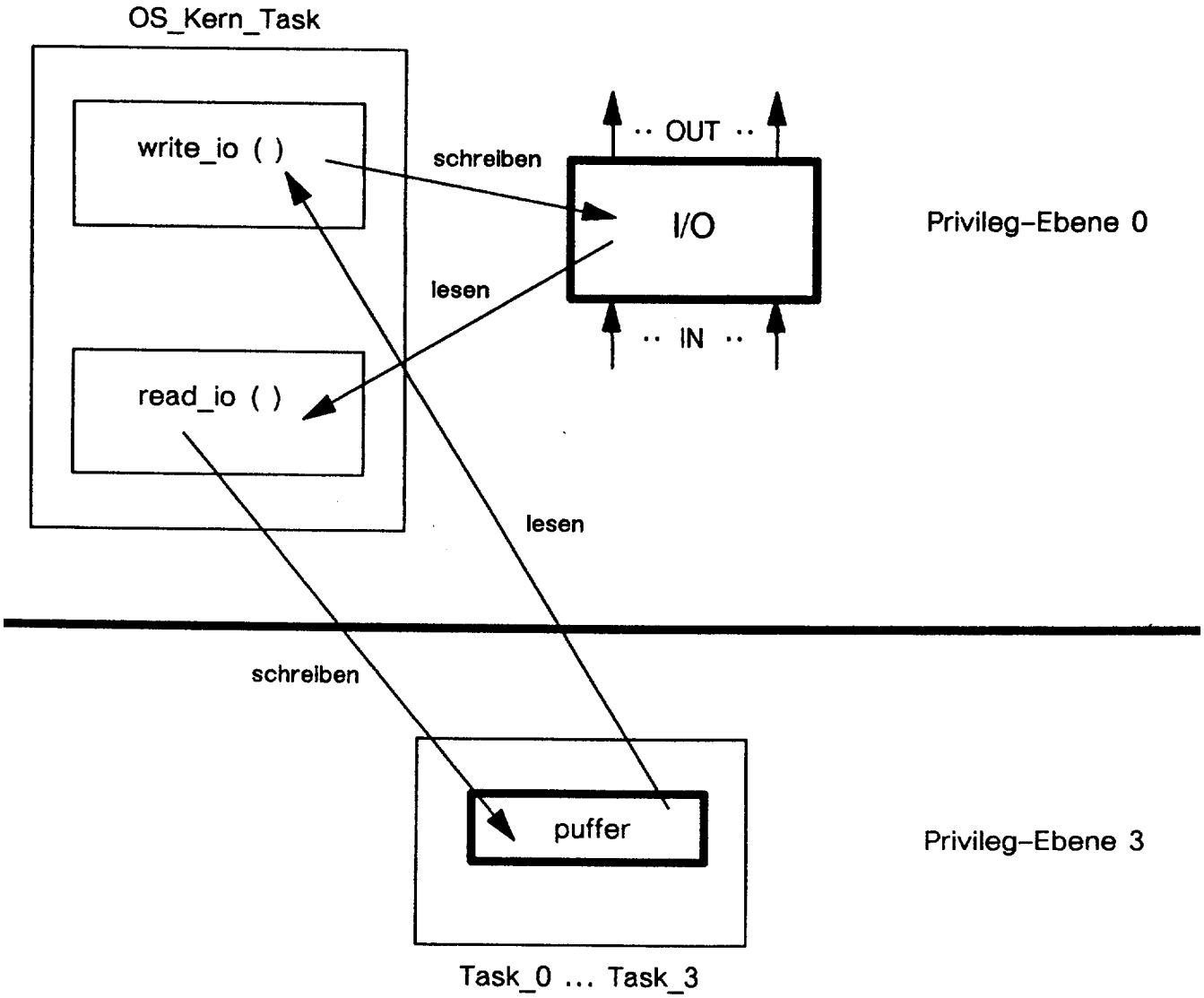


Bild 97: Verbindungen zwischen den Anwender-Tasks und den I/O-Ports

Der Aufruf von read_io bzw. write_io erfolgt über Call-Gate-Deskriptoren mit jeweils **zwei** Parametern, nämlich

- `KN_read_io (short int,char *);` und
- `KN_write_io(short int,char *);`

Dabei wird an

- den **linken** Parameter die 16-Bit-Adresse eines I/O-Ports übergeben (port_0 = 300H...port_3 = 303H) und an
- den **rechten** Parameter die 48-Bit-Adresse des Anwender-Puffers (&puffer = 16-Bit-Selektor, 32-Bit-Offset).

Wie im einzelnen die Anwender-Tasks mit den beiden Betriebssystem-Funktionen korrespondieren, zeigt das Bild 98:

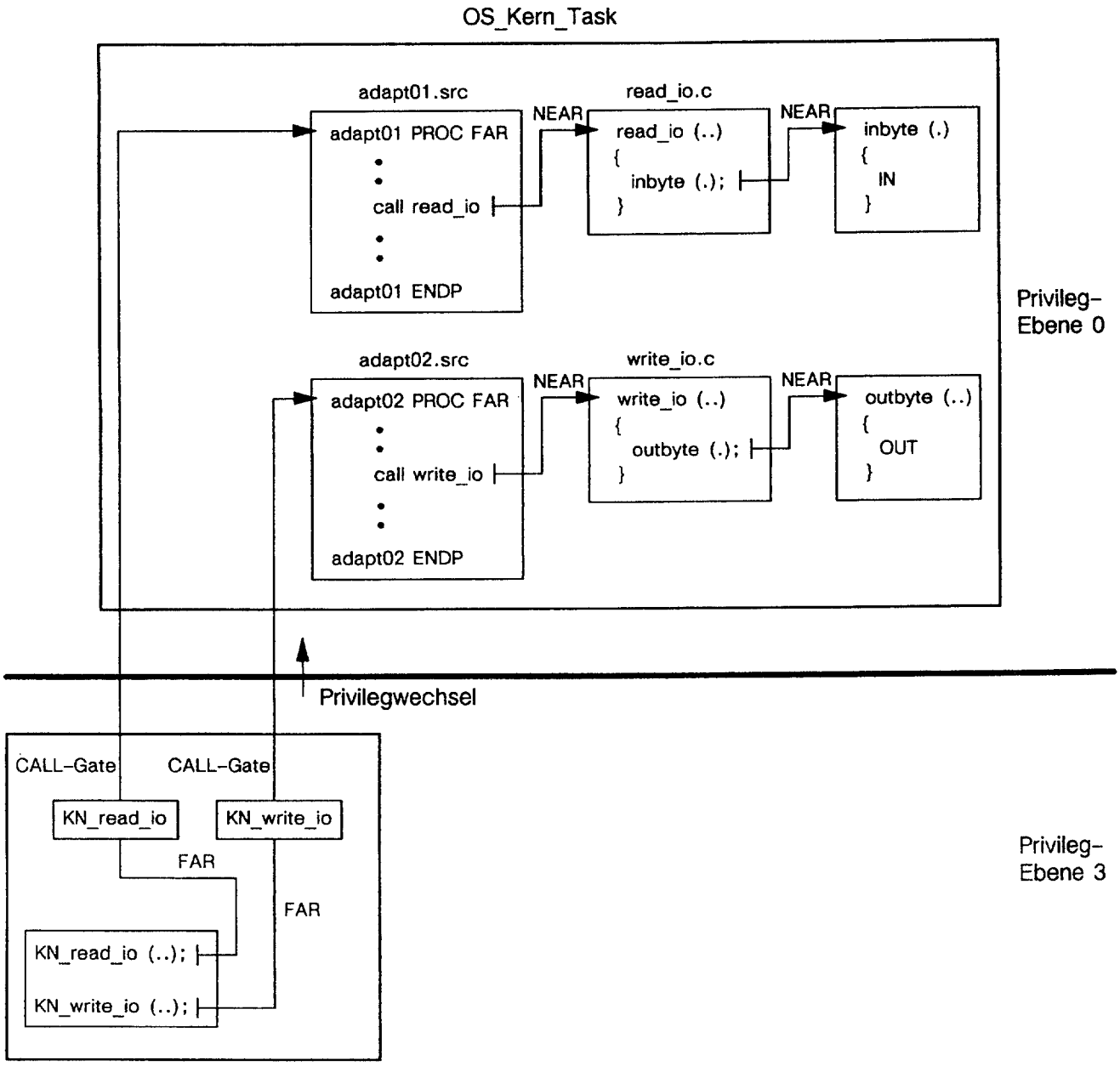


Bild 98: Aufruf der Betriebssystem-Funktionen read_io und write_io

Zwischen den Anwender-Tasks und den Betriebssystem-Funktionen liegen die Assembler-Adapter **adapt01** und **adapt02**. Sie sperren als erstes potentielle Interrupt-Anforderungen mit **CLI** (**C**lear **I**nterrupt-Enable-Flag) und schützen so die OS_Kern_Task vor Unterbrechungen. Danach verschieben sie die übergebenen Parameter **port_0**...**port_3** und **&puffer** an die richtigen Stellen

im Privileg-Ebene-0-Stack, laden den DS-Selektor um, rufen die Betriebssystem-Funktionen read_io oder write_io auf und geben vor der Rückkehr zu den Anwender-Tasks die Interrupt-Anforderungen mit **STI** (**S**et **I**nterrupt-Enable-Flag) wieder frei.
Auf diese Weise läßt das Multitasking-System nur Unterbrechungen der Anwender-Tasks zu.

Die Assembler-Programme der beiden Adapter zeigen die folgenden Bilder 99 und 100:

```
            NAME adapt01

PUBLIC adapt01
EXTRN read_io:NEAR

DATA SEGMENT RW PUBLIC
 DD  ?
DATA ENDS

CODE32  SEGMENT ER PUBLIC

adapt01 PROC FAR WC(3)
    cli                     ;Interrupt sperren
    push ebp                ;Prolog
    mov  ebp,esp
    push ds                 ;DS-Selektor der Anwender-Task retten
    mov  eax,[ebp+20]       ;Parameter: port_nr
    push eax
    mov  eax,[ebp+16]       ;Parameter: Adresse des Anwenderpuffers
    push eax                ;DS-Selektor
    mov  eax,[ebp+12]
    push eax                ;Offset
    mov  ax,data            ;DS-Selektor/Deskriptor der OS-Kern-Task
    mov  ds,ax              ;laden
    call read_io            ;Aufruf der Betriebssystem-Funktion
    pop  ds                 ;DS-Selektor/Deskriptor der Anwender-Task
    pop  ebp                ;Epilog
    sti                     ;Interrupt freigeben
    ret  12                 ;Zurueck zur Anwender-Task und
                            ;Parameter entfernen
adapt01 ENDP

CODE32  ENDS
        END
```

Bild 99: Adapter adapt01 für Betriebssystem-Funktion read_io (Datei adapt01.src)

```
            NAME adapt02

PUBLIC adapt02
EXTRN write_io:NEAR

DATA SEGMENT RW PUBLIC
 DD  ?
DATA ENDS

CODE32  SEGMENT ER PUBLIC

adapt02 PROC FAR WC(3)
    cli                     ;Interrupt sperren
    push ebp                ;Prolog
    mov  ebp,esp
    push ds                 ;DS-Selektor der Anwender-Task retten
    mov  eax,[ebp+20]       ;Parameter: port_nr
    push eax
    mov  eax,[ebp+16]       ;Parameter: Adresse des Anwenderpuffers
    push eax                ;DS-Selektor
    mov  eax,[ebp+12]
    push eax                ;Offet
    mov  ax,data            ;DS-Selektor/Deskriptor der OS-Kern-Task
    mov  ds,ax
    call write_io           ;Aufruf der Betriebssystem-Funktion
    pop  ds                 ;DS-Selektor/Deskriptor der Anwender-Task
    pop  ebp                ;Epilog
    sti                     ;Interrupt freigeben
    ret  12                 ;Zurueck zur Anwender-Task und
                            ;Parameter entfernen
adapt02 ENDP

CODE32  ENDS
        END
```

Bild 100: Adapter adapt02 für Betriebssystem-Funktion write_io (Datei adapt02.src)

Betrachten wir als nächstes die Betriebssystem-Funktion **read_io**:

```
#include<i86.h>

void read_io(short int port_nr,char *puffer)
{
 *puffer = inbyte(port_nr);  /*Eingabe-Byte an Anwender-Task*/
}
```

Bild 101: Betriebssystem-Funktion read_io (Datei read_io.c)

Sie ruft die Funktion **inbyte** auf, die in der Header-Datei i86.h folgendermaßen definiert ist:

- `unsigned char inbyte(unsigned short port)`

Diese Funktion ermöglicht einem C-Programm **Lese**-Zugriffe auf I/O-Ports, und read_io benutzt sie, um die Eingabekanäle von port_0...port_3 zu lesen. So generiert der 32-Bit-C-Compiler ic386/486 für den Funktionsaufruf inbyte(port_nr) den folgenden Assembler-Code:

```
• MOV DX,port_nr
  IN  AL,DX
```

Weil eine Task in der Privileg-Ebene 0 **uneingeschränktes** Zugriffsrecht auf Daten in niedrigeren Privileg-Ebenen hat, kann read_io mit

```
• *puffer = inbyte(port_nr)
```

die Eingabedaten problemlos in den puffer der aktuellen Anwender-Task in der Privileg-Ebene 3 schreiben.

Betrachten wir nun die Betriebssystem-Funktion **write_io**.

```
#include<i86.h>

void write_io(short int port_nr,char *puffer)
{
 outbyte(port_nr,*puffer);      /*Ausgabe-Byte an Anwender-Task*/
}
```

Bild 102: Betriebssystem-Funktion write_io (Datei write_io.c)

Sie ruft die Funktion **outbyte** auf, die ebenfalls in der Header-Datei i86.h definiert ist:

```
• void outbyte(unsigned short port,unsigned char value)
```

Diese Funktion ermöglicht einem C-Programm **Schreib**-Zugriffe auf I/O-Ports, und write_io benutzt sie, um in die Ausgabekanäle von port_0...port_3 zu schreiben. (Siehe hierzu auch Bild 45). Sie bekommt zwei Parameter, nämlich eine Portadresse (port_nr = 300H...303H) und den augenblicklichen Pufferinhalt (*puffer) der aktuellen Anwender-Task, und sie bringt mit OUT DX,AL den Pufferinhalt zur Anzeige.

1.14 Realisierung eines Preemptiv-Multitasking-Systems

Mit Hilfe der bisher vorgestellten Betriebssystem-Funktionen kann nun eine Anzahl von Anwender-Tasks **parallel** ablaufen. So soll daher ein Multitasking-System realisiert werden, das aus der OS_Kern_Task, der Start_Task und vier Anwender-Tasks Task_0, Task_1, Task_2 und Task_3 besteht. Mit Ausnahme der Start_Task laufen alle Tasks in einer Endlos-Schleife, und sie sind alle aus einer Menge von Modulen zusammengesetzt. So ergibt sich die folgende modularisierte Software-Struktur:

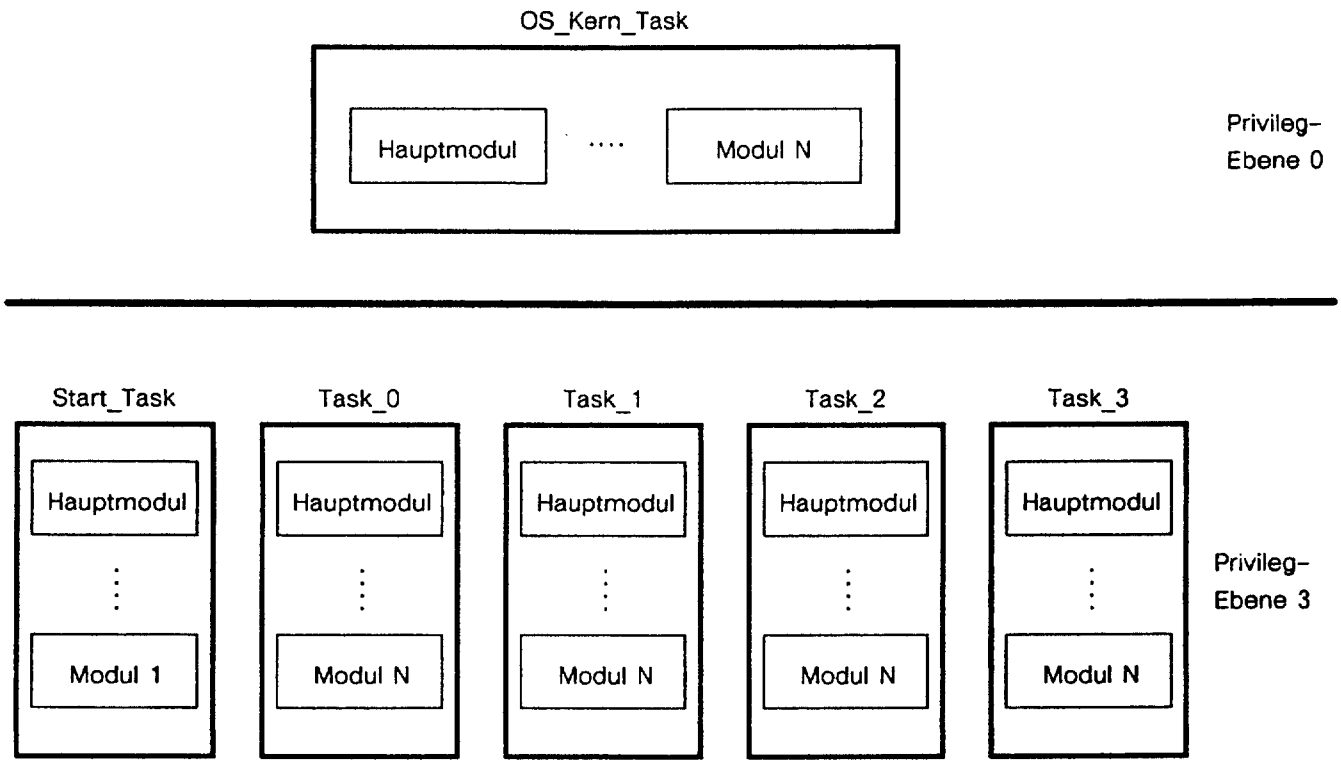


Bild 103: Modularisiertes Multitasking-System

Genauso wie die OS_Kern_Task oder die Start_Task bestehen alle Anwender-Tasks aus einem Assembler-**Hauptmodul** und einer Reihe von Anwender-Modulen (C-Module).

Die Module aller Anwender-Tasks sind in folgenden Bildern zusammengestellt:

1. Module der Task_0:

```
            NAME start_0

PUBLIC start_0_proc
EXTRN  task_0_proc:NEAR

DATA SEGMENT RW PUBLIC
 DD ?
DATA ENDS

STACK STACKSEG 100H

CODE32 SEGMENT ER PUBLIC

start_0_proc PROC  NEAR

  call task_0_proc       ;Aufruf des Anwender-Moduls
                         ;der Task_0
start_0_proc ENDP

CODE32 ENDS
       END start_0_proc,DS:DATA,SS:STACK
```

Bild 104: Hauptmodul start_0 (Datei start_0.src)

```
extern void far KN_write_io(short int,char *);
extern void far KN_read_io(short int,char *);

static char puffer;

task_0_proc()
{
 while(1)
 {
  short int port_0=0x300;

  KN_read_io(port_0,&puffer);
  KN_write_io(port_0,&puffer);
 }
}
```

Bild 105: Anwender-Modul task_0 (Datei task_0.c)

2. Module der Task_1:

```
            NAME start_1

PUBLIC start_1_proc
EXTRN  task_1_proc:NEAR

DATA SEGMENT RW PUBLIC
DD ?
DATA ENDS

STACK STACKSEG 100H

CODE32 SEGMENT ER PUBLIC

start_1_proc PROC  NEAR

      call task_1_proc          ;Aufruf des Anwender-Moduls der
                                ;Task 1
start_1_proc ENDP

CODE32 ENDS
       END start_1_proc,DS:DATA,SS:STACK
```

Bild 106: Hauptmodul start_1 (Datei start_1.src)

```
extern void far KN_write_io(short int,char *);
extern void far KN_read_io(short int,char *);

static char puffer;

task_1_proc()
{
 while(1)
 {
  short int port_1=0x301;

  KN_read_io(port_1,&puffer);
  KN_write_io(port_1,&puffer);
 }
}
```

Bild 107: Anwender-Modul task_1 (Datei task_1.c)

3. Module der Task_2:

```
          NAME start_2

PUBLIC start_2_proc
EXTRN  task_2_proc:NEAR

DATA SEGMENT RW PUBLIC
DD ?
DATA ENDS

STACK STACKSEG 100H

CODE32 SEGMENT ER PUBLIC

start_2_proc PROC  NEAR

      call task_2_proc       ;Aufruf des Anwender-Moduls
                             ;der Task_2
start_2_proc ENDP

CODE32 ENDS
       END start_2_proc,DS:DATA,SS:STACK
```

Bild 108: Hauptmodul start_2 (Datei start_2.src)

```
extern void far KN_write_io(short int,char *);
extern void far KN_read_io(short int,char *);

static char puffer;

task_2_proc()
{
 while(1)
 {
  short int port_2=0x302;

  KN_read_io(port_2,&puffer);
  KN_write_io(port_2,&puffer);
 }
}
```

Bild 109: Anwender-Modul task_2 (Datei task_2.c)

4. Module der Task_3:

```
          NAME start_3

PUBLIC start_3_proc
EXTRN  task_3_proc:NEAR

DATA SEGMENT RW PUBLIC
DD ?
DATA ENDS

STACK STACKSEG 100H

CODE32 SEGMENT ER PUBLIC

start_3_proc PROC  NEAR

      call task_3_proc      ;Aufruf des Anwender-Moduls
                            ;der Task_3
start_3_proc ENDP

CODE32 ENDS
       END start_3_proc,DS:DATA,SS:STACK
```

Bild 110: Hauptmodul start_3 (Datei start_3.src)

```
extern void far KN_write_io(short int,char *);
extern void far KN_read_io(short int,char *);

static char puffer;

task_3_proc()
{
 while(1)
 {
  short int port_3=0x303;

  KN_read_io(port_3,&puffer);
  KN_write_io(port_3,&puffer);
 }
}
```

Bild 111: Anwender-Modul task_3 (Datei task_3.c)

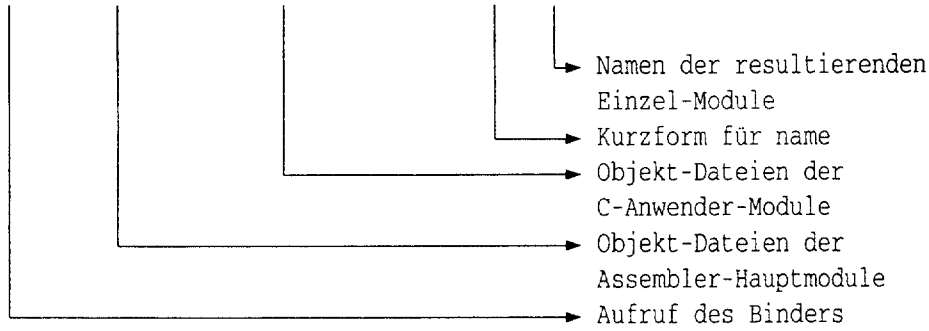
Für die Realisierung des Multitasking-Systems übernimmt der System-Builder BLD386/486 nahezu die gesamte Arbeit. Bevor er aber seine Arbeit aufnehmen kann, muß zuerst von **jeder** Task ein **einzelnes** Objekt-Modul erzeugt werden. Diese Arbeit erledigt der Binder BND386/486 mit folgenden Kommandos:

1. Start_Task:

```
BND386 main_sta.obj,main.obj  noload na(Anw_Start)
```

2. Anwender-Tasks:

```
BND386 start_0.obj,task_0.obj noload na(anw_task_0)
BND386 start_1.obj,task_1.obj noload na(anw_task_1)
BND386 start_2.obj,task_2.obj noload na(anw_task_2)
BND386 start_3.obj,task_3.obj noload na(anw_task_3)
```



3. **OS_Kern_Task**:

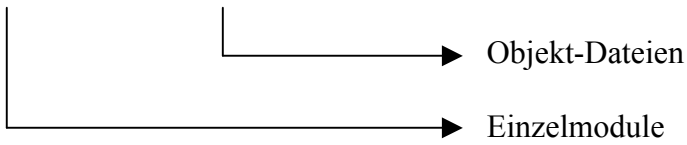
BND386 task_ur.obj,init.obj,ioinit.obj,task_ini.obj, & <task.co1

Inhalt der Datei task.co1:

```
adapt00.obj, &
adapt01.obj, &
adapt02.obj, &
schedule.obj, &
dispatch.obj, &
taskwech.obj, &
einkette.obj, &
auskette.obj, &
read_io.obj, &
write_io.obj, &
round_ro.obj, &
mlf.obj, &
timslice.obj, &
t0_mlf.obj, &
t1_mlf.obj, &
t2_mlf.obj, &
t3_mlf.obj, &
t0_roro.obj, &
t1_roro.obj, &
t2_roro.obj, &
t3_roro.obj, &
int_hand.obj, &
fehl_han.obj, &
noload na(OS_Kern)
```

BND386 erzeugt **sechs** Einzelmodule und speichert sie in den folgenden Objekt-Dateien:

- Anw_Start in main_start.lnk
- anw_task_0 in start_0,lnk
- anw_task_1 in start_1.lnk
- anw_task_2 in start_2.1nk
- anw_task_3 in start_3.1nk
- OS_Kern in task_ur.lnk



Diese Dateien werden nun zusammen mit der **Build**-Datei **mtask**.bld, dem System-Builder mit folgendem Kommando zugeführt:

bld386 task_ur.lnk,main_start.lnk,start_0.1nk, & <task.co2

Inhalt der Datei task.co2:

```
start_1.lnk, &
start_2.1nk, &
start_3.lnk &
bf(mtask.bld) oj(mtask)
```

So erzeugt der System-Builder einen ablauffähigen Code und hinterlegt ihn im ladbaren Objekt-Modul **mtask**.
In der Build-Datei ist das sogenannte **Build-Programm** gespeichert, das die System-Spezifikationen zur Strukturierung des Multitasking-Systems enthält. Siehe folgendes Bild 112.

Hinweis: Die Elemente und die Syntax der deklarativen Sprache BLD386 zur Entwicklung eines Build-Programms sind im Buch „ *Echtzeit-Multitasking – Memory Management und System Design im Protected Mode der x86/Pentium-Architektur*“ (SHAKER Verlag) erklärt.

```
Build_Programm;                                  --Name des Systemprogramms

--*****************************************      --Neue Segmente kreieren

CREATESEG Alias_Task_0_TSS,
          Alias_Task_1_TSS,
          Alias_Task_2_TSS,
          Alias_Task_3_TSS;

SEGMENT --*********************************       Definition der Segmente

                                                 --Segmente der OS_Kern_Task

OS_Kern.CODE32 (DPL=0,USE32,                     --Nur ausfuehrbares
                NOT READABLE,NOT CONFORMING),    --USE32-Segment

OS_Kern.STACK  (DPL=0,USE32,                     --Les- und beschreibbares
                WRITABLE,NOT EXPANDDOWN,         --USE32-Stacksegment
                ALIGN=DWORD),
OS_Kern.DATA   (DPL=0,USE32,                     --Les- und beschreibbares
                WRITABLE,NOT EXPANDDOWN,         --USE32-Datensegment
                ALIGN=DWORD),
                                                 --Segmente der Start_Task

Anw_Start.CODE32 (DPL=3,USE32,                   --Nur ausfuehrbares
                NOT READABLE,NOT CONFORMING),    --USE32-Segment

Anw_Start.STACK  (DPL=3,USE32,                   --Les- und beschreibbares
                WRITABLE,NOT EXPANDDOWN,         --USE32-Stacksegment
                ALIGN=DWORD),
Anw_Start.DATA   (DPL=3,USE32,                   --Les- und beschreibbares
                WRITABLE,NOT EXPANDDOWN,         --USE32-Datensegment
                ALIGN=DWORD),

                                                 --Segmente der Task_0

Anw_task_0.CODE32 (DPL=3,USE32,                  --Nur ausfuehrbares
                NOT READABLE,NOT CONFORMING),    --USE32-Segment
Anw_task_0.STACK  (DPL=3,USE32,                  --Les- und beschreibbares
                WRITABLE,NOT EXPANDDOWN,         --USE32-Stacksegment
                ALIGN=DWORD),
Anw_task_0.DATA   (DPL=3,USE32,                  --Les- und beschreibbares
                WRITABLE,NOT EXPANDDOWN,         --USE32-Datensegment
                ALIGN=DWORD),

                                                 --Segmente der Task_1

Anw_task_1.CODE32 (DPL=3,USE32,                  --Nur ausfuehrbares
                NOT READABLE,NOT CONFORMING),    --USE32-Segment
Anw_task_1.STACK  (DPL=3,USE32,                  --Les- und beschreibbares
                WRITABLE,NOT EXPANDDOWN,         --USE32-Stacksegment
                ALIGN=DWORD),
Anw_task_1.DATA   (DPL=3,USE32,                  --Les- und beschreibbares
                WRITABLE,NOT EXPANDDOWN,         --USE32-Datensegment
                ALIGN=DWORD),
```

```
                                          --Segmente der Task_2

Anw_task_2.CODE32 (DPL=3,USE32,           --Nur ausfuehrbares
                NOT READABLE,NOT CONFORMING), --USE32-Segment
Anw_task_2.STACK  (DPL=3,USE32,           --Les- und beschreibbares
                WRITABLE,NOT EXPANDDOWN,  --USE32-Stacksegment
                ALIGN=DWORD),
Anw_task_2.DATA   (DPL=3,USE32,           --Les- und beschreibbares
                WRITABLE,NOT EXPANDDOWN,  --USE32-Datensegment
                ALIGN=DWORD),

                                          --Segmente der Task_3

Anw_task_3.CODE32 (DPL=3,USE32,           --Nur ausfuehrbares
                NOT READABLE,NOT CONFORMING), --USE32-Segment
Anw_task_3.STACK  (DPL=3,USE32,           --Les- und beschreibbares
                WRITABLE,NOT EXPANDDOWN,  --USE32-Stacksegment
                ALIGN=DWORD),
Anw_task_3.DATA   (DPL=3,USE32,           --Les- und beschreibbares
                WRITABLE,NOT EXPANDDOWN,  --USE32-Datensegment
                ALIGN=DWORD),

--*******************************************   Definition der TSS-Alias-
                                          --Datensegmente

Alias_Task_0_TSS  (DPL=3,USE32,           --Les- und beschreibbares
                LIMIT=67H,                --USE32-Alias-Datensegment
                WRITABLE,NOT EXPANDDOWN,  --der TASK_0
                ALIGN=DWORD),
Alias_Task_1_TSS  (DPL=3,USE32,           --Les- und beschreibbares
                LIMIT=67H,                --USE32-Alias-Datensegment
                WRITABLE,NOT EXPANDDOWN,  --der TASK_1
                ALIGN=DWORD),
Alias_Task_2_TSS  (DPL=3,USE32,           --Les- und beschreibbares
                LIMIT=67H,                --USE32-Alias-Datensegment
                WRITABLE,NOT EXPANDDOWN,  --der TASK_2
                ALIGN=DWORD),
Alias_Task_3_TSS  (DPL=3,USE32,           --Les- und beschreibbares
                LIMIT=67H,                --USE32-Alias-Datensegment
                WRITABLE,NOT EXPANDDOWN,  --der TASK_3
                ALIGN=DWORD);

TABLE --*************************************   Definition der lokalen
                                          --Systemtabellen

Start_LDT    (DPL=0,PRESENT,              --Lokale Deskriptor-Tabelle
                                          --der Start_Task
                RESERVE=(0..1),
          ENTRY = (3:Anw_Start.CODE32,    --Offset=18H
                   4:Anw_Start.STACK,     --Offset=20H
                   5:Anw_Start.DATA)),    --Offset=28H
```

```
Task_0_LDT   (DPL=0,PRESENT,                    --Lokale Deskriptor-Tabelle
                                                --der Task_0
                RESERVE=(0..1),
           ENTRY = (3:Anw_task_0.CODE32,        --Offset=18H
                    4:Anw_task_0.STACK,         --Offset=20H
                    5:Anw_task_0.DATA)),        --Offset=28H

Task_1_LDT   (DPL=0,PRESENT,                    --Lokale Deskriptor-Tabelle
                                                --der Task_1
                RESERVE=(0..1),
           ENTRY = (3:Anw_task_1.CODE32,        --Offset=18H
                    4:Anw_task_1.STACK,         --Offset=20H
                    5:Anw_task_1.DATA)),        --Offset=28H

Task_2_LDT   (DPL=0,PRESENT,                    --Lokale Deskriptor-Tabelle
                                                --der Task_2
                RESERVE=(0..1),
           ENTRY = (3:Anw_task_2.CODE32,        --Offset=18H
                    4:Anw_task_2.STACK,         --Offset=20H
                    5:Anw_task_2.DATA)),        --Offset=28H

Task_3_LDT   (DPL=0,PRESENT,                    --Lokale Deskriptor-Tabelle
                                                --der Task_3
                RESERVE=(0..1),
           ENTRY = (3:Anw_task_3.CODE32,        --Offset=18H
                    4:Anw_task_3.STACK,         --Offset=20H
                    5:Anw_task_3.DATA));        --Offset=28H

TASK --************************************      Definition der Task-Status-
                                                --Segmente
OS_Kern_Task (DPL=0,PRESENT,
              CODE=task_ur_proc,
              DATA=OS_Kern.DATA,
              STACKS=(OS_Kern.STACK),
                                                --Kein LDT-Selektor
              IOPRIVILEGE=0,                    --IOPL=0
              NOT INTENABLED,                   --Interrupt gesperrt
              INITIAL),                         --Initialisierungs_Task

Start_Task    (DPL=0,PRESENT,
              CODE=main_start,
              DATA=Anw_Start.DATA,
              STACKS=(Anw_Start.STACK,
                      OS_Kern.Stack),
              LDT=Start_LDT,                    --LDT-Selektor der Start_Task
              IOPRIVILEGE=0,                    --IOPL=0
              NOT INTENABLED),                  --Interrupt gesperrt
```

```
Task_0        (DPL=0,PRESENT,
               CODE=start_0_proc,
               DATA=Anw_task_0.DATA,
               STACKS=(Anw_task_0.STACK,
                       OS_Kern.STACK),
               LDT=TASK_0_LDT,                 --LDT-Selektor der Task_0
               IOPRIVILEGE=0,                  --IOPL=0
               INTENABLED),                    --Interrupt zugelassen

Task_1        (DPL=0,PRESENT,
               CODE=start_1_proc,
               DATA=Anw_task_1.DATA,
               STACKS=(Anw_task_1.STACK,
                       OS_Kern.STACK),
               LDT=TASK_1_LDT,                 --LDT-Selektor der Task_1
               IOPRIVILEGE=0,                  --IOPL=0
               INTENABLED),                    --Interrupt zugelassen

Task_2        (DPL=0,PRESENT,
               CODE=start_2_proc,
               DATA=Anw_task_2.DATA,
               STACKS=(Anw_task_2.STACK,
                       OS_Kern.STACK),
               LDT=TASK_2_LDT,                 --LDT-Selektor der Task_2
               IOPRIVILEGE=0,                  --IOPL=0
               INTENABLED),                    --Interrupt zugelassen

Task_3        (DPL=0,PRESENT,
               CODE=start_3_proc,
               DATA=Anw_task_3.DATA,
               STACKS=(Anw_task_3.STACK,
                       OS_Kern.STACK),
               LDT=TASK_3_LDT,                 --LDT-Selektor der Task_3
               IOPRIVILEGE=0,                  --IOPL=0
               INTENABLED);                    --Interrupt zugelassen

ALIAS  --**********************************    Verbindung zwischen den
                                               --Alias-Segmenten und den
                                               --Task-Status-Segmenten
         Task_0(Alias_Task_0_TSS),
         Task_1(Alias_Task_1_TSS),
         Task_2(Alias_Task_2_TSS),
         Task_3(Alias_Task_3_TSS);

GATE   --**********************************    Definition der Gates

     KN_create_task(CALL,DPL=3,WC=3,USE32,ENTRY=adapt00),
     KN_read_io(CALL,DPL=3,WC=3,USE32,ENTRY=adapt01),
     KN_write_io(CALL,DPL=3,WC=3,USE32,ENTRY=adapt02),
     timer_interrupt_gate(INTERRUPT,DPL=3,USE32,ENTRY=interrupt_handler),
     fehler_interrupt_gate(INTERRUPT,DPL=3,USE32,ENTRY=fehler_handler),
     KN_start_scheduler(TASK,DPL=3,ENTRY=OS_Kern_Task);
```

```
MEMORY --**********************************   Definition des nicht
                                            --verfuegbaren Speicherbereichs
         (RESERVE = (0..5FFFH,
                    80000H..0FFFFFFFFH));

TABLE --**********************************   Definition der Interrupt-
                                            --Tabelle
IDT
  (DPL=0,PRESENT,
   RESERVE = (21H..26H),
   ENTRY   = (20H:timer_interrupt_gate,
              27H:fehler_interrupt_gate));

TABLE --**********************************   Definition der globalen
                                            --Systemtabelle

GDT
  (DPL=0,PRESENT,
   RESERVE= (3..15),                        --belegt
   ENTRY  = (16:Alias_Task_0_TSS,           --Offset  80H
             17:Alias_Task_1_TSS,           --Offset  88H
             18:Alias_Task_2_TSS,           --Offset  90H
             19:Alias_Task_3_TSS,           --Offset  98H
             20:OS_Kern_Task,               --Offset  A0H
             21:Start_Task,                 --Offset  A8H
             22:Task_0,                     --Offset  B0H
             23:Task_1,                     --Offset  B8H
             24:Task_2,                     --Offset  C0H
             25:Task_3,                     --Offset  C8H
             26:Start_LDT,                  --Offset  D0H
             27:Task_0_LDT,                 --Offset  D8H
             28:Task_1_LDT,                 --Offset  E0H
             29:Task_2_LDT,                 --Offset  E8H
             30:Task_3_LDT,                 --Offset  F0H
             31:OS_Kern.CODE32,             --Offset  F8H
             32:OS_Kern.DATA,               --Offset 100H
             33:OS_Kern.STACK,              --Offset 108H
             34:KN_create_task,             --Offset 110H
             35:KN_start_scheduler,         --Offset 118H
             36:KN_read_io,                 --Offset 120H
             37:KN_write_io));              --Offset 128H
 END;
```

Bild 112: BLD386/486-Spezifikationen des Multitasking-Systems (Datei mtask.bld)

Wir wollen das Build-Programm als Grundausstattung eines Implementierungsbausatzes auffassen und ihm bei Bedarf neue Spezifikationen hinzufügen.