UniMOS

Das modulare Softwarepaket für universelle Multitasking Anwendungen

User Manual

UM-UNIMOS-V1.1

UniMOS Garantiebedingungen

Wir garantieren für einen Zeitraum von einem Jahr, daß UniMOS im wesentlichen gemäß diesem Benutzerhandbuch arbeitet. Etwaige gesetzliche Gewährleistungs- oder Haftungsansprüche gegen den Händler, von dem Sie Ihr Exemplar bezogen haben, werden hierdurch weder ersetzt noch beschränkt.

Es besteht keine Haftung für Folgeschäden.

Die gesamte Haftung seitens des Herstellers und alleiniger Anspruch des Kunden besteht nach Wahl des Herstellers entweder in der Rückerstattung des bezahlten Preises oder in der Reparatur oder dem Ersatz von UniMOS. Der Hersteller schließt für sich jede weitere Gewährleistung bezüglich der Software und des zugehörigen Handbuches aus. Weder der Hersteller noch der Lieferant sind für irgendwelche Schäden ersatzpflichtig, die aufgrund der Benutzung von UniMOS oder der Unfähigkeit UniMOS zu benutzen, entstehen. Das gilt auch für Schäden aus entgangenem Gewinn, Betriebsunterbrechung, Verlust von geschäftlichen Informationen oder Daten oder anderen finanziellen Verlust, auch wenn der Hersteller von der Möglichkeit eines solchen Schadens unterrichtet worden ist. Dieser Ausschluß gilt nicht für Schäden, die durch Vorsatz oder grobe Fahrlässigkeit seitens des Herstellers oder des Lieferanten verursacht wurden. Ebenfalls bleiben solche Ansprüche unberührt, die auf unabdingbaren gesetzlichen Vorschriften zur Produkthaftung beruhen.

Wichtiger Hinweis: Dieses Handbuch darf nicht ohne schriftliche Zustimmung des Autors vervielfältigt werden.

Michael Krämer
Edenkobener Weg 24
40229 Düsseldorf
Tel.: +49 - 211 - 72 26 47
Email: KraemerM@Compuserve.Com

Inhaltsverzeichnis

1. Einführung 5
2. UniMOS Übersicht 6
  2.1. Taskzustände 7
  2.2. Systemvoraussetzungen 9
  2.3. Speichermodelle und Parameterübergabe 10
  2.4. Der Systemtimer 11
  2.5. Intertask Kommunikation 11
    2.5.1. Interrupts 12
    2.5.2. Ereignisse 13
    2.5.3. Nachrichten 14
    2.5.4. Resourcen 15
  2.6. Speicherverwaltung 17
  2.7. UniMOS in DOS Umgebung 18
3. Funktionsbeschreibungen 21
  3.1. Hilfs- und Testfunktionen 59
  3.2. Debugfunktionen 60
4. Anhang 61
  4.1. UniMOS Publics und Externals 61
  4.2. Speicherbelegung und Speicherbedarf 62
  4.3. Die Konfigurationstabelle 64
    4.3.1. Die Elemente der Konfigurationstabelle 64
  4.4. Der System Control Block (SCB) 65
    4.4.1. Die Elemente des SCB 65
  4.5. Der Task Control Block (TCB) 66
    4.5.1. Die Elemente des TCB 67
  4.6. Die READY-Liste 68
  4.7. Der Nachrichtenblock 68
  4.8. Der Resourcen-Kontrollblock 69
  4.9. Der Memory-Pool Kontrollblock 70
  4.10. Fehlercodes 70
  4.11. Leistungsdaten 73

1. Einführung

Insbesondere bei Controller-Anwendungen im industriellen Bereich haben sich Multitasking-Betriebssysteme seit einiger Zeit durchgesetzt. Mit Ihrer Hilfe können mehrere Aufgaben eines Systems quasi gleichzeitig abgearbeitet werden. Da dem System jedoch nur ein Prozessor zur Verfügung steht, muß dieser nacheinander alle notwendigen Aufgaben erledigen. Die Umschaltung zwischen den Aufgaben, den sogenannten Tasks, geschieht dabei so schnell, daß man von einer gleichzeitigen Ausführung sprechen kann.

Ein Multitasking-Kern stellt Funktionen zur Verfügung, die unter anderem die Umschaltung der Tasks vornehmen. Man unterscheidet anhand des Verfahrens das zum Taskwechsel zur Anwendung kommt, zwischen einem zeitscheibengesteuerten und einem ereignisgesteuerten Multitasking-System.

Die meisten Multitasking-Betriebssysteme für PCs, Workstations und Großrechner sind zeitscheibengesteuert (MVS, UNIX, OS/2). Das bedeutet, daß jeder Task in einem solchen Systems den Prozessor nur für einen bestimmten vordefinierten Zeitraum erhält. Danach wird der Prozessor zum nächsten Task weitergeschaltet, egal ob der vorherige fertig war oder nicht und egal ob er wichtiger oder unwichtiger ist. Dieses Verhalten ist bei einem Rechner, der mehrere Tasks (Benutzer) bedienen muß, normalerweise unumgänglich und gerecht, da so niemand den Rechner für sich alleine beanspruchen kann.

Für einen industriellen Controller ist dieses Verhalten jedoch meist unerträglich, da hier die Priorität von Tasks oft sehr starr festgelegt ist und ein Task mit hoher Priorität normalerweise nicht von einem solchen mit niedriger Priorität unterbrochen werden darf.

Betrachten wir dazu als Beispiel einen normalen PKW, dessen Hersteller sich dazu entschlossen hat, alle elektronischen Funktionen von einem Prozessor ausführen zu lassen (er wird das vermutlich aus Sicherheitsgründen in der Praxis nicht tun!). Zur Zuteilung des Prozessors an die verschiedenen Tasks setzt er einen ereignisgesteuerten Betriebssystem-Kern ein, der folgendermaßen funktionieren könnte: Die wichtigste Aufgabe ist die Steuerung des Antiblockiersystems der Bremsen. Der zugehörige Task wird die höchste Priorität des Systems erhalten. Damit kann er von keinem anderen Task unterbrochen werden und er erhält den Prozessor solange, bis er ihn selbständig wieder abgibt. Die zweitwichtigste Aufgabe ist die Motorsteuerung. Hier muß jeweils zum exakt richtigen Zeitpunkt der Treibstoff eingespritzt werden. Im Normalbetrieb ist dies die wichtigste Aufgabe, da sie den optimalen Lauf des Motors garantiert. Bei einer Vollbremsung muß diese Aufgabe jedoch zurückgestellt werden. Aufgaben mit niedrigerer Priorität wären z.B. die elektronische Steuerung von Außenspiegeln, die Zentralverriegelung oder die Klimaregelung.

Das Beispiel macht deutlich, daß die Aktionen eines Controllers fast immer durch äußere Ereignisse angestoßen werden, worauf dann die Reaktionen unverzüglich erfolgen müssen (Bremse betätigen ⇒ bremsen, Kolben an der errechneten Position ⇒ einspritzen, ein Schloß bedienen ⇒ alle Schlösser öffnen oder schließen). Daher nennt man ein solches Multitasking System "ereignisgesteuert" oder "event driven". Da die Reaktionen in einer genau im voraus zu bestimmenden Zeit eingeleitet werden können, spricht man auch von einem Echtzeitsystem.

2. UniMOS Übersicht

Wichtige Eigenschaften von UniMOS:

- Verwendbar auf allen Systemen mit 8088-kompatiblem Befehlssatz (PC und NEC-V-Serie)
- optimierte Version für Prozessoren mit 80188-kompatiblen Befehlssatz (80188/186, 80286, 80386, 80486 und NEC V-Serie)
- parametrierbar für verschiedene Systemgrößen
  (Die maximale Anzahl der Tasks, Prioritäten, Mailboxen und Resourcen ist praktisch nur durch den verfügbaren Speicher begrenzt)
- Intertask Kommunikation
- Speicher Management Funktionen
- Resource-Management-Funktionen
- Interrupt-Behandlungs-Funktionen
- "timeouts" für wartende Funktionen
- Echtzeit und /oder Zeitscheibensteuerung
- sehr schnell durch Programmierung in Assembler
- Funktionsbibliotheken für Borland- und Microsoft-Tools
- für die Speichermodelle Small, Medium, Compact, Large und Huge
- auch in DOS-Umgebung lauffähig

UniMOS ist als (Assembler-) Quelltext (.ASM) sowie als Bibliotheksdatei (.LIB) auf MS-DOS kompatiblen Disketten (3 1/2" und 5 1/4") erhältlich. Der Assembler Quelltext ist für den Turbo-Assembler geschrieben. Die Bibliotheken sind mit Borland- und Microsoft-Werkzeugen kompatibel. Die Parameter für die Systemfunktionen werden nach dem C-Standard übergeben. Damit kann UniMOS direkt in C- und Assembler-Programme eingebunden werden. Andere Programmiersprachen können auch benutzt werden, sofern sie diese Konventionen unterstützen.

Da UniMOS für die Verwendung in Controller-Umgebungen ohne DOS konzipiert wurde, ist auch ein Umwandlungsprogramm notwendig, das eine .EXE-Datei in ein Format umwandelt, das von einem normalen Programmiergerät eingelesen werden kann. Hier hat sich als Standard das *Intel-HEX* und das *Extended Intel-Hex* Format durchgesetzt. Ein solches Umwandlungsprogramm mit dem Namen "UniLOC" ist bei Ihrem Händler verfügbar. Damit ist der UniMOS-Kern und die Anwendungs-Software an jedes beliebige Segment innerhalb des 1 MB Adressraumes relokierbar.

UniMOS ist ein Multitasking-Betriebssystem, das sowohl prioritäts- als auch zeitscheibengesteuerte Taskumschaltung erlaubt. UniMOS wurde in erster Linie für "Embedded Control" Systeme entwickelt, jedoch sind auch Anwendungen unter MS-DOS möglich (siehe dazu das Kapitel "UniMOS in DOS Umgebung" auf Seite 18). Der Benutzer muß beim Starten des Kernes unter anderem die maximale Anzahl der Tasks sowie die maximale Anzahl der Task-Prioritäten angeben.

Jedem Task muß eine Priorität zugeordnet werden. Es können mehrere Tasks mit gleicher Priorität existieren, wodurch der Prozessor dann in einem Zeitscheiben-Verfahren zwischen den Tasks aufgeteilt wird. Wird andererseits jeder Priorität nicht mehr als ein Task zugeordnet, so ist das System prioritätsgesteuert. In diesem Falle läßt sich vorhersagen, wie lange das System für die Reaktion auf ein externes Ereignis braucht. Für UniMOS gilt immer, daß ein höher priorisierter Task niemals von einem Task mit niedrigerer Priorität unterbrochen werden kann. Falls er den Prozessor nicht selbständig (z.B. durch Warten) wieder abgibt, dann kann nur ein Task mit gleicher oder höherer Priorität ihn unterbrechen (abgesehen von Interrupt-Routinen, die jedoch keine Tasks sind). Wenn ein höherpriorisierter Task den Prozessor benötigt, so erhält er ihn (fast) sofort, ein Task mit gleicher Priorität jedoch erst nach Ablauf der momentanen Zeitscheibe.

Zeitscheibengesteuerte Tasks erhalten den Prozessor immer nur für eine bestimmte (beim Starten des Tasks vordefinierte) "Zeitscheibe". Wenn diese Zeitscheibe abgelaufen ist, dann kommt der nächste Task an die Reihe. Normalerweise kann also kein Task den Prozessor länger behalten, als es für ihn vorher definiert wurde. Um Zeitscheibensteuerung zu gestatten, muß ein System-Timer installiert sein (Seite 11). Jeder Aufruf dieses Systemtimers entspricht dann einem Systemtakt, der die Einheit für alle zeitabhängigen Steuerungen darstellt. Das heißt für die Zeitscheibensteuerung, daß die Dauer einer Zeitscheibe in "Anzahl Systemtakte" angegeben wird.

2.1. Taskzustände

Ein Task kann sich, während das System läuft, in verschiedenen Zuständen befinden. Ein Übergang zwischen den Zuständen kann dabei nur über vordefinierte Systemfunktionen erfolgen. Diese Zustände und die wichtigsten Übergangsmöglichkeiten sind hier grafisch dargestellt und unten erläutert.

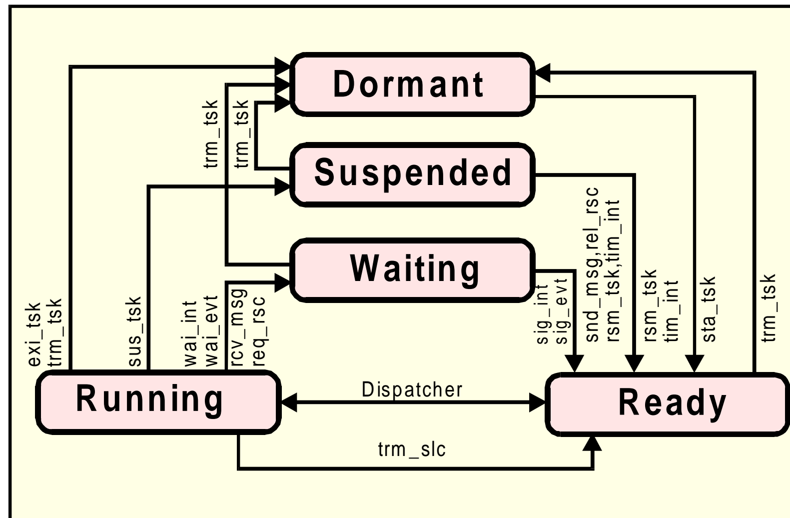


Bild 1: Das UniMOS Zustandsdiagramm

Nach dem Starten des Systems befinden sich alle Tasks außer dem Hintergrundtask im Zustand ***DORMANT***. In diesem Zustand "schläft" der Task. Er bekommt zu keiner Zeit den Prozessor zugeteilt, jedoch sind alle notwendigen Kontrollblöcke angelegt. Damit belegt dieser Task Speicherplatz, kostet aber keine CPU-Zeit.

Wenn ein Task im Zustand ***RUNNING*** ist, dann hat er den Prozessor zur Verfügung.

Ein Task im ***READY*** Zustand konkurriert um die CPU. Wenn ein Task ***READY*** ist, so bedeutet das, daß er den Prozessor haben möchte. Ob er ihn bekommt, entscheidet eine Systemroutine, der ***Dispatcher***. Der Dispatcher wählt aus allen Tasks, die den Prozessor haben wollen, die also READY sind, denjenigen aus, der die höchste Priorität hat. Wenn mehrere Tasks die gleiche Priorität haben, dann wird bei jedem Timer Interrupt die Zeitscheibe des gerade ausgeführten Tasks dekrementiert. Wenn null erreicht wird, dann wird der momentane Task an das Ende der Warteschlange angehängt und der nächste Task mit gleicher Priorität erhält den Prozessor für die beim Starten des Tasks vorgesehene Zeitscheibe. Dieses Verfahren bezeichnet man als ***Round Robin***

In den Zustand ***SUSPENDED*** gelangt ein Task, wenn er für eine bestimmte oder unbestimmte Zeit nicht benötigt wird. Der Zustand ist damit ähnlich zu DORMANT (schlafend), jedoch kann ein suspendierter Task später wieder weiterlaufen, während er aus dem Zustand DORMANT neu gestartet werden muß. Der schlafende Task hat im Gegensatz zum suspendierten Task keinen gültigen Stack. Ist der Task für eine bestimmte Zeit suspendiert, so läuft ein Timer mit, nach dessen Ablauf der Task automatisch wieder nach READY überführt wird.

Gerade bei Steuerungen muß häufig eine bestimmte Reaktion auf ein Ereignis erfolgen. Der Prozessor wird also nur dann benötigt, wenn dieses Ereignis eingetreten ist, das sich entweder über einen Interrupt bemerkbar macht oder von einem anderen Task ausgelöst werden kann. Im Zustand ***WAITING*** wartet ein Task auf ein solches Ereignis. Um bei ausbleibendem Ereignis die Möglichkeit einer Fortsetzung zu haben, kann man entweder ein "Timeout" spezifizieren oder den Task von einem anderen Task aus wieder in Gang setzen.

Welche Möglichkeiten bietet UniMOS, um zwischen den einzelnen Task-Zuständen zu wechseln? Die möglichen Übergänge sind in Abbildung 1 detailliert beschrieben.

Aus dem DORMANT-Zustand kommt ein Task durch die Systemfunktion ***sta_tsk*** in den READY-Zustand. ***sta_tsk*** kann von jedem anderen Task, also auch von dem Hintergrundtask (siehe unten) aufgerufen werden. Am Ende der Funktion ***sta_tsk*** wird der Dispatcher aufgerufen, eine Systemroutine, die entscheidet, welcher

derjenigen Tasks, die READY sind, nach RUNNING überführt wird. Ist also der gerade gestartete Task höher priorisiert als der startende, so läuft der neue Task sofort los. Will man das verhindern, um z.B. ungestört die Initialisierung zu beenden, so kann man zeitweilig die Taskumschaltung mit der Funktion ***dis_pre*** abschalten. Durch ***sta_tsk*** erfährt UniMOS die Startadresse des Tasks und die Adresse des für ihn exclusiv reservierten Stackbereiches, sowie seine Priorität. Auch die Segmentadresse des Datenbereiches wird hier angegeben. Bei zeitscheibengesteuerten Tasks wird außerdem die Dauer einer Zeitscheibe spezifiziert.

Der Aufruf des Dispatchers ist für den Programmierer völlig transparent. Jede Systemfunktion, die einen Taskzustand von oder nach READY oder RUNNING ändert, ruft automatisch den Dispatcher auf. Außerdem wird er von der Funktion ***ena_pre*** aufgerufen.

Von READY kann der Task nach RUNNING überführt werden. Das wird durch den Dispatcher erledigt, falls bestimmte Bedingungen erfüllt sind. Zunächst muß auf jeden Fall die Taskumschaltung gestattet sein (siehe ***ena_pre***) und es darf keinen Task mit höherer Priorität im System geben. Alle Tasks gleicher Priorität sind linear verkettet, so daß der Dispatcher immer den ersten in dieser Kette auswählt. Wenn kein höher priorisierter Task READY wird, dann bleibt dieser Task solange RUNNING, bis seine Zeitscheibe abgelaufen ist. Ein Task kann von einem anderen Task mit der Funktion ***trm_tsk*** von READY nach DORMANT überführt werden.

Ein laufender Task kann sich selbst durch Aufruf der Funktion ***trm_tsk*** oder ***exi_tsk*** beenden, also nach DORMANT überführen. Sein Stackbereich, also sein gesamter Kontext, wird damit frei und der Task kann nicht mehr weitergeführt, sondern nur noch neu gestartet werden. Vorsicht ist bei Resourcen und Speicherblöcken angebracht. Sie werden nicht automatisch freigegeben und sind damit bis zum Neustart des Systems verloren.

Die Funktion ***trm_slc*** erlaubt einem Task, seine momentane Zeitscheibe aufzugeben. Das ist nur für Zeitscheiben-Tasks sinnvoll, wenn also weitere Tasks gleicher Priorität READY sind. Der jetzige Task wird an das Ende der Warteschlange angehängt und der nächste Task gleicher Priorität erhält den Prozessor. ***trm_slc*** wird ignoriert, wenn kein weiterer Task gleicher Priorität READY ist. Die Zeitscheibe des Tasks, der ***trm_slc*** aufruft, wird nicht neu geladen, d.h. daß der Task nur noch für die verbleibende Zeit wieder den Prozessor erhält. ***trm_slc*** gestattet kurzzeitiges Warten, ohne in den WAITING oder SUSPENDED Zustand überzugehen.

Mit der Funktion ***sus_tsk*** kann sich ein Task zeitweise suspendieren. Er behält seinen Kontext und kann damit hinter dem ***sus_tsk***-Funktionsaufruf weitergeführt werden. Der Task kann entweder automatisch nach Ablauf eines Timeouts oder durch die Funktion ***rsm_tsk*** von einem anderen Task weitergeführt werden. Außerdem kann ein suspendierter Task mit der Funktion ***trm_tsk*** terminiert werden.

In den WAITING Zustand gelangt ein Task immer dann, wenn er auf ein Ereignis warten muß. Wenn das Ereignis eingetreten ist, dann wird der Task wieder nach READY überführt. Externe Ereignisse machen sich durch Interrupts bemerkbar. Ein Task kann mit der Funktion ***wai_int*** auf einen Interrupt warten. Im Interrupt-Handler wird die Funktion ***sig_int*** aufgerufen, um den Interrupt zu signalisieren und den wartenden Task wieder nach READY zu bringen. Ereignisse können innerhalb von Tasks mit ***wai_evt*** und ***sig_evt*** erwartet bzw. weitergemeldet werden. Das kann z.B. auch ein Interrupt sein, der an einen Task signalisiert und von dem dann weitergemeldet wird. Außerdem kann ein Task mit ***rcv_msg*** und ***req_rsc*** auf Nachrichten oder Resourcen warten. Mit ***snd_msg*** und ***rel_rsc*** kann ein anderer Task den wartenden Task aus seinem Zustand befreien.

Beim Warten auf ein Ereignis besteht prinzipiell immer die Möglichkeit, daß dieses Ereignis nicht eintritt. Deshalb gibt es zwei Möglichkeiten das Warten dennoch zu beenden. Alle Funktionen, die einen Task warten lassen können, haben die Möglichkeit, eine maximale Wartezeit, also einen Timeout zu spezifizieren. Wenn diese Zeit abgelaufen ist, ohne daß das Ereignis eintrat, dann wird die ***tim_int*** Funktion den Task mit einem entsprechenden Fehlercode weiterführen. Außerdem kann ein anderer Task auch hier die Funktion ***rsm_tsk*** benutzen um den Task "wiederzubeleben". Auch ein wartender Task kann mit der Funktion ***trm_tsk*** beendet werden. Beachten Sie jedoch auch hier, daß Speicherblöcke oder Resourcen nicht automatisch freigegeben werden. Allerdings wird der Task aus der Warteschlange für die Mailbox oder die Resource herausgenommen.

UniMOS wird durch Aufruf der Funktion ***ini_tsk*** gestartet. Nach dem Reservieren und Initialisieren der Systemtabellen macht ***ini_tsk*** ein normales *Return*. Ab diesem Zeitpunkt ist jedoch der folgende Code ein Task und zwar der sogenannte ***Hintergrund-Task*** mit der niedrigsten Priorität und der höchsten Tasknummer. Der Hintergrund-Task kann weitere Tasks starten, er darf sich selbst jedoch niemals in einen von ***READY*** verschiedenen Zustand setzen (z.B. durch sus_tsk, exi_tsk oder wai_int). Wenn er das dennoch versucht, so wird ein "fataler Fehler" generiert (siehe "Die Konfigurationstabelle" ab Seite 64) und zu dem entsprechenden Funktion verzweigt. Der Hintergrund-Task wird immer dann ausgewählt, wenn kein anderer Task ***READY*** ist.

Der Hintergrund-Task hat normalerweise keine Systemfunktion zu erfüllen. Deshalb könnte man mit der Assembler-Anweisung "JMP $" eine Endlosschleife programmieren. Zustandsänderungen des Systems werden jedoch immer asynchron über Interrupts gemeldet. Daher kann man den Prozessor in einen Modus mit niedrigem Stromverbrauch setzen, solange dieser durch einen Interrupt wieder aufgehoben werden kann. Der folgende Code ist z.B. als *IDLE*-Code im Hintergrund-Task für ein V25 System verwendbar:

```
        ....
        ini_tsk config_table
        dis_pre             ;Preemption verhindern
        sta_tsk task1,dataseg,stack1,37,1,0
        sta_tsk task2,dataseg,stack3,40,8,1
        sta_tsk task3,dataseg,stack4,41,8,3
        ena_pre             ;Jetzt laufen die gestarteten Tasks los.

;Hier kommt der IDLE-Code. Prozessor in HALT Modus setzen.
idle:   enable_int          ;Interrupts müssen freigegeben sein
        halt                ;Prozessor anhalten. Clock läuft weiter
        nop
        nop
        jmp                 short idle
```

2.2. Systemvoraussetzungen

Der UniMOS Multitasking Kern ist auf allen Prozessoren ablauffähig, die zum Befehlssatz des 8088 Prozessors aufwärtskompatibel sind. Das sind außer dem 8088 und 8086 die Prozessoren 80188/80186, 80286, 80386 und 80486. Außerdem läuft der Code auch auf den NEC V-Serie Prozessoren V20, V30, V40, V50 sowie dem V33 und dem V53. Auch die V-Serie Controller V25 und V35 können UniMOS ausführen. Da alle genannten Prozessoren außer dem 8088 und dem 8086 einige optimierte Befehle haben, wurde eine UniMOS-Version speziell für diese Prozessoren zusammengestellt. Wenn Sie sicher sind, daß Ihr Code nicht auf dem 8088 oder 8086 ausgeführt werden muß, dann sollten Sie diese Version bevorzugen, da sie kleiner und schneller ist. Pinkompatible Austauschtypen zu 8088/8086 sind die NEC-Prozessoren V20 und V30. Da sie aufgrund der internen Struktur bereits bei direktem Ersatz unter sonst gleichen Bedingungen eine Leistungssteigerung bringen, kann ein Austausch des 8088 oder 8086 sowieso empfohlen werden.

Zur Entwicklung der Anwendersoftware wird ein PC mit MS-DOS Betriebssystem (oder dazu kompatibler Umgebung) benötigt. Üblicherweise verwendet man heute die Programmiersprache C oder Assembler zur Entwicklung von Software für Controller. Als Standard-Entwicklungswerkzeuge haben sich die C-Compiler von Borland (Turbo-C und Borland-C) und Microsoft (MS-C) etabliert. Von beiden Firmen werden auch Assembler und Quellcode-Debugger angeboten. Beide genannten Systeme können mit UniMOS zusammenarbeiten. Der UniMOS Quellcode ist in Assembler geschrieben und kann direkt mit dem Turbo-Assembler übersetzt werden. Für den Microsoft Assembler sind einige kleine Änderungen notwendig, da die Makrobehandlung und die Optimierung von Sprunganweisungen etwas unterschiedlich ist. Das ist jedoch nur dann relevant, wenn Sie den Quellcode erwerben wollen. Die UniMOS Funktionen werden einfach mit dem Linker zu den anderen Anwendungsprogrammen hinzugebunden. Soll UniMOS unter DOS laufen und/oder sollen Bibliotheksfunktionen des Compilers benutzt werden, so beachten Sie bitte ganz besonders das Kapitel "UniMOS in DOS Umgebung" ab Seite 18.

Prinzipiell ist UniMOS auch mit anderen Programmiersprachen verwendbar. Voraussetzung ist allerdings, daß eine zu "C" kompatible Parameterübergabe möglich ist. Außerdem müssen für jedes zu verwendende Speichermodell auch *FAR*-Zeiger auf Code und auf Daten verwendbar sein (siehe "Speichermodelle und Parameterübergabe" auf Seite 10). Dies ist deshalb notwendig, da einige UniMOS Parameter unabhängig vom Speichermodell immer *FAR*-Zeiger verwenden.

2.3. Speichermodelle und Parameterübergabe

Aufgrund der segmentierten Architektur der Prozessoren aus der 8088-Familie ergeben sich verschiedene Möglichkeiten zur Speicheradressierung. Zunächst kann man auf jeden Wert im gesamten Adressraum des Prozessors zugreifen, indem man das Segment und den Abstand vom Anfang des Segments, den Offset, spezifiziert. Einen solchen Zeiger, bestehend aus Segment und Offset, bezeichnet man als einen ***FAR***-pointer (weiter Zeiger). Er ist vier Bytes lang und belegt damit zwei Worte. Wenn alle zu adressierenden Elemente in ein Segment passen, dann kann der Zeiger auf ein Wort, den Offset, verkürzt werden. Damit vereinfachen sich die Parameterübergabe und Adressrechnungen erheblich. Einen solchen Zeiger bezeichnet man als einen ***NEAR***-pointer (nahen Zeiger).

Die verschiedenen Speichermodelle unterscheiden sich nun gerade in der Länge der Zeiger. Da die Zeigerlänge für Code und Daten unabhängig voneinander gewählt werden kann, ergeben sich folgende Speichermodelle:

Speichermodell	Daten-Zeiger	Code-Zeiger
Small	near	near
Medium	near	far
Compact	far	near
Large	far	far

Für spezielle Anwendungen gibt es noch weitere Speichermodelle. Wenn ein Datenelement größer als ein Segment (64 kB) werden kann, dann muß man das Speichermodell ***Huge*** verwenden. Damit wird bei jeder Änderung des Pointers auch der Segmentwert angepaßt (Normalisierung). In den anderen Speichemodellen wird jeweils nur der Offset neu berechnet. Das Speichermodell ***Tiny*** kann verwendet werden, wenn das gesamte Programm in ein Segment von 64 kB passt.

Wichtiger Hinweis: Bei den Speichermodellen mit kurzen Datenzeiger muß der Stack immer in demselben Segment wie die Daten liegen. Lokale Variablen auf dem Stack können sonst nicht adressiert werden.

Nach C-Konvention werden Parameter auf dem Stack übergeben und zwar wird der letzte Parameter der Funktion als erstes auf dem Stack abgelegt. Anschließend wird der Stackpointer dekrementiert und soweit vorhanden der nächste Parameter abgelegt. Der Stackpointer zeigt somit immer auf das zuletzt gespeicherte Wort. Nachdem alle Parameter abgelegt sind, wird das Unterprogramm aufgerufen. Die Rücksprungadresse wird als letztes auf dem Stack gesichert. Das aufgerufene Unterprogramm kann auf die Variablen auf dem Stack zugreifen und für eigene Variablen selbst einen Bereich des Stack reservieren. Die Korrektur des Stackpointers muß nach Rückkehr aus dem Unterprogramm vom aufrufenden Programm selbst vollzogen werden.

Rückgabewerte des Unterprogramms stehen im Register AX oder bei einem Doppelwort (z.B. FAR pointer) in DX,AX. In DX steht dann das höherwertige Wort oder der Segmentwert, in AX das niederwertige Wort oder der Offset.

Ein kurzes Assemblerprogramm soll die Verhältnisse am Beispiel des Aufrufes der Funktion ***ini_tsk*** (Beschreibung siehe Seite 34) verdeutlichen. ***ini_tsk*** hat nur einen Parameter und zwar ist den Zeiger auf die Konfigurationstabelle. Dieser Zeiger wird zunächst auf dem Stack abgelegt, dann wird die Funktion aufgerufen und anschließend wird der Stackpointer korrigiert. Der folgende Code gilt für Speichermodelle mit NEAR Zeigern auf Daten (Small oder Medium)

```
push    offset config ;Adresse der Konfigurationstabelle
call    ini_tsk       ;Unterprogramm aufrufen
add     sp,2          ;Stackpointer korrigieren
```

Für ein Speichermodell mit FAR Zeigern auf Daten (Compact, Large oder Huge) würde der Code folgenderma-ßen aussehen:

```
push   seg config    ;Segment der Konfigurationstabelle
push   offset config ;Offset der Konfigurationstabelle
call   ini_tsk       ;Unterprogramm aufrufen
add    sp,4          ;Stackpointer korrigieren
```

In beiden Fällen steht der Rückgabewert im Register AX.

2.4. Der Systemtimer

Für einige Systemfunktionen wird ein Systemtimer benötigt. Dieser Timer ist ein Hardware-Block, der in regelmäßigen Abständen einen Interrupt generiert. Die Initialisierung dieses Timers und die Installation des zugehörigen Interrupt-Handlers muß vom Benutzerprogramm vollzogen werden. Damit liegt die Wahl der Interrupt-Quelle sowie die Rate der Interrupts ganz im Ermessen des Anwenders. Am Ende des Interrupt-Handlers muß lediglich ein Sprung zur UniMOS-Funktion ***tim_int*** ausgeführt werden. Das gestattet UniMOS die Verwaltung von System-Timern zum Erkennen abgelaufener Wartezeiten oder zur Task-Weiterschaltung für die Tasks gleicher Priorität (Round-Robin).

Die Rate der Timer-Interrupts bestimmt, mit welcher Auflösung Systemzeiten definiert werden können. Einige Punkte sind bei der Wahl der Interrupt-Rate zu beachten.

Zunächst muß garantiert werden, daß alle Aufgaben, die der UniMOS Timer-Interrupt-Handler zu erledigen hat, in der ihm zur Verfügung stehenden Zeit bewältigt werden können. Das setzt eine untere Grenze für den Abstand zwischen zwei Timer Interrupts. Abschätzungen über die benötigte Zeit für diese Aufgaben, findet man im Kapitel "Leistungsdaten" ab Seite 73.

Da Programm und Hardware-Timer asynchron zueinander laufen, ist die Genauigkeit von Timer-Angaben grundsätzlich abzüglich einem Timer-Interrupt zu verstehen. Wenn z.B. ***sus_tsk(3)*** aufgerufen wird, so kann die tatsächliche Zeitdauer zwischen 2 und 3 liegen und sie wird im Mittel genau 2,5 sein. Das liegt daran, daß das Programm den genauen Zeitpunkt des Timer-Interrupts nicht kennt. Wenn sofort nach Ende der Funktion ***sus_tsk*** ein Timer-Interrupt auftritt, dann wird quasi ohne daß der Task überhaupt schon suspendiert war, der Timer auf zwei heruntergezählt. Insgesamt wartet der Task also dann nur zwei Timer-Intervalle. Wenn jedoch gerade vor dem Aufruf von ***sus_tsk*** ein Timer-Interrupt stattfand, so wartet der Task tatsächlich drei Timer-Intervalle ab. Die gleichen Überlegungen gelten für zeitscheibengesteuerte Tasks. Im Mittel wird eine Funktion genau in der Mitte des Intervalls zwischen zwei Timer-Interrupts aufgerufen werden, so daß man von den Timer-Angaben 0,5 abziehen kann. Wenn man eine bestimmte mindest-Wartezeit garantieren muß, dann muß man zu dieser Zeit noch ein Intervall hinzuzählen.

2.5. Intertask Kommunikation

Die einzelnen Tasks in einem Multitaskingsystem sind normalerweise nicht so stark voneinander entkoppelt, daß sie völlig unabhängig ausgeführt werden können. In der Regel müssen viele Informationen zwischen den einzelnen Tasks ausgetauscht werden.

Im einfachsten Falle ist das die Mitteilung an einen anderen Task, daß ein gewisses Ereignis eingetreten ist. Ein solches Ereignis kann asynchron erfolgen und es wird dann dem System durch einen Interrupt angezeigt. Der daraufhin aufgerufene Interrupt-handler kann das Ereignis dann an einen Task melden. Wird das Ereignis dagegen von einem Task erkannt, so kann es als "normales" Ereignis (Event) an einen anderen Task weitergemeldet werden.

Wenn die zwischen Tasks ausgetauschte Information umfangreicher ist als nur eine einfache Meldung, dann muß man eine Nachricht versenden, mit der der sendende Task den empfangenden Task zu einer Aktion auffordert und ihm normalerweise die notwendigen Daten zur Verfügung stellt. Die so ausgetauschte Information kann dann eine beliebige Struktur haben, da letzten Endes ein Zeiger übergeben wird. Sende- und Empfangs-

task müssen den Aufbau dieser Struktur kennen. Dennoch muß die Information in einen standardisierten "Umschlag" gesteckt werden, dessen Aufbau im Kapitel "Der Nachrichtenblock" ab Seite 68 beschrieben ist.

Wenn im System Einheiten zur Verfügung stehen, die zur gleichen Zeit nur von einem Task benutzt werden können (sogenannte Resourcen), dann muß auch dafür ein Kommunikationsmechanismus geschaffen werden. UniMOS bietet dafür eine "Belegungsliste", die jederzeit die Verfügbarkeit einer Resource wiederspiegelt und Belegungswünsche nach verschiedenen Algorithmen verwaltet.

2.5.1. Interrupts

Die einfachste Art der Kommunikation ist gegeben, wenn sich ein externes Ereignis durch einen Interrupt bemerkbar macht und daraufhin ein Task informiert werden soll. Das Problem wäre im einfachsten Falle so zu lösen, daß man eine globale Variable definiert, die den Eintritt des Ereignisses anzeigt indem sie von der Interrupt-Routine gesetzt und vom Task regelmäßig abgefragt wird. Der erhebliche Nachteil dieser Methode ist jedoch, daß der Task "aktiv" warten muß, was natürlich Prozessorzeit kostet.

Dieses Problem wird in UniMOS durch die beiden Funktionen ***wai_int*** und ***sig_int*** gelöst.

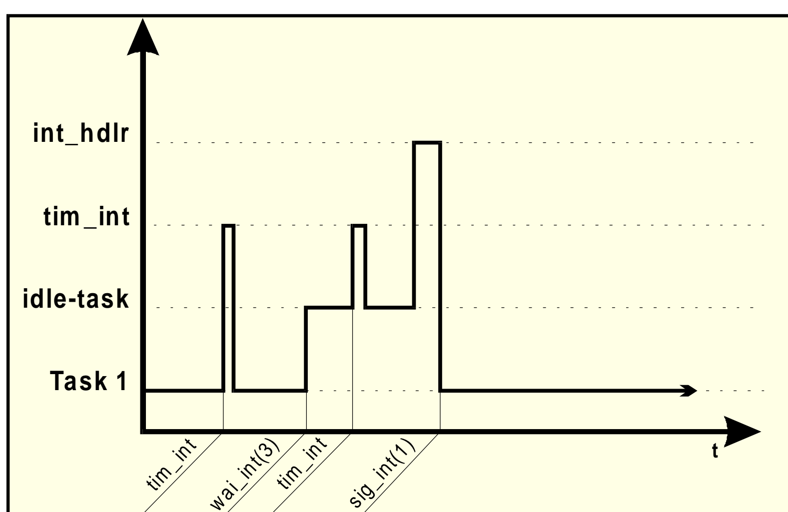


Bild 2: Warten und Signalisieren von Interrupts

Bild 2 zeigt, wie Task 1 mit ***wai_int*** auf einen Interrupt wartet, dessen Eintritt von dem Interrupt-handler mit ***sig_int*** gemeldet wird. Task 1 hat hier einen Timeout von drei Timer Ticks spezifiziert. Diese Funktionen können beispielsweise zum Empfang eines Nachrichtenpakets über eine serielle Schnittstelle sehr sinnvoll eingesetzt werden. Wenn der Task die Ankunft eines solchen Pakets erwartet, dann ruft er ***wai_int*** auf und wartet damit solange, bis das gesamte Paket angekommen und in einem ihm bekannten Speicherbereich abgespeichert wurde. Der Interrupt-handler wird bei jedem empfangenen Zeichen aufgerufen, da von der seriellen Schnittstelle ein Interrupt generiert wird. Er speichert das Zeichen an der ihm bekannten Stelle ab und überprüft, ob es das letzte Zeichen war. Wenn ja, dann signalisiert er dem wartenden Task mit ***sig_int***, daß der komplette Block nun zur Verfügung steht. ***sig_int*** wird also nur nach Empfang des letzten Zeichens aufgerufen, ansonsten wird ganz normal mit *iret* von der Interrupt-Routine zurückgekehrt.

Nun kann es aber gerade bei der seriellen Datenübertragung passieren, daß keine Zeichen empfangen werden, oder zumindest das letzte Zeichen nicht korrekt übertragen wurde. In diesem Falle würden Interrupt-handler und Task für immer auf eben dieses letzte Zeichen warten. Deswegen wurde die Möglichkeit vorgesehen, eine maximale Wartezeit zu spezifizieren. Wenn diese sogenannte "Timeout"-Zeit abgelaufen ist, dann wird der Task automatisch mit einer entsprechenden Fehlermeldung weitergeführt. Ein Timeout wird immer in Anzahl "***tim_int***-Aufrufe" angegeben. Ein Beispiel dazu zeigt Bild 3.

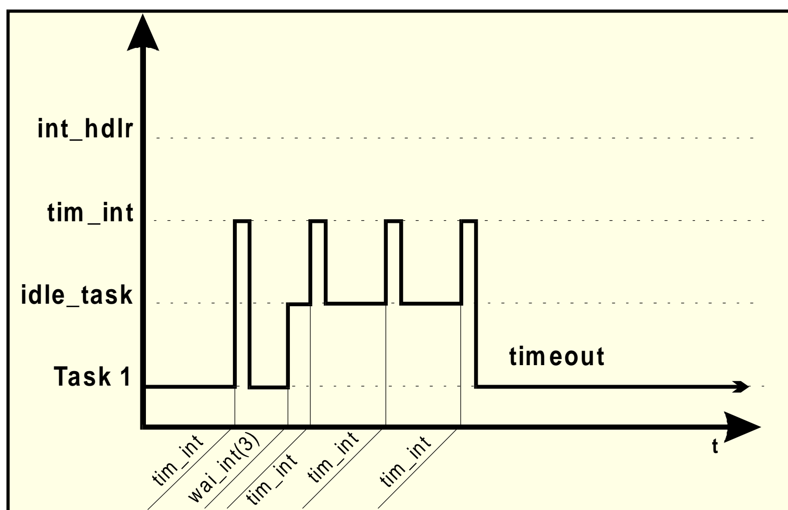


Bild 3: Warten auf einen Interrupt mit Timeout

2.5.2. Ereignisse

Ereignisse müssen sich nicht zwangsläufig durch einen Interrupt bemerkbar machen. Es kann beispielsweise auch sein, daß ein Task mit einer Aufgabe fertig ist und er das einem oder mehreren anderen Tasks mitteilen muß, die darauf warten. Vielleicht soll auch die Nachricht, daß ein Interrupt eingetreten ist, an weitere Tasks verbreitet werden. Es kommt auch oft vor, daß ein Task auf mehrere Ereignisse wartet. All diese Situationen können von UniMOS mit den Ereignis- oder Event-Funktionen behandelt werden.

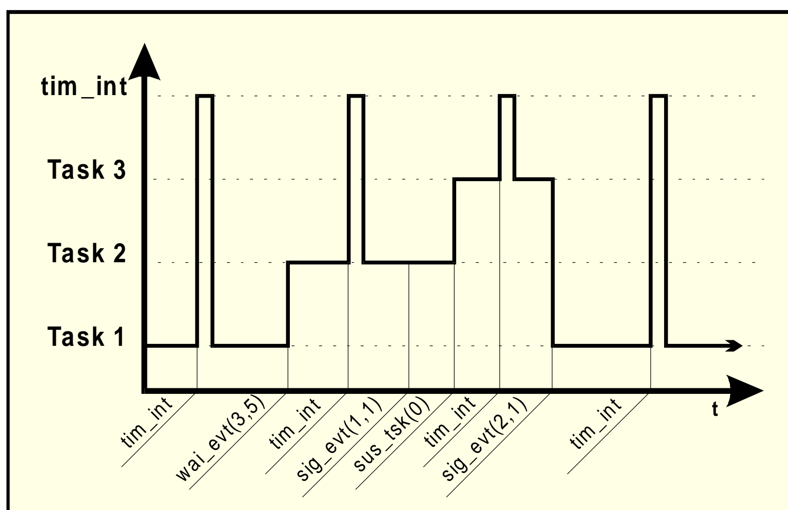


Bild 4: Warten und Signalisieren zweier Ereignisse

Bild 4 zeigt die Tasks 1 bis 3, deren Taskprioritäten mit steigender Tasknummer abnehmen (also z.B. Priorität 5 für Task 1, Priorität 8 für Task 2 und Priorität 9 für Task 3). Zunächst wird Task 1 ausgeführt, der dann aber auf die Ereignisse 1 und 2 warten muß (der Parameter für ***wai_evt*** wird bitweise interpretiert, also: 0x0001 | 0x0002 = 0x0003). Wenn Task 1 wartet, dann wird aufgrund seiner Priorität Task 2 ausgeführt. Er signalisiert das Ereignis 2, jedoch bleibt Task 1 weiterhin wartend, da Ereignis 1 noch nicht eingetreten ist. Task 2 suspendiert sich und damit kann Task 3 weiterlaufen, der schließlich das Ereignis 1 signalisiert. Damit kann Task 1 fortgeführt werden, bevor ein Timeout eintritt.

Für jeden Task gibt es die Möglichkeit, auf bis zu 16 verschiedene Ereignisse zu reagieren. Der Eintritt eines oder mehrerer dieser Ereignisse kann dann von anderen Tasks signalisiert werden. Dazu steht in jedem TCB ein 16-bit Wort zur Verfügung, das bitweise interpretiert wird. Jedes Bit entspricht einem Ereignis. Wenn das Bit gesetzt ist (1), dann bedeutet das, daß das Ereignis noch nicht eingetreten ist, während das Bit nach Eintritt gelöscht (0) wird.

Mit der Funktion ***ini_evt*** wird zunächst das Wort im TCB so initialisiert, daß diejenigen Bits (=Ereignisse) gesetzt werden, auf die der Task warten soll. Alle anderen Bits werden gelöscht. Mit ***pol_evt*** kann jetzt der Task den Status aller oder ausgewählter Ereignisse abfragen. Diese Funktion kehrt sofort mit dem Ergebnis zurück, auch wenn die Ereignisse noch nicht eingetreten sind. Der Task kann also selber entscheiden, was er in der verbleibenden Zeit machen will. Es ist zu beachten, daß in diesem Fall ein Ereignis von einem niedriger priorisierten Task überhaupt nicht gemeldet werden kann, da er den Prozessor nicht erhält. Um auch diesem Task eine Chance zu geben und auch um Prozessorzeit zu sparen, kann man mit der Funktion ***wai_evt*** auf den Eintritt von Ereignissen warten, die dann mit der Funktion ***sig_evt*** an den Task gemeldet werden. ***sig_evt*** prüft jeweils, ob alle Ereignisse eingetreten sind und setzt den Task READY, falls das der Fall ist.

Um endloses Warten beim Ausbleiben von Ereignissen zu vermeiden, kann man auch bei der Funktion ***wai_evt*** einen Timeout angeben. Nach dessen Ablauf wird der Task automatisch mit dem entsprechenden Fehlercode wieder READY gesetzt.

2.5.3. Nachrichten

Wenn mehr Informationen benötigt werden als nur die Nachricht über den Eintritt eines Ereignisses, dann können mit UniMOS auch Nachrichten beliebigen Inhaltes verschickt werden. Das ist eine sehr flexible Einrichtung, da hier die Art und die Bedeutung der Nachricht nicht von vorneherein festgelegt ist. Außerdem können beliebig viele Nachrichten verschickt werden, die vom Empfangs-Task der Reihe nach aus einer Warteschlange abgeholt werden können. Der oder die sendenden Tasks müssen also nicht jeweils abwarten, bis die vorherige Nachricht verarbeitet worden ist. In Anlehnung an das weltweite System der Postverteilung von Briefen und Paketen bezeichnet man die zu übertragenden Informationen mit dem englischen Wort "Mail" und die Warteschlange als eine "Mailbox", also einen Briefkasten.

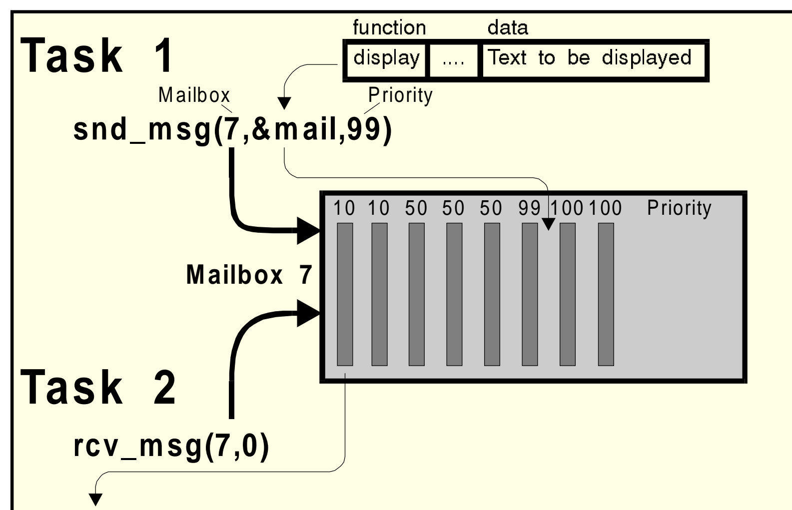


Bild 5: Nachrichtentransport in UniMOS

Wie Bild 5 zeigt, werden Nachrichten nicht an einen Task geschickt, sondern an eine Mailbox. Aus dieser Mailbox kann die Nachricht von jedem Task abgeholt werden, normalerweise wird man aber einen Task als "Eigentümer" der Mailbox haben, der sie als einziger leeren darf. In dem gezeigten Beispiel sendet Task 1 eine Nachricht mit der Priorität 99 an Mailbox 7. Die Nachricht wird hinter anderen Nachrichten gleicher Priorität einsortiert, jedoch vor solchen mit niedrigerer Priorität (niedrige Zahl = hohe Priorität). Beim Empfangen einer Nachricht aus einer Mailbox wird immer die erste Nachricht aus der Warteschlange abgeholt. Das ist immer die älteste derjenigen Nachrichten mit der höchsten Priorität. Hier ist angenommen, daß Task 2 Daten auf

einem Ausgabegerät darstellt. Der Funktionscode "Display" gibt die Gerätenummer an, während im Datenteil der darzustellende Text steht.

Die maximale Anzahl der Mailboxen wird beim Initialisieren des Systems in der Konfigurationstabelle festgelegt. Für jede Mailbox wird ein Speicherblock angelegt, wie das im Anhang näher beschrieben ist. Eine Mailbox wird über ihre Nummer angesprochen. Mailbox-Nummern gehen von null bis zu dem in der Konfigurationstabelle angegebenen Maximalwert minus eins.

Wie bei einem Brief, der per Post verschickt wird, muß auch eine Nachricht in UniMOS in einen "Umschlag" gesteckt werden. Die Struktur dieses Umschlages ist vorgegeben. Eine Beschreibung findet sich im Kapitel "Der Nachrichtenblock" auf Seite 68 im Anhang. Diese Struktur ist notwendig, um die Nachricht in der Mailbox zu verketten und einen Absender einzutragen. Außerdem kann man der Nachricht eine Priorität mitgeben, um sie vor oder nach anderen Nachrichten mit niedriger oder höherer Priorität aus der Mailbox abzuholen. Normalerweise wird man dem empfangenden Task einen Funktionscode übergeben, der ihm mitteilt, was er mit dieser Nachricht zu tun hat. Dies könnte zwar als Teil der Daten übertragen werden, es wurde jedoch ein Byte in der Mail-Struktur vorgesehen, um hier einen gewissen Standard zu schaffen. Die Benutzung dieses Bytes obliegt dem Programmierer. Wenn der empfangende Task sowieso weiß, was er mit der Nachricht machen soll, dann braucht man diesen Funktionscode nicht.

Die eigentliche Nachricht wird im variabel langen Datenbereich der Mail übertragen. Hier kann beispielsweise eine Zeichenkette, eine Struktur oder ein Zeiger übergeben werden. Der Systementwickler legt fest, wie dieser Bereich genutzt wird. Er kann sogar ganz fehlen, wenn der Funktionscode zum Übertragen der Nachricht ausreicht.

Da der "Umschlag", mit dem die Nachricht verschickt wird, Variable Bereiche enthält, die vom System zum Verketten benötigt werden, kann ein solcher Umschlag nur im RAM-Bereich stehen. Sollen trotzdem Nachrichten verschickt werden, die im ROM stehen, so muß man einen Zeiger auf diese Nachrichten verschicken.

Zum Verschicken einer Nachricht an eine Mailbox stellt UniMOS die Funktion ***snd_msg*** zur Verfügung. Wenn bereits ein Task auf die Nachricht wartet, dann wird ein Zeiger an diesen Task übergeben und er wird in den READY-Zustand überführt. Andernfalls wird die Nachricht in der Mailbox abgelegt und zwar so, daß sie hinter alle anderen Nachrichten mit gleicher oder höherer Priorität angehängt wird und vor solche mit niedrigerer Priorität.

Die erste Nachricht in einer so verketteten Liste kann entweder mit der Funktion ***rcv_msg*** oder ***pol_msg*** abgeholt werden, je nachdem, ob gewartet werden soll, falls keine Nachricht vorhanden ist, oder nicht. ***rcv_msg*** wartet gegebenenfalls auf eine Nachricht, wobei man einen Timeout spezifizieren kann, um endloses Warten zu verhindern. Wie auch bei den anderen Funktionen mit Timeout, bedeutet der Parameter 0, daß ohne Timeout gewartet werden soll. ***pol_msg*** prüft hingegen, ob eine Nachricht vorhanden ist und holt sie gegebenenfalls aus der Warteschlange ab. Ist keine Nachricht vohanden, so kehrt die Funktion sofort zurück. Sowohl bei ***pol_msg*** als auch bei ***rcv_msg*** wird im Erfolgsfalle ein Zeiger auf die Nachricht übergeben. Ein NULL-Zeiger bedeutet, daß keine Nachricht erhalten wurde. Der erweiterte Fehlercode im TCB zeigt dann die Ursache an.

2.5.4. Resourcen

Resourcen, auch Semaphore (Ampeln) genannt, werden benötigt, um den Zugriff auf physikalische oder logische Einheiten zu steuern, von denen nur eine begrenzte Anzahl vorhanden ist. Das läßt sich am einfachsten an einem Beispiel erklären (siehe Bild 6). Ein Flugplatz, der von vielen Flugzeugen benutzt werden soll, habe nur eine Start/Landebahn. Diese Bahn kann zwar von allen Flugzeugen zum Starten und Landen benutzt werden, jedoch nur nacheinander und nicht zur gleichen Zeit. Der Pilot, der die Bahn benutzen will, muß das beantragen. Wenn die Bahn momentan frei ist, wird er sie sofort erhalten, anderenfalls muß er sich in die Warteschlange einfügen. Er bekommt die Bahn erst dann, wenn alle, die sich vorher angemeldet haben, mit ihren Aktionen (Starten, Landen aber auch Schnee räumen oder Gras mähen) fertig sind. Man faßt hier die Bahn als sogenannte Resource auf, also etwas von dem nur begrenzt viel (hier nur eine) vorhanden ist.

Bild 6: Resourcen am Beispiel eines Flugplatzes

Die Verwaltung einer Resource muß immer von einer zentralen Stelle vorgenommen werden. Auf einem Flugplatz ist das der "Turm", bei UniMOS ist es der System-Kern. In UniMOS wird die Resource mit einem Aufruf der Funktion ***pol_rsc*** oder ***req_rsc*** beantragt. ***pol_rsc*** belegt die Resource nur dann, wenn sie verfügbar ist. Anderenfalls kehrt die Funktion sofort mit einem entsprechenden Fehlercode zurück. Das Programm kann dann entscheiden, was weiter zu tun ist. Soll das Programm solange warten, bis die Resource verfügbar ist, dann muß die Funktion ***req_rsc*** verwendet werden. Dabei kann eine Priorität angegeben werden, mit der der Task die Resource benötigt. Wie bei den Nachrichten wird auch hier eine Warteschlange mit Prioritäten unterstützt. Ein Task, der die Resource dringend benötigt, kann damit gegenüber anderen Tasks bevorzugt werden. Im Beispiel mit den Flugzeugen kann man mit dieser Strategie demjenigen Flugzeug das landen will und vielleicht nur noch wenig Treibstoff hat, die Bahn mit hoher Priorität zur Verfügung stellen, obwohl vielleicht zehn andere starten wollen und schon länger darauf warten.

Ein Task, der die Resource nicht mehr länger benötigt, kann sie durch Aufruf der Funktion ***rel_rsc*** wieder freigeben.

Die bisher beschriebene Art von Resourcen bezeichnet man auch als binäre Resourcen, da es jeweils nur eine ihrer Art gibt. UniMOS gestattet jedoch auch die Verwaltung mehrerer gleichartiger, ununterscheidbarer Resourcen mit einem Resourcen-Kontrollblock. Dazu wird beim Initialisieren der Resource die maximal verfügbare Anzahl Einheiten angegeben. Jeder Aufruf von ***pol_rsc*** oder ***req_rsc*** nimmt eine dieser Einheiten weg und erst wenn keine mehr verfügbar ist, wartet der Task (***req_rsc***) oder liefert eine Fehlermeldung zurück (***pol_rsc***). Diese Art von Resourcen kommen in der Praxis wesentlich seltener vor als die binären Resourcen. UniMOS unterscheidet auch nicht zwischen beiden Arten. Bei einer binären Resource muß lediglich die maximale Anzahl Einheiten auf eins gesetzt werden. Bevor man eine Resource benutzen kann, muß man sie mit ***ini_rsc*** initialisieren. Dadurch erfährt UniMOS die maximale Anzahl der Einheiten dieser Resource.

Es soll hier nochmals deutlich gesagt werden, daß die Freigabe einer Resource demjenigen Task obliegt, der sie auch belegt hat. Wenn dieser Task beispielweise terminiert wird, ohne daß er die Resource vorher freigeben konnte, so bleibt sie normalerweise für alle Zeiten belegt. Das liegt in erster Linie daran, daß UniMOS keine Information über den Inhaber einer Resource ablegt. Damit kann das System beim Terminieren auch nicht feststellen, ob und welche Ressourcen belegt sind um sie dann automatisch freizugeben. Das hat außerdem zu Folge, daß ein Task eine Resource releasen kann, die er überhaupt nicht hat. Auf nicht gerade elegante Weise kann man so also auch eine Resource wiederbekommen, von der man weiß, daß sie auf die oben beschriebene Art "verloren gegangen" ist.

2.6. Speicherverwaltung

Obwohl es nicht unbedingt zu einem Multitasking-Kern gehört, stellt UniMOS auch Funktionen zur Speicherverwaltung zur Verfügung. Diese Funktionen gestatten es, Speicherplatz anzufordern, der normalerweise nur zeitweise benötigt wird. Beispielsweise kann man sich so einen Speicherblock beschaffen, in den man eine Nachricht ablegt, die zu einem anderen Task geschickt werden kann. Der empfangende Task verarbeitet die Nachricht und gibt den Speicherblock wieder frei, wenn er nicht mehr länger benötigt wird.

In UniMOS wurde eine relativ einfache Art der Speicherverwaltung realisiert. Es werden zunächst sogenannte Speicherpools definiert, von denen jeder eine bestimmte Anzahl Speicherblöcke fester Länge beinhaltet. Die Verwaltung wird durch die feste Blocklänge sehr stark vereinfacht. Es können so keine nicht mehr benutzbaren Lücken im Speicherbereich entstehen, wie es bei variabel langen Blöcken der Fall ist. Wenn Speicherblöcke unterschiedlicher Länge benötigt werden, so kann man weitere Pools mit anderen Blocklängen festlegen. Die maximale Anzahl Speicherpools wird beim Initialisieren des Systems in der Konfigurationstabelle angegeben. ***ini_tsk*** legt dann für jeden Pool einen sogenannten "Pool Control Block", den PCB, im unimos-Segment an (siehe "Speicherbelegung und Speicherbedarf" ab Seite 62).

Alle PCBs müssen in ein Segment passen, dürfen also insgesamt nicht länger als 64 kB sein. Damit sind sicher für alle Anwendungen genügend Pools möglich. Normalerweise wird man nicht mehr als drei Pools benötigen.

Die maximale Anzahl Speicherblöcke pro Pool ist auf 65535 begrenzt. Die Länge eines Speicherblockes muß mindestens vier Bytes betragen, damit der Zeiger zur Verkettung der Blöcke eingetragen werden kann. Die maximale Blocklänge beträgt 64 kB. Dazu muß beim Initialisieren die Blocklänge null spezifiziert werden. Andere Beschränkungen der Blocklänge gibt es nicht. Es sind also auch Blocklängen von beispielsweise 7 oder 100 Bytes möglich.

Der erste Speicherblock beginnt unmittelbar an der beim Initialisieren angegebenen Adresse. Es erfolgt also keine Ausrichtung. Alle Speicherblöcke eines Pools werden unmittelbar hintereinander abgelegt. Sollen also die einzelnen Speicherblöcke ausgerichtet sein, so muß zunächst eine ausgerichtete Anfangsadresse angegeben werden und die Blocklänge muß ein ganzzahliges vielfaches der Größe sein, nach der ausgerichtet wird.

Bevor ein Speicherpool benutzt werden kann, muß er mit ***ini_mem*** initialisiert werden. Dadurch erfährt UniMOS die Anfangsadresse des Speicherpools, sowie die Länge eines Speicherblocks und die Anzahl der Speicherblöcke in diesem Pool. UniMOS verkettet diese einzelnen Blöcke mit Zeigern, so daß jeder Block auf einen nachfolgenden (noch freien) Block zeigt. Ein Zeiger auf den ersten freien Block wird im PCB gespeichert, der einfach und schnell mit ***get_mem*** abgeholt werden kann. Der so zur Verfügung gestellte Speicherblock steht dem Aufrufer in voller Länge zur Verfügung. Es ist also nicht notwendig, den Platz freizuhalten, in dem der Zeiger auf den nächsten Block war, da er nur zur Verkettung der unbenutzten Speicherblöcke benötigt wird.

Ein mit ***rel_mem*** an das System zurückgegebener Speicherblock wird vor dem momentan ersten Block wieder eingekettet. Er steht damit für weitere ***get_mem*** Aufrufe wieder zur Verfügung, jedoch darf er nachdem er zurückgegeben wurde nicht mehr manipuliert werden. Insbesondere dürfen die Zeiger nicht verändert werden.

Im Gegensatz zu ***get_mem*** muß bei ***rel_mem*** keine Pool-Nummer angegeben werden. Da sich Speicherpools nicht überlappen können, kann UniMOS, anhand des Zeigers auf den freizugebenden Block, den Pool selbst bestimmen. Das bedeutet zwar einen etwas erhöhten Aufwand, jedoch muß man so nicht die Poolnummer beim Versenden des Blocks als Nachricht mit angeben, damit der empfangende Task den Block freigeben kann.

Die Länge der Zeiger ist bei den Memory-Management Funktionen vom verwendeten Speichermodell abhängig, falls nichts anderes angegeben wurde. Wird also das Small- oder Medium-Speichermodell verwendet, dann sind die erwarteten Zeiger zwei Bytes lang. Die Speicherpools müssen dann im near-Datenbereich liegen, auf den zur Laufzeit des Programmes das DS-Segmentregister zeigt. Die Gesamtgröße dieses Datenbereiches ist auf 64 kB begrenzt.

Im Compact- und Large-Speichermodell sind die Datenzeiger vier Bytes lang, sind also far-Zeiger aus Segment und Offset. Damit kann ein Pool dann auch größer als 64 kB sein und er muß nicht mehr zwangsläufig im near-Datenbereich stehen.

Da es nun auch häufig den Fall gibt, daß man zwar im Small- oder Medium-Speichermodell arbeitet, aber trotzdem einen Memorypool verwalten möchte, der nicht im near-Datenbereich liegt, kann man die Memory-Management Funktionen alternativ auch mit far-Zeigern benutzen. Auch der andere Fall, daß man einen Pool im near-Segment hat, jedoch im Speichermodell Compact oder Large arbeitet, ist nicht ungewöhnlich. Deshalb

werden in diesen Speichermodellen auch near-Zeiger verwendet, wenn man alternative Funktionen benutzt. Die Namen der alternativ zu verwendenden Funktionen enden nicht mit ***_mem***, sondern mit ***_nmem***, falls near-Zeiger verwendet werden und mit ***_fmem***, falls es far-Zeiger sind. Die folgende Tabelle gibt einen Überblick.

Speichermodell	ini_mem get_mem rel_mem	ini_nmem get_nmem rel_nmem	ini_fmem get_fmem rel_fmem
Small	near	near	far
Medium	near	near	far
Compact	far	near	far
Large	far	near	far

Je nach verwendetem Speichermodell werden also beim Aufruf der ***_mem***-Funktionen die ***_nmem*** oder ***_fmem*** Funktionen verwendet.

Die Far-Zeiger der ***_fmem***-Funktionen sind immer normalisiert, d.h. der Offset liegt zwischen 0 und 15. Ist die Anfangsadresse des Pools auf Paragraphengrenze ausgerichtet, so kann man also immer davon ausgehen, daß es keinen Überlauf des Offset gibt. Auch bei der Verkettung der Speicherblöcke eines near-Pools werden far-Zeiger verwendet. Deshalb muß ein Speicherblock unabhängig vom Speichermodell immer mindestens vier Bytes lang sein. Die Zeiger eines near-Pools sind jedoch nicht normalisiert, da sowieso alle Daten in einem Segment liegen und deshalb ein Überlauf nicht vorkommen darf. Damit man einen Speicherblock, dessen Pool man nicht kennt, auf jeden Fall freigeben kann, darf man mit der Funktion ***rel_fmem*** auch einen Block eines near-Pools zurückgeben. Der Zeiger muß allerdings ein far-Zeiger sein, was man in "C" durch einen Typecast erreichen kann. In Assembler muß zusätzlich das Datensegment angegeben werden.

2.7. UniMOS in DOS Umgebung

Da UniMOS letztendlich ein Programm wie jedes andere ist, kann es auch unter DOS betrieben werden. UniMOS greift nur auf die vorgesehenen Datenbereiche zu und benutzt selbst keinerlei DOS- oder BIOS-Systemfunktionen. Dadurch besteht eine weitestgehende Entkopplung beider Systeme. Es muß sichergestellt werden, daß kein DOS-Programm die internen Kontrollblöcke von UniMOS in unzulässiger Weise verändert. Auf der Demodiskette sind einige Beispielprogramme auch im Quelltext vorhanden, die unter DOS laufen.

Nun zu der wesentlichen Einschränkung, die sich durch DOS und BIOS ergibt:

Weder DOS noch das System-BIOS sind reentrant!

Diese Einschränkung hat weitreichende Folgen. Sie bedeutet, daß DOS/BIOS-Funktionen nicht benutzt werden können, bevor der vorherige Aufruf beendet ist. In einem Multitasking-System kann ein Task jedoch jederzeit unterbrochen und ein anderer Task weitergeführt werden. Die Unterbrechung kann beispielsweise auch mitten in einem DOS-Funktionsaufruf passieren. Wenn der erste Task gerade begonnen hat eine Datei zu lesen und der zweite genau dasselbe mit einer anderen Datei versucht, dann werden mit hoher Wahrscheinlichkeit interne DOS-Puffer überschrieben. Katastrophal können die Auswirkungen werden, wenn Hardware-Bausteine nur teilweise initialisiert werden.

Es gibt verschiedene Möglichkeiten um dennoch unter DOS Multitasking-Fähigkeiten zu haben. Die einfachste Lösung ist, daß man vor dem Aufruf von DOS- oder BIOS- Funktionen, die Taskumschaltung blockiert (Funktion dis_pre auf Seite 24) und sie anschließend wieder freigibt. Dieses Verfahren hat jedoch erhebliche Auswirkungen auf die Systemleistung. Während zum Beispiel eine Datei von einer Diskette gelesen wird, kann kein anderer Task ausgeführt werden. Falls außerdem die Diskette nicht eingelegt ist, "hängt" möglicherweise das gesamte Programm. Normalerweise verbietet sich dieses Vorgehen also von selbst. Hat man jedoch nur solche

Systemfunktionen, von denen man sicher weiß, daß sie in einer vorgegebenen (kurzen) Zeit beendet sind (z.B. Bildschirmausgabe oder Speicher allokieren), dann ist diese Strategie durchaus anwendbar.

Eine weitere einfache Lösung ist es, alle DOS- und BIOS-Funktionen in jeweils einen einzigen Task zu verlagern. Das ist oft möglich, da diese Funktionen sowieso zu einem Aufgabenbereich gehören, der praktischerweise von einem Task erledigt wird. Ein Datenbanksystem könnte aus drei Tasks bestehen: Task 1 liest und schreibt die Dateien, Task 2 übernimmt die Bediener-Schnittstelle (Tastatur und Bildschirm) und Task 3 erledigt Sortierfunktionen im Hintergrund. Dabei benutzt Task 1 DOS Funktionen (INT 21h), Task 2 BIOS Funktionen (INT 10h und INT 16h) und Task 3 greift nur auf interne Datenbereiche zu. Zu beachten ist, daß auch DOS letztendlich BIOS-Funktionen verwendet. Man darf also nicht glauben, daß ein Task per DOS eine Datei lesen kann, während ein anderer über den BIOS Interrupt 13h direkt auf den Datenträger zugreift. Auch BIOS kann teilweise intern wieder andere BIOS-Funktionen aufrufen. Das oben genannte Beispiel ist jedoch unkritisch.

Empfehlenswert ist auch die Einrichtung eines *Server-Tasks*, der für alle DOS- und BIOS-Funktionen zuständig ist. Jeder andere Task, der eine solche Funktion benötigt, muß sich mit einer entsprechenden Anfrage an den Server-Task wenden. Das geschieht am sinnvollsten über Mailboxen. Es wird eine Mailbox eingerichtet und der Server wartet auf Nachrichten dafür. Wenn eine Nachricht ankommt, dann wird sie interpretiert und ausgeführt. Weitere Nachrichten werden in der Mailbox abgelegt und vom Server erst dann bearbeitet, wenn die vorherige abgeschlossen ist. Reentrancy-Probleme werden damit vermieden, da immer der vorherige Aufruf beendet ist, bevor der nächste begonnen wird.

Man kann DOS und BIOS auch als Resourcen auffassen, was wohl das eleganteste Verfahren zur Sicherstellung der Integrität ist (leider aber auch ein unbefriedigendes). Dazu werden die entsprechenden Interrupts (INT 21h für DOS, verschiedene für BIOS) abgefangen und mit einem "request resource" Funktionsaufruf geschützt. Damit ist sichergestellt, daß immer nur eine Funktion den entsprechenden Aufruf benutzt. Hier gibt es jedoch leider mehrere Hindernisse. Der Interrupt 21h wird auch oft von DOS selbst verwendet, so daß ein DOS Aufruf weitere DOS Aufrufe zur Folge haben kann. Der zweite DOS-Aufruf wird abgefangen und damit wartet das System dann für immer auf die Freigabe der Resource. Eine Lösung dieses Dilemmas ist, daß der Interrupt-Handler sich die Nummer desjenigen Task merkt, der die Resource "DOS" momentan hat. Wenn derselbe Task DOS nochmals verlangt, dann wird req_rsc nicht nochmals aufgerufen. Ein weiteres Problem sind diejenigen DOS-Funktionen, die erst nach relativ langer Zeit (drucken) oder auch nach unbestimmter Zeit (Warten auf Tastatureingaben) beendet werden. Man muß diese dann separat als eigene Resource abfangen. Dabei muß man dann hoffen, daß andere DOS-Aufrufe gefahrlos möglich sind, während ein Task in DOS auf eine Tastatureingabe wartet. Daß das dann auch noch in zukünftigen DOS-Versionen der Fall sein wird garantiert niemand.

Um das zuletzt beschriebene Verfahren anwenden zu können, muß man recht genau wissen, welche Funktionen von den Tasks verwendet werden. Wenn der DOS-Funktionsaufruf (INT 21h) durch direkte BIOS-Aufrufe umgangen wird, dann nützt in diesem Fall die Semaphore nichts.

Die vorigen Abschnitte haben gezeigt, mit welchen Problemen derjenige rechnen muß, der unter DOS ein Multitaskingsystem betreiben will. Welche Schwierigkeiten ergeben sich aber in einem System ohne DOS?

Auch hier hat man Probleme mit der Reentrance-Fähigkeit von Programmen. Prinzipiell sind solche Programmteile nicht reentrant, die auf globale Daten zugreifen, die von mehreren Tasks benutzt werden. Wenn also globale Daten nur von einem Task benutzt werden oder wenn lokale Daten (Stackvariable) verwendet werden, so kann das als sicher gelten. Eigene Programme kann man auf diese Eigenschaften hin auslegen. Was aber ist mit den Systemfunktionen? UniMOS-Funktionen sind selbstverständlich aus allen Tasks ohne besondere Vorkehrungen aufrufbar. Da diese Funktionen jedoch in hohem Maße globale Daten verändern müssen (nämlich die Kontrollblöcke), wurden Vorkehrungen getroffen, um die Reentrancy zu gewährleisten. Immer dann, wenn eine Funktion nicht durch eine andere unterbrochen werden darf, werden Interrupts gesperrt. Damit ist die Sicherheit der Kontrollblöcke gewährleistet. Diese Interrupt-Sperrzeiten sind so kurz wie möglich gehalten worden, ganz vermeidbar sind sie aber nicht.

Problematisch sind die Funktionen der System-Bibliothek des Compilers. Einige dieser Funktionen sind nur in einer DOS-Umgebung anwendbar und einige sind leider nicht reentrant. Normalerweise ist recht gut dokumentiert, welche Funktionen nicht ohne DOS laufen, über die Reentrancy läßt sich aber kein Handbuch aus. Das bedeutet leider das man entweder die Funktion ausprobieren oder den Quellcode dazu kaufen muß. Der ist für die z.Zt. marktführenden Compiler verfügbar.

Es gibt jedoch auch verschiedene Wege, die Systembibliotheken dennoch in mehreren Tasks zu verwenden. Abgesehen von der (nicht zu empfehlenden) Änderung des Quellcodes laufen alle diese Lösungen auf eine

Separierung der globalen Datensegmente hinaus. Eine gern genutzte Möglichkeit sind die speicherresidenten Funktionen unter DOS (***T***erminate and ***S***tay ***R***esident = ***TSR***). Man benutzt hier voneinander vollkommen unabhängige Programme, die fest in den Systemspeicher geladen werden. Über vorher vereinbarte Schnittstellen (z.B. Interrupts) kommunizieren sie mit dem zuerst installierten Echtzeitkern. Dieses Verfahren ist zwar einfach und zuverlässig, hat jedoch einige Nachteile. Zunächst läßt sich ein TSR-Programm nicht mehr ohne weiteres aus dem Speicher entfernen, wenn es nicht mehr benötigt wird. Man muß normalerweise den Rechner neu starten. Außerdem wird jede Funktion für jeden Task komplett neu geladen und belegt damit Speicherplatz für Programm und Daten. Wenn die Tasks untereinander kommunizieren wollen oder Systemfunktionen aufrufen wollen, so müssen immer FAR-Pointer verwendet werden, da alle Segmente voneinander getrennt sind. Der Benutzer muß abwägen, ob die genannten Einschränkungen tragbar sind.

3. Funktionsbeschreibungen

Dieses Kapitel beschreibt zunächst die Aufrufkonventionen der UniMOS-Funktionen und anschließend die Funktionen selbst, deren Aufrufparameter und Rückgabewerte.

UniMOS-Funktionen werden als normale (externe) Unterprogramme aufgerufen. Der Linker fügt die Funktionen zum Anwenderprogramm hinzu. Abhängig von der Funktion und vom verwendeten Speichermodell werden NEAR- oder FAR- Unterprogrammaufrufe verwendet. Diejenigen Funktionen, die einen Taskwechsel verursachen können, werden immer mit einem FAR-Call aufgerufen. Das ist jeweils in der Funktionsbeschreibung dokumentiert.

Die Länge von Datenzeigern ist in der Regel vom Speichermodell abhängig. Wenn NEAR-Zeiger verwendet werden, so geht UniMOS immer davon aus, daß das Datensegmentregister DS auf den Datenbereich zeigt. Bei Verwendung der Programmiersprache "C" ist das automatisch der Fall. Während der Initialisierung wird DS mit der Adresse der ***DGROUP*** geladen. Diese Gruppe von Segmenten beinhaltet Daten, Konstanten und den Stack. Wenn diese Initialisierungsroutine nicht benutzt wird (z.B. in einem Controller ohne DOS oder beim Programmieren in Assembler), dann muß der Benutzer dafür sorgen, daß das DS-Register geladen wird. Zur detaillierten Beschreibung wird auf das Benutzerhandbuch des Assemblers oder des Compilers verwiesen.

Alle Funktionen werden mit der ***CALL***-Instruktion aufgerufen und sind somit direkt aus einem C-Programm heraus verwendbar. Da UniMOS-Funktionen so geschrieben sind, daß sie mit C-Programmen zusammengebunden werden können, übergeben sie den Rückgabewert im AX-Register oder, bei 32-bit Werten, im DX,AX Registerpaar. Auch ein Fehlercode wird normalerweise in AX zurückgegeben. Das führt zu Konflikten bei solchen Funktionen, die einen Zeiger zurückgeben. Diese Funktionen liefern im Fehlerfall einen NULL-Zeiger zurück (AX=0 bzw. AX=0 und DX=0). Der eigentliche Fehlercode, der die Ursache des Fehlers beschreibt, wird als erweiterter Fehlercode im Task Kontroll Block (TCB) abgespeichert (siehe "Der Task Control Block (TCB)" auf Seite 66).

Wichtiger Hinweis: Wenn nicht anders angegeben, dann sollten UniMOS Funktionen nur während der normalen Ausführung eines Tasks aufgerufen werden, nicht jedoch in einem Interrupt-handler. Der Grund liegt darin, daß eine Interrupt-Routine normalerweise sehr schnell ausgeführt werden muß, da weitere Ereignisse auftreten können, die wieder Interrupts generieren. Die meisten UniMOS-Funktionen können eine Taskumschaltung hervorrufen und damit ist ihre Rückkehr vollkommen unbestimmt.

UniMOS-Funktionen halten sich an die Konventionen der Borland- und Microsoft C-Compiler, daß nur die Registerwerte in **SI**, **DI**, **DS**, **SS** und **BP** erhalten werden müssen. Alle anderen Registerinhalte können überschrieben werden.

Eigene Datentypen: In den folgenden Funktionsbeschreibungen und auch in den Beispielen werden einige selbstdefinierte Datentypen verwendet, deren Benutzung sich als sehr bequem herausgestellt hat. Sie sind in der Include-Datei "MYTYPE.H" definiert, die auch bei nicht-UniMOS-Projekten nützlich sein kann. Hier sind die wichtigsten Definitionen aus dieser Datei:

```
typedef unsigned char   BYTE;
typedef unsigned int    WORD;
#define REGISTER        register unsigned int
#define FALSE 0
#define TRUE 1
```

*INTHDLR far *def_int(intnr, addr)*

unsigned int intnr;	Interrupt Nummer (0 .. 255)
void (far *addr) (void);	Addresse des Interrupt handlers

Aufruf: near oder far call

■ Beschreibung

Die Funktion ***def_int*** definiert einen Interrupt handler für einen bestimmten Interrupt. Die Adresse des Interrupt handlers *addr* wird an der zum Interrupt *intnr* gehörenden Stelle in der Interrupt Vektor Tabelle eingetragen.

■ Return Wert

def_int liefert den vorherigen Interruptvektor zurück.

■ Siehe auch

get_int (Seite 28)

■ Beispiel

Assembler:

```
        include misc.mac        ;Miscellaneous macros
        include unimos.inc      ;UniMOS includes
        include unimosif.inc    ;UniMOS interface
        .....
        get_int tim_int         ;get previous vector
        .....
        def_int tim_int,tint_hdlr ;define vector
        .....
```

C:

```
#include "unimos.h"
typedef void (interrupt INTHDLR) (void) ;
INTHDLR *oldtimint ;
oldtimint = get_int(TIMINT) ;
        /* get address of old interrupt handler */
def_int(TIMINT,newtimint) ;
        /* install new interrupt handler */
.....
void interrupt newtimint(void)
{
    (oldtimint)() ;
     .....
}
```

Hinweis: Im C-Programmbeispiel wurden das Schlüsselwort "interrupt" des Borland C-Compilers verwendet, das andere Compiler möglicherweise nicht unterstützen. Die Interrupt-Funktion selbst ruft wieder die vorherige Interrupt-Funktion auf (oldtimint). Wenn sichergestellt werden kann, daß Interrupts erst auftreten können, nachdem *oldtimint* den korrekten Wert erhalten hat, kann man alternativ zu oben auch folgenden Funktionsaufruf wählen:

oldtimint = def_int(TIMINT,newtimint) ;

Vorsicht: hier wird *oldtimint* erst gesichert, nachdem bereits der neue Vektor abgespeichert wurde. Wenn genau dazwischen ein Interrupt auftritt, dann führt der Interrupt-handler den Code aus, zu dem *oldtimint* zeigt.

unsigned int dis_pre(void)

keine Parameter

Aufruf: near oder far call

■ Beschreibung

dis_pre inkrementiert den Preemption-Zähler. Dadurch wird die Taskumschaltung verhindert (disable preemption). Anschließend wird zum aufrufenden Programm zurückgekehrt.

dis_pre gestattet quasi das Anhalten des Multi-Tasking Systems. Der momentane Task behält den Prozessor solange, bis er wieder ***ena_pre*** aufruft und damit die normale Task-Umschaltung wieder freigibt. Diese Funktion sollte nur mit großer Vorsicht verwendet werden, da man damit auch die Ausführung höherpriorisierter Tasks verhindert. Diejenigen Funktionen, die den momentanen Task-Status verändern wollen (***sus_tsk***, ***exi_tsk***, ***wai_int, rcv_msg***), werden mit einem Fehlercode beendet.

dis_pre wird dann benötigt, wenn ein Task mit niedriger Priorität kritische Programmteile ausführen muß, wobei nicht zwischendurch ein anderer Task ausgeführt werden darf. Interrupts bleiben jedoch freigegeben, so daß z.B. eine serielle Schnittstelle bedient werden kann. Der Interrupt-Handler für diese serielle Schnittstelle darf auch ***sig_int*** aufrufen und damit einen wartenden Task wieder ***READY*** setzen. Ein Taskwechsel findet jedoch erst beim Aufruf von ***ena_pre*** statt. Zu weiteren Erläuterungen siehe auch ***ena_pre*** auf Seite 26.

■ Return Wert

dis_pre gibt den Wert des Preemption-Zählers nach dem Inkrementieren zurück.

■ Siehe auch

ena_pre (Seite 26)

■ Beispiel

Assembler:

```
        include misc.mac        ;Miscellaneous macros
        include unimos.inc      ;UniMOS includes
        include unimosif.inc    ;UniMOS interface
        .....
        ini_tsk config_table    ;initialize UniMOS
        dis_pre                 ;prevent tasks from being started
        sta_tsk task1,DGROUP,stack0,0,0,0
        sta_tsk task2,DGROUP,stack1,1,1,0
        sta_tsk task3,DGROUP,stack2,2,1,0
        ena_pre                 ;start task with highest priority
```

C:

```
#include "unimos.h"
#define STACKSIZE 128      /* number of words for the stack */
extern WORD _seg *dgroup ; /* Segment for DGROUP */
        .....
        ini_tsk (&ct) ;
        dis_pre () ;
        sta_tsk (task1,dgroup,&stack1[STACKSIZE],1,1,5) ;
        sta_tsk (task2,dgroup,&stack2[STACKSIZE],2,1,5) ;
        sta_tsk (task3,dgroup,&stack3[STACKSIZE],3,1,5) ;
        ena_pre () ;
```

Beide Beispiele zeigen die gleiche Startsequenz für UniMOS. Nach dem Aufruf von ini_tsk wird dieser Code zum Hintergrundtask mit der niedrigsten Priorität. Sobald dann ein weiterer Task mit höherer Priorität gestartet wird, bekommt dieser den Prozessor. Um die Initialisierung ungestört ablaufen zu lassen, wird hier die Taskumschaltung verhindert.

unsigned int far ena_pre(void)

keine Parameter

Aufruf: far call

■ Beschreibung

ena_pre dekrementiert den Preemption-Zähler. Wenn dieser dadurch zu null geworden ist, dann wird die Taskumschaltung freigegeben (enable preemption). Anschließend wird der Dispatcher aufgerufen, so daß eine Taskumschaltung stattfindet, falls das nötig ist. Wurde der Zähler nicht null, so bleibt die Taskumschaltung verhindert. Damit lassen sich Aufrufe von ***ena_pre*** und ***dis_pre*** schachteln. Es muß also immer zu einem ***dis_pre*** genau ein ***ena_pre*** folgen. Als Zähler wird ein Wort verwendet, so daß für die Praxis genügend Schachtelungen möglich sind. Wenn der Zähler bereits null ist, so wird er nicht mehr weiter dekrementiert, sondern es wird sofort zum aufrufenden Programm zurückgekehrt. Das wurde deshalb eingebaut, damit man durch mehrfachen Aufruf von ***ena_pre*** die Taskumschaltung sicher wieder einschalten kann. Zu weiteren Erläuterungen siehe auch ***dis_pre*** auf Seite 24.

■ Return Wert

ena_pre gibt den Wert des Preemption-Zählers nach dem Dekrementieren zurück.

■ Siehe auch

dis_pre (Seite 24)

■ Beispiel

siehe Beispiele zu ***dis_pre*** auf Seite 24

unsigned int far exi_tsk(void)

keine Parameter

Aufruf: far call

■ Beschreibung

exi_tsk führt den Task vom ***RUNNING*** in den ***DORMANT-***Zustand über. Der Task wird also nicht mehr benötigt und es kann anschließend unter dieser Tasknummer mit ***sta_tsk*** auch ein ganz anderer Task neu gestartet werden. Im Gegensatz zum ***SUSPENDED*** Zustand kann der Task nicht mehr fortgeführt werden, sondern er kann nur wieder ganz neu gestartet werden. Der für den Task reservierte Stack-Bereich ist frei und kann anderweitig verwendet werden.

Wichtiger Hinweis: Wenn der Task Resourcen oder Speicher belegt, dann werden diese nicht automatisch freigegeben. Sie sind dann bis zum nächsten Initialisieren des Systems verloren. Das liegt daran, daß das System nicht weiß, wem eine Resource oder ein Speicherblock gehört.

■ Return Wert

exi_tsk gibt einen Fehlercode zurück, wenn momentan die Taskumschaltung durch ***dis_pre*** (siehe Seite 24) verhindert ist. Ist dies nicht der Fall, so findet kein Return statt.

■ Siehe auch

trm_tsk (Seite 56)

■ Beispiel

Assembler:

```
        include misc.mac        ;Miscellaneous macros
        include unimos.inc      ;UniMOS includes
        include unimosif.inc    ;UniMOS interface
        .....
        exi_tsk                 ;stop this task
        jmp fatal_error         ;should never arrive here!
```

C:

```
#include "unimos.h"
        .....
        exi_tsk () ;        /* stop this task */
        fatal_error () ;    /* call fatal error hdlr (just in case) */
```

Die Funktion ***exi_tsk*** wird ohne Parameter aufgerufen. Es sollte der Aufruf einer Fehlerbehandlungs-Funktion folgen, da ***exi_tsk*** im Fehlerfalle zurückkehrt! Soll ein fremder Task gestoppt werden, so muß die Funktion ***trm_tsk*** mit der Nummer des zu stoppenden Tasks als Parameter benutzt werden.

*INTHDLR far *get_int(intnr)*

unsigned int intnr; Nummer des Interrupt Vektors (0 .. 255)

Aufruf: near oder far call

■ Beschreibung

get_int liefert den zum Interrupt *intnr* gehörenden Interrupt-Vektor an die aufrufende Funktion zurück. *intnr* ist einer der Interrupts von 0 bis 255.

■ Return Wert

get_int liefert einen Zeiger zur momentanen Interrupt-Behandlungsfunktion zurück.

■ Siehe auch

def_int (Seite 22)

■ Beispiel

Siehe Beispiel zu ***def_int*** auf Seite 22.

*unsigned char *get_mem(pool)*
*unsigned char near *get_nmem(near_pool)*
*unsigned char far *get_fmem(far_pool)*

unsigned int pool;	Nummer des Memory-Pools
unsigned int near_pool;	Nummer des near Memory-Pools
unsigned int far_pool;	Nummer des far Memory-Pools

Aufruf: near oder far call

■ Beschreibung

Nach dem Initialisieren des Systems können ein oder mehrere Speicherbereiche, sogenannte Pools, definiert werden, die von den Speicher-Management Funktionen verwaltet werden. Jeder Pool beinhaltet eine vordefinierte Anzahl gleichlanger Speicherblöcke, die von den Tasks auf Bedarf angefordert (***get_mem***) und wieder freigegeben (***rel_mem***) werden können.

Im Small und Medium Speichermodell ruft ***get_mem*** die Funktion ***get_nmem*** auf, während bei Compact und Large ***get_fmem*** verwendet wird. Weitere Informationen zu den Speicher-Management Funktionen finden Sie im Kapitel "Speicherverwaltung" ab Seite 17.

■ Return Wert

get_mem liefert im Erfolgsfalle einen *near* oder *far* Zeiger auf den Speicherbereich zurück. Ein NULL-Zeiger bedeutet, daß entweder kein weiterer Block mehr verfügbar ist oder das eine ungültige Pool-Nummer angegeben wurde. Die genaue Fehlerursache wird durch einen erweiterten Fehlercode im TCB angegeben.

■ Siehe auch

ini_mem (Seite 32), ***rel_mem*** (Seite 41)

■ Beispiel

Assembler:

```
        include misc.mac            ;Miscellaneous macros
        include unimos.inc          ;UniMOS includes
        include unimosif.inc        ;UniMOS interface
        .....
        ini_tsk config_table        ;initialize UniMOS
        ini_mem 0,pool0,40,100      ;initialize pool 0
        get_mem 0                   ;get one block from pool 0
        mov mem_addr,ax             ;save address
        .....
        rel_mem mem_addr            ;release the memory block
```

C:

```
#include "unimos.h"
        .....
        ini_tsk (&ct) ;
        ini_mem (0,&pool0,40,100) ;
        mem_addr = get_mem (0) ;
        .....
        rel_mem (mem_addr) ;
```

void ini_evt(list)

unsigned int list; Bit-Liste der Ereignisse

Aufruf: near oder far call

■ Beschreibung

Mit der Funktion ***ini_evt*** wird die Liste derjenigen Ereignisse initialisiert, auf die dieser Task mit einem nachfolgenden ***pol_evt*** oder ***wai_evt*** warten soll. Der Parameter *list* ist ein Wort, das bitweise interpretiert wird. Damit können bis zu 16 unterschiedliche Ereignisse spezifiziert werden. Wenn z.B. Bit 0 gesetzt ist, so ist damit das Ereignis null gemeint. Sind die Bits 2,3 und 7 gesetzt, so sind die Ereignisse zwei, drei und sieben ausgewählt. Die Bedeutung eines Ereignisses ist vollkommen dem Benutzer überlassen.

ini_evt kopiert zum Initialisieren lediglich den Wert *list* in den TCB.

■ Return Wert

ini_evt gibt einen ungültigen Wert zurück.

■ Siehe auch

sig_evt (Seite 46), ***wai_evt*** (Seite 57), ***pol_evt*** (Seite 36)

■ Beispiel

Assembler:

```
        include misc.mac        ;Miscellaneous macros
        include unimos.inc      ;UniMOS includes
        include unimosif.inc    ;UniMOS interface
ev0     equ 0000000000000001b ;event 0
ev1     equ 0000000000000010b ;event 1
ev7     equ 0000000010000000b ;event 7
        .....
Task 1:
        .....
        ini_evt ev0+ev1+ev7  ;initialize the events
wai:    pol_evt ev0          ;active wait for event 0
        or ax,ax             ;zero returned ?
        bne wai              ;loop
        wai_evt ev7,0        ;wait for event 7. Others don´t care
        .....

Task 2:
        sig_evt ev0,1        ;signal event 0 to task 1
        sus_tsk 100          ;wait a while
        sig_evt ev7,1        ;signal event 7 to task 1
```

C:

```
#include "unimos.h"
#define ev0 0x0001              /* event 0 */
#define ev1 0x0002              /* event 1 */
#define ev7 0x0080              /* event 7 */

Task 1:
        .....
        ini_evt (ev0|ev1|ev7)       /* init events */
        while (pol_evt (ev0) != 0) ; /* active wait for event 0 */
        wai_evt (ev7,0)             /* wait for event 7 */
        .....

Task 2:
        sig_evt (ev0,1)             /* signal event 0 to task 1 */
        sus_tsk (100)               /* wait a while */
        sig_evt (ev7,1)             /* signal event 7 to task 1 */
```

*unsigned char *ini_mem(pool,addr,size,count)*
*unsigned char near *ini_nmem(near_pool,near_addr,size,count)*
*unsigned char far *ini_fmem(far_pool,far_addr,size,count)*

unsigned int pool;	Nummer des Memory-Pools
unsigned int near_pool;	Nummer des near Memory-Pools
unsigned int far_pool;	Nummer des far Memory-Pools
unsigned char *addr;	Zeiger zum Anfang des Bereiches
unsigned char near *addr;	near-Zeiger zum Anfang des Bereiches
unsigned char far *addr;	far-Zeiger zum Anfang des Bereiches
unsigned int size;	Größe eines Blockes in Byte (4 .. 65535, 0)
unsigned int count;	Anzahl der Blöcke

Aufruf: near oder far call

■ Beschreibung

Die Funktion ***ini_mem*** kann nach dem Initialisieren des Systems aufgerufen werden um einen oder mehrere Speicherbereiche, sogenannte Pools, zu definieren, die von den Speicher-Management Funktionen verwaltet werden. Jeder Pool beinhaltet Speicherblöcke, deren Anzahl mit dem Parameter *count* und deren jeweilige Länge mit dem Parameter *size* angegeben werden. Die Länge eines Speicherblockes kann zwischen 4 und 65536 (=64 kB) betragen. Soll die Länge 64 kB sein, dann muß der Parameter 0 angegeben werden. Die maximale Länge eines near-Pools kann 64 kB betragen, während ein far-Pool bis zu 1 MB groß sein kann.

Die Anfangsadresse desjenigen RAM-Bereiches, den UniMOS benutzen darf, wird mit dem Parameter *addr* angegeben. Die Nummer des Pools muß zwischen null und der in der Konfigurationstabelle angegebenen maximalen Anzahl liegen.

Im Small und Medium Speichermodell ruft ***ini_mem*** die Funktion ***ini_nmem*** auf, während bei Compact und Large ***ini_fmem*** verwendet wird. Weitere Informationen zu den Speicher-Management Funktionen finden Sie im Kapitel "Speicherverwaltung" ab Seite 17.

■ Return Wert

ini_mem liefert einen *near* oder *far* Zeiger auf das erste Byte zurück, das von diesem Pool nicht mehr benötigt wird. Dieser Wert kann dann benutzt werden, um mit weiteren ***ini_mem***-Aufrufen weitere Pools zu definieren. Ein NULL-Zeiger bedeutet, daß ein Fehler aufgetreten ist. Dabei kann eine ungültige Pool-Nummer die Ursache sein oder ein Überlauf des belegten Speicherbereiches. Das passiert entweder bei near-Zeigern, wenn der Offset beim Initialisieren überläuft, oder bei far-Zeigern, wenn size*count größer als 1 MB ist. Die Fehlerursache wird durch den erweiterten Fehlercode im TCB beschrieben.

■ Siehe auch

get_mem (Seite 29), ***rel_mem*** (Seite 41)

■ Beispiel

Siehe Beispiel zu get_mem auf Seite 29

unsigned int ini_rsc(resource,maxcount)

unsigned int resource;	Nummer der Resource (0 .. max)
unsigned int maxcount;	Maximale Anzahl von Einheiten dieses Resourcen-Typs

Aufruf: near oder far call

■ Beschreibung

Mit der Funktion ***ini_rsc*** werden Resourcen initialisiert. Mit einer Resource können mehrere Einheiten dieses Types verwaltet werden. Die Funktionsweise ist im Kapitel "Resourcen" ab Seite 15 genauer erläutert. Eine Resource muß initialisiert werden, bevor sie zum erstenmal benutzt werden kann. Vorsicht ist geboten beim Initialisieren einer Resource während des Betriebs. Mit ***ini_rsc*** wird eine Resource unabhängig von ihrem momentanen Zustand initialisiert. Wenn also beispielsweise ein Task auf diese Resource wartet, so wird er nicht freigegeben. Er kann auch anschließend nicht mehr freigegeben werden, da sein Eintrag in der Warteschlange verlorengegangen ist. Die einzige Funktion die hier noch hilft (außer einer Neu-Initialisierung) ist ***trm_tsk***, mit der ein Task komplett terminiert wird. Er kann anschließend mit ***sta_tsk*** wieder neu gestartet werden.

■ Return Wert

ini_rsc gibt bei ungültiger Resourcen-Nummer einen Fehlercode zurück.

■ Siehe auch

pol_rsc (Seite 39), ***req_rsc*** (Seite 43), ***rel_rsc*** (Seite 42)

■ Beispiel

Assembler:

```
        include misc.mac        ;Miscellaneous macros
        include unimos.inc      ;UniMOS includes
        include unimosif.inc    ;UniMOS interface
        .....
        ini_tsk    ct
        ini_rsc    0,3
        ini_rsc    1,1
        ini_rsc    2,1
        dis_pre
        sta_tsk    task1,dgroup,stack0,0,0,1
```

C:

```
#include "unimos.h"
        .....
        ini_tsk (&ct) ;
        ini_rsc (0,3) ;
        ini_rsc (1,1) ;
        ini_rsc (2,1) ;
        dis_pre () ;
        sta_tsk (task1,dgroup,&stack0[STACKSIZE],0,0,1) ;
```

unsigned int ini_tsk(table)

CONFIG_TABLE *table; near/far pointer zur Konfigurationstabelle

Aufruf: near oder far call

■ Beschreibung

Die Funktion ***ini_tsk*** wird benutzt, um UniMOS zu initialisieren. Keine der anderen UniMOS-Funktionen darf aufgerufen werden, bevor das System initialisiert ist. ***ini_tsk*** hat nur einen Parameter, der zu einer Konfigurationstabelle zeigt, die die benötigten Systemparameter enthält. Diese Tabelle ist im Kapitel "Die Konfigurationstabelle" auf Seite 64 beschrieben. Bevor ***ini_tsk*** ausgeführt wird, können bereits andere, von UniMOS unabhängige Programme ausgeführt werden, wie z.B. die Initialisierung von Peripherie-Einheiten. Wenn der Systemspeicher batteriegepuffert ist, so kann möglicherweise auch die erneute Initialisierung des Systems durch ***ini_tsk*** umgangen werden. Voraussetzung dafür ist jedoch, daß das System vorher nicht "unkontrolliert", also z.B. während eines Systemaufrufes beendet wurde.

ini_tsk legt alle System-Tabellen in der benötigten Größe an. Für jeden Task wird ein TCB (Task Kontroll Block) angelegt und initialisiert und es wird eine "READY-Liste" angelegt, in der jeder Eintrag eine Priorität repräsentiert. Jeder Eintrag in dieser Liste ist ein Zeiger auf eine verkettete Liste von TCBs im ***READY*** Zustand. Wenn kein Task mit der betreffenden Priorität ***READY*** ist, dann wird hier ein NULL-Zeiger eingetragen. Wenn nur ein Task mit der entsprechenden Priorität ***READY*** ist, dann besteht die Liste aus nur diesem einen TCB. Es können aber beliebig viele TCBs verkettet werden. Wenn der Task in einen anderen Zustand als ***RUNNING*** oder ***READY*** übergeht, dann wird er aus dieser Liste entfernt. Der Übergang zwischen ***RUNNING*** und ***READY*** wird vom "Dispatcher" vollzogen, der die READY-Liste benutzt, um schnell den nächsten ***READY***-Task zu finden.

Derjenige Programmteil, der ***ini_tsk*** aufruft, wird automatisch zum Hintergrund-Task, dem Task mit der niedrigsten Priorität im System. Der Stack wird beibehalten. Das bedeutet also, daß vor dem Aufruf von ***ini_tsk*** der Stackpointer (SP) sowie das Stacksegment (SS) geladen werden müssen und daß der reservierte Bereich groß genug ist. Beachten Sie, daß bei einem niedrig ausgelasteten System die Interrupts meistens während der Ausführung des Hintergrund-Tasks auftreten (aus statistischen Gründen). Der Stackbereich muß entsprechend groß dimensioniert werden.

In dem Modul ***ini_tsk*** sind einige Werte definiert, die von anderen Modulen benötigt werden. Daher wird ini_tsk immer vom Linker aus der Bibliothek dazugebunden, auch wenn es nicht explizit aufgerufen wurde.

■ Return Wert

Die Anzahl der Resourcen und der Mailboxen ist beschränkt. Alle Resourcen und Mailboxen müssen jeweils in ein 64 kB großes Segment passen. Wurden beim Initialisieren zu große Werte angegeben, so wird hier ein entsprechender Fehlercode zurückgegeben.

■ Siehe auch

sta_tsk (Seite 50), ***ini_rsc*** (Seite 33)

■ Beispiel

Assembler:

```
                include misc.mac        ;Miscellaneous macros
                include unimos.inc      ;UniMOS includes
                include unimosif.inc    ;UniMOS interface
n_tasks         equ     64              ;number of tasks
n_prio          equ     10              ;number of priorities
n_mailboxes     equ     5               ;number of mailboxes
n_rsc           equ     5               ;number of resources
                .....
                .data
config_table    conf_tab_s    <n_tasks,n_prio,n_mailboxes,n_rsc, error_hdlr, disp_hook>
              .....
              ini_tsk config_table  ;initialize UniMOS
              .....
```

C:

```
#include "unimos.h"
              .....
#define N_TASKS 64          /* number of tasks */
#define N_PRIO 10           /* number of priorities */
#define N_MAILBOXES 10      /* number of mailboxes */
#define N_RSC 8             /* number of resources */
conftab ct = {N_TASKS,
              N_PRIO,
              N_MAILBOXES,
              N_RSC,
              error_hdlr,
              disp_hook} ;

              ini_tsk (&ct) ;
```

unsigned int pol_evt(list)

unsigned int list; Bit-Liste der Ereignisse

Aufruf: near oder far call

■ Beschreibung

Mit der Funktion ***pol_evt*** wird geprüft, ob eine Menge von Ereignissen eingetreten ist. Sind noch nicht alle Ereignisse eingetreten, so wartet dieser Task im Gegensatz zu ***wai_evt*** nicht, sondern er kehrt sofort zurück. Der Rückgabewert zeigt direkt die Liste derjenigen Ereignisse an, die noch nicht eingetreten sind.

■ Return Wert

pol_evt gibt eine Liste derjenigen Ereignisse zurück, die noch nicht eingetreten sind. Der Wert null wird zurückgegeben, falls alle Ereignisse eingetreten sind. ***pol_evt*** führt eine "Und"-Verknüpfung zwischen dem Parameter *list* und dem "*events*"-Eintrag im TCB aus. Das Ergebnis wird zurückgeliefert, jedoch nicht gespeichert.

■ Siehe auch

sig_evt (Seite 46), ***wai_evt*** (Seite 57), ***ini_evt*** (Seite 30)

■ Beispiel

Siehe Beispiel zu ini_evt auf Seite 30

*mail *pol_msg(mailbox)*

unsigned int mailbox; Nummer der Mailbox

<u>Aufruf:</u> near oder far call

■ Beschreibung

Mit der Funktion ***pol_msg*** wird die nächste Nachricht aus einer Mailbox abgeholt. Wenn keine Nachricht vorhanden ist, dann wird sofort in das aufrufende Programm zurückgekehrt ohne daß der Task warten muß. Damit unterscheidet sich die Funktion von ***rcv_msg*** (siehe Seite 40), die in diesem Falle wartet.

Nachrichten sind in einer Mailbox verkettet und zwar sortiert nach fallender Priorität. Wenn eine Nachricht mit der niedrigst möglichen Priorität verschickt wird (***snd_msg*** Seite 49), so wird sie hinten an bereits abgelegte Nachrichten angehängt. Beim Senden einer Nachricht mit höherer Priorität wird sie vor allen Nachrichten mit niedrigerer Priorität eingekettet, jedoch hinter solchen mit gleicher oder höherer Priorität. Somit wird immer die Nachricht mit der höchsten Priorität abgeholt.

Die Struktur einer Nachricht ist im Kapitel "Der Nachrichtenblock" auf Seite 68 beschrieben.

■ Return Wert

Bei ungültiger Mailbox-Nummer oder wenn keine Nachricht vorliegt, wird ohne daß gewartet wird ein NULL pointer zurückgegeben. Ein erweiterter Fehlercode im TCB (siehe "Der Task Control Block (TCB)" auf Seite 66) gibt dann die genaue Ursache an. Anderenfalls wird die Nachricht aus der Mailbox entfernt und ein Zeiger zu der Nachricht zurückgegeben. Dieser Zeiger ist abhängig vom Speichermodell ein *near* oder ein *far* Zeiger.

■ Siehe auch

rcv_msg (Seite 40), ***snd_msg*** (Seite 49)

■ Beispiel

Assembler:

```
          include misc.mac        ;Miscellaneous macros
          include unimos.inc      ;UniMOS includes
          include unimosif.inc    ;UniMOS interface
task1:
          snd_msg  3,<offset DGROUP:msg1>,<DGROUP>,10
          snd_msg  4,<offset DGROUP:msg2>,<DGROUP>,-1
          .....
task2:
t2_loop:  pol_msg  3
          or       ax,ax          ;any message received?
          jne      proc_msg2      ;yes, so process message
          call     any_thing      ;no, do something else
          jmp      t2_loop        ;and try it again
proc_msg2:         .....

task3:
          rcv_msg  4,0      ;wait for message without timeout
          or       ax,ax    ;any errors?
          jne      proc_msg3      ;no, so process message
          call     error_hdlr     ;errors occured, recover
proc_msg3:   .....
```

C:

```
#include "unimos.h"

task1:
        snd_msg(3,&msg1,10) ;
        snd_msg(4,&msg2,-1) ;

task2:
        while ((msg = pol_msg(3)) == NULL) do_anything() ;
        process (msg)

task3:
        if ((msg = rcv_msg (4,0)) == NULL) error_hdlr() ;
        else process (msg) ;
```

unsigned int pol_rsc(resource)

unsigned int resource; Nummer der Resource (0 .. max)

Aufruf: near oder far call

■ Beschreibung

Mit der Funktion ***pol_rsc*** versucht der aufrufende Task ein Element einer Resource zu erhalten. Das ist gelungen, wenn die Funktion ohne Fehler zurückkehrt. Wenn kein Element mehr verfügbar ist, dann kehrt die Funktion im Gegensatz zu ***req_rsc*** sofort mit einem entsprechenden Fehlercode zurück. Die Funktionsweise ist im Kapitel "Resourcen" ab Seite 15 genauer erläutert. Eine Resource muß mit ***ini_rsc*** initialisiert werden, bevor sie zum erstenmal benutzt werden kann.

■ Return Wert

pol_rsc gibt bei ungültiger Resourcen-Nummer und wenn kein Element mehr verfügbar ist einen Fehlercode zurück.

■ Siehe auch

ini_rsc (Seite 33), ***req_rsc*** (Seite 43), ***rel_rsc*** (Seite 42)

■ Beispiel

Assembler:

```
        include misc.mac        ;Miscellaneous macros
        include unimos.inc      ;UniMOS includes
        include unimosif.inc    ;UniMOS interface
        .....
        ini_tsk   ct
        ini_rsc   0,3
        ini_rsc   1,1
        ini_rsc   2,1
        pol_rsc   1       ;will get the resource
        pol_rsc   1       ;no more elements available
        rel_rsc   1       ;release resource
        req_rsc   1,0,-1  ;will get the resource
        req_rsc   1,0,-1  ;deadlock! will wait forever
        .....
```

C:

```
#include "unimos.h"
        .....
        ini_tsk (&ct) ;
        ini_rsc (0,3) ;
        ini_rsc (1,1) ;
        ini_rsc (2,1) ;
        pol_rsc (1)         /* will get the resource */
        pol_rsc (1)         /* no more elements available */
        rel_rsc (1)         /* release resource */
        req_rsc (1,0,-1)    /* will get the resource */
        req_rsc (1,0,-1)    /* deadlock! will wait forever */
        .....
```

*mail * far rcv_msg(mailbox,timeout)*

unsigned int mailbox;	Nummer der Mailbox
unsigned int timeout;	Maximale Wartezeit in Timer-ticks

Aufruf: far call

■ Beschreibung

Mit der Funktion ***rcv_msg*** wird eine Nachricht von einer Mailbox abgeholt. Wenn keine Nachricht vorhanden ist, dann wird der Task wartend gesetzt bis eine Nachricht an diese Mailbox geschickt wird. Die Funktion ***pol_msg*** (siehe Seite 37) kehrt bei nicht vorhandener Nachricht sofort zurück.

Optional kann ein timeout gesetzt werden, der die maximale Wartezeit in Anzahl von Timer-Interrupts angibt. Wenn hier eine 0 eingetragen wird, dann wartet der Task solange, bis tatsächlich eine Nachricht in der Mailbox abgelegt wird. Mit der Funktion ***rsm_tsk*** kann ein anderer Task diesen wartenden Task wieder weiterlaufen lassen. Das gilt unabhängig vom timeout.

Nachrichten sind in einer Mailbox verkettet und zwar sortiert nach fallender Priorität. Wenn eine Nachricht mit der niedrigst möglichen Priorität verschickt wird (***snd_msg*** Seite 49), so wird sie hinten an bereits abgelegte Nachrichten angehängt. Beim Senden einer Nachricht mit höherer Priorität wird sie vor allen Nachrichten mit niedrigerer Priorität eingekettet, jedoch hinter solchen mit gleicher oder höherer Priorität. Somit wird immer die Nachricht mit der höchsten Priorität abgeholt.

Die Struktur einer Nachricht ist im Kapitel "Der Nachrichtenblock" auf Seite 68 beschrieben.

■ Return Wert

Bei ungültiger Mailbox-Nummer oder bei abgeschalteter Taskumschaltung wird ein NULL pointer zurückgegeben, anderenfalls ein Zeiger auf die empfangene Nachricht. Abhängig vom Speichermodell ist das ein *near* oder ein *far* Zeiger. Auch bei einem Abbruch des Wartens (durch timeout oder durch ***rsm_tsk***) wird ein NULL pointer zurückgegeben. Der erweiterte Fehlercode im TCB gibt jeweils die Ursache des Fehlers an.

■ Siehe auch

pol_msg (Seite 37), ***snd_msg*** (Seite 49)

■ Beispiel

siehe Beispiel zu pol_msg auf Seite 38

unsigned int rel_mem(addr)
unsigned int rel_nmem(near_addr)
unsigned int rel_fmem(far_addr)

unsigned char *addr;	Zeiger zum Speicherblock
unsigned char near *addr;	near-Zeiger zum Speicherblock
unsigned char far *addr;	far-Zeiger zum Speicherblock

Aufruf: near oder far call

■ Beschreibung

Die Funktion ***rel_mem*** gibt einen vorher mit ***get_mem*** reservierten Speicherbereich wieder frei. Der Parameter ist ein *near* oder *far* Zeiger auf den Speicherblock, der freigegeben werden soll. Die Nummer des Pools, zu dem der Speicherblock gehört, muß nicht angegeben werden. Sie wird von ***rel_mem*** bestimmt, da sich verschiedene Pools nicht überlappen können. ***rel_fmem*** kann auch benutzt werden, um Speicherblöcke eines near-Speicherpools freizugeben.

Im Small und Medium Speichermodell ruft ***rel_mem*** die Funktion ***rel_nmem*** auf, während bei Compact und Large ***rel_fmem*** verwendet wird. Weitere Informationen zu den Speicher-Management Funktionen finden Sie im Kapitel "Speicherverwaltung" ab Seite 17.

■ Return Wert

rel_mem liefert einen Fehlercode zurück, falls die angegebene Adresse nicht zu einem Speicherblock gehört.

■ Siehe auch

get_mem (Seite 29), ***ini_mem*** (Seite 32)

■ Beispiel

Siehe Beispiel zu get_mem auf Seite 29

unsigned int far rel_rsc(resource)

unsigned int resource; Nummer der Resource (0 .. max)

Aufruf: far call

■ Beschreibung

Mit der Funktion ***rel_rsc*** gibt der Task ein Element einer Resource frei, das er sich vorher mit ***pol_rsc*** oder ***req_rsc*** geholt hat. Die Funktionsweise ist im Kapitel "Resourcen" ab Seite 15 genauer erläutert.

■ Return Wert

rel_rsc gibt bei ungültiger Resourcen-Nummer einen Fehlercode zurück. Auch wenn ***rel_rsc*** aufgerufen wird, obwohl alle Elemente der Resource verfügbar sind, wird ein Fehlercode erzeugt.

■ Siehe auch

ini_rsc (Seite 33), ***req_rsc*** (Seite 43), ***pol_rsc*** (Seite 39)

■ Beispiel

Siehe Beispiel zu pol_rsc auf Seite 39

unsigned int far req_rsc(resource, ticks, prio)

unsigned int resource;	Nummer der Resource (0 .. max)
unsigned int ticks;	Timeout
unsigned char prio;	Priorität (0 .. 255)

Aufruf: far call

■ Beschreibung

Mit der Funktion ***req_rsc*** fordert ein Task ein Element einer Resource an. Wenn es verfügbar ist, dann wird dieses Element belegt und die Funktion kehrt sofort zurück. Andernfalls wartet der Task darauf, daß ein solches Element frei wird. Seine Position innerhalb der Warteschlange kann über den Parameter *prio* bestimmt werden. Dabei ist 0 die höchste Priorität und 255 die niedrigste. Man kann mit dem Parameter *ticks* eine maximale Wartezeit angeben. Soll der Task unbegrenzt lange warten, so muß für *ticks* eine null eingegeben werden.

■ Return Wert

req_rsc gibt bei ungültiger Resourcen-Nummer einen Fehlercode zurück. Wenn das Warten durch einen Timeout oder von einem anderen Task terminiert wurde, wird ebenfalls ein Fehlercode zurückgegeben.

■ Siehe auch

ini_rsc (Seite 33), ***rel_rsc*** (Seite 42), ***pol_rsc*** (Seite 39)

■ Beispiel

Siehe Beispiel zu pol_rsc auf Seite 39

unsigned int far rsm_tsk(tasknumber)

unsigned int tasknumber;   Tasknummer des wiederzustartenden Tasks

Aufruf: far call

■ Beschreibung

rsm_tsk führt den Task vom ***SUSPENDED*** in den ***RUNNING***-Zustand über, ist damit also das Gegenstück zum Aufruf von ***sus_tsk*** (siehe Seite 52). Der Task *tasknumber* wird an der Stelle fortgeführt, wo er von ***sus_tsk*** unterbrochen wurde. Ein eventuell noch laufender Timer wird zurückgesetzt, d.h. daß ein Task, der eigentlich noch länger warten wollte, vorzeitig wieder in den ***READY***-Zustand gesetzt werden kann.

Die Funktion ***rsm_tsk*** kann außerdem benutzt werden, um einen Task, der aufgrund eines ***wai_int*** Aufrufes auf ein externes Ereignis wartet, wieder von ***WAITING*** nach ***READY*** zu überführen. Der so wiedergestartete Task gibt einen Fehlercode zurück, damit er die Ursache seines Weiterlaufens erfährt. Auch ein Task, der auf eine Nachricht oder eine Resource wartet (***rcv_msg*** oder ***req_rsc***) kann mit ***rsm_tsk*** weitergeführt werden.

■ Return Wert

rsm_tsk gibt einen Fehlercode zurück. Für eine Beschreibung der Fehlercodes siehe das Kapitel ***Fehlercodes*** ab Seite 70.

■ Siehe auch

sus_tsk (Seite 52)

■ Beispiel

Assembler:

```
        include misc.mac        ;Miscellaneous macros
        include unimos.inc      ;UniMOS includes
        include unimosif.inc    ;UniMOS interface
task1:
        snd_msg  10,<offset DGROUP:msg1>,<DGROUP>,-1
        sus_tsk  0              ;wait until message is processed
        .....                   ; now msg1 can be reused

task2:
        rcv_msg  10,0           ;wait for message without timeout
        .....                   ;process message
        rsm_tsk  1              ;acknowledge
        .....
```

C:

```
#include "unimos.h"

task1:
    snd_msg(10,&msg1,-1) ;  /* send a message */
    sus_tsk(0) ;            /* wait until message is processed */
    .....                   /* now msg1 can be reused */

task2:
    if ((msg = rcv_msg (10,0)) == NULL) error_hdlr() ;
    else     {process (msg) ;
```

rsm_tsk (1) ; }

unsigned int far sig_evt(list,task)

unsigned int list;	Bit-Liste der Ereignisse
unsigned int task;	Task Nummer

Aufruf: far call

■ Beschreibung

sig_evt signalisiert den Eintritt eines oder mehrerer Ereignisse an einen Task. Es wird eine "Und"-Verknüpfung zwischen dem invertierten Wert des Parameters *list* und dem "*events*"-Eintrag im TCB ausgeführt. Das Ergebnis wird im TCB gespeichert. Ist das Ergebnis null geworden und wartet der Task auf Ereignisse, so wird er aus dem Wartezustand nach READY überführt.

■ Return Wert

sig_evt gibt einen Fehlercode zurück, wenn eine ungültige Tasknummer angegeben wurde.

■ Siehe auch

pol_evt (Seite 36), ***wai_evt*** (Seite 57), ***ini_evt*** (Seite 30)

■ Beispiel

Siehe Beispiel zu ini_evt auf Seite 30

unsigned int far sig_int(tasknumber)

unsigned int tasknumber; Tasknummer des wiederzustartenden Tasks

Aufruf: far call

■ Beschreibung

sig_int signalisiert an einen wartenden Task den Eintritt eines Interrupts. Der wartende Task wird wieder ***READY*** gesetzt und wenn es der Dispatcher gestattet, dann erhält er den Prozessor. Wenn der Task nicht auf einen Interrupt wartet, dann wird diese Funktion ignoriert. Zur Vermeidung von "deadlocks" muß also sichergestellt werden, daß ***wai_int*** vor ***sig_int*** aufgerufen wird.

Da durch ***sig_int*** eine Taskumschaltung erfolgen kann, ist der Zeitpunkt der Rückkehr aus dieser Funktion vollkommen unbestimmt. Daher sollte der Aufruf von ***sig_int*** am Ende des Interrupt-Handlers erfolgen. Falls ein "End of Interrupt"-Kommando benötigt wird, dann sollte es auf jeden Fall vorher erfolgen.

Hinweis: ***sig_int*** wird normalerweise aus einem Interrupt-Handler heraus aufgerufen. Ein Aufruf von einem Task ist jedoch nicht ausgeschlossen.

■ Return Wert

Normalerweise gibt ***sig_int*** den Wert "kein Fehler" zurück. Bei falscher Tasknummer oder falls der Task nicht auf ein ***sig_int*** wartet, wird jedoch ein entsprechender Fehlercode zurückgeliefert.

■ Siehe auch

wai_int (Seite 58), ***tim_int*** (Seite 53)

■ Beispiel

Assembler:

```
                include misc.mac        ;Miscellaneous macros
                include unimos.inc      ;UniMOS includes
                include unimosif.inc    ;UniMOS interface
task1:
                wai_int     100   ;wait for interrupt with timeout
                cmp         ax,err_noerror ;any error ?
                bne         error ;wait was terminated
                .....             ;continue

interrupt handler:
                push        regs  ;save registers
                .....             ;process interrupt
                pop         regs  ;restore registers
                fint              ;end of interrupt (V25)
                sig_int     1     ;signal an interrupt
                .....             ;continue
```

C:

```
#include "unimos.h"

task1:
        if (wai_int(100) != ERR_NOERROR) error_hdlr();
        else .... /* continue if no error */

interrupt handler:
void interrupt inthdlr(void)
{
        ...                     /* process interrupt */
        fint ()                 /* finish interrupt (V25 only) */
        sig_int(1) ;
}
```

unsigned int far snd_msg(mailbox,mail,priority)

unsigned int mailbox;	Nummer der Mailbox
mail *mail;	Zeiger zu der Nachricht
unsigned char priority;	Priorität der Nachricht

Aufruf: far call

■ Beschreibung

Mit ***snd_msg*** wird eine Nachricht zu einer Mailbox geschickt. Wenn bereits ein anderer Task auf diese Nachricht wartet (aufgrund eines ***rcv_msg*** Aufrufes), so wird dieser Task in den READY Zustand überführt und der Scheduler aufgerufen. Anderenfalls wird die Nachricht in die Mailbox an passender Stelle eingefügt.

Nachrichten sind in einer Mailbox verkettet und zwar sortiert nach fallender Priorität. Eine Nachricht mit niedriger Priorität wird hinten an bereits abgelegte Nachrichten angehängt. Nachrichten mit hoher Priorität werden vor allen Nachrichten mit niedrigerer Priorität eingekettet, jedoch hinter solchen mit gleicher oder höherer Priorität. Prioritäten können von 0 (höchste Priorität) bis 255 (niedrigste Priorität) vergeben werden.

Die Struktur einer Nachricht ist im Kapitel "Der Nachrichtenblock" auf Seite 68 beschrieben.

■ Return Wert

snd_msg gibt einen Fehlercode zurück. Für eine Beschreibung der Fehlercodes siehe das Kapitel "Fehlercodes" ab Seite 70.

■ Siehe auch

pol_msg (Seite 37), ***rcv_msg*** (Seite 40)

■ Beispiel

siehe Beispiel zu pol_msg auf Seite 38

unsigned int far sta_tsk(code, data, stack, tasknumber, priority, timeslice)

void code(void);	Startadresse des Task (near/far)
unsigned int _seg *data;	Segmentadresse des Datensegments
unsigned int *stack;	Stackpointer (near/far)
unsigned int tasknumber;	Nummer des Task
unsigned int priority;	Priorität des Task
unsigned char timeslice;	Dauer einer Zeitscheibe

Aufruf: far call

■ Beschreibung

Mit der Funktion ***sta_tsk*** wird ein neuer Task gestartet. Der Task muß im Zustand ***DORMANT*** sein, d.h. es muß ein Task-Kontrollblock existieren. ***sta_tsk*** führt den Task in den Zustand ***READY*** über und ruft anschließend sofort den Dispatcher auf, so daß der Task unmittelbar in den ***RUN***-Zustand übergeht, falls er der höchstpriorisierte ***READY***-Task ist und Taskumschaltung freigegeben ist.

Mit dem Parameter ***code*** wird die Startadresse des Task spezifiziert. Für die Speichermodelle mit kurzem Codezeiger (small, compact) ist es ein *near pointer*, ansonsten ein *far pointer*. Die Adresse des Datensegmentes wird mit dem Parameter ***data*** angegeben. Dieser Wert ist unabhängig vom Speichermodell immer eine Segmentadresse, also ein Wort (zwei Bytes). Bei Compilern, die diese Adressierungsart nicht unterstützen, sollte dieser Datentyp ein Wort sein. Der Wert wird vor dem Start des Tasks in die Register DS und ES geladen. Wenn diese Register vom Task selbst geladen werden, dann kann hier auch irgendein anderer Wert angegeben werden. ***stack*** zeigt zum Ende des für diesen Task reservierten Stackbereiches, da ein Stack bei Prozessoren dieser Art nach unten wächst. Bei den Speichermodellen mit kurzem Datenzeiger (small, medium) wird hier nur der Offset übergeben. Es wird angenommen, daß der Stack in der DGROUP liegt. Da Stackoperationen immer Wortoperationen sind, sollte darauf geachtet werden, daß der Stackpointer auf Wortgrenze ausgerichtet ist.

Der Parameter ***priority*** ordnet dem Task eine Anfangs-Priorität zu. Die maximale Anzahl von Prioritäten kann beim Initialisieren des Systems in der Konfigurationstabelle (siehe Anhang ab Seite 61) angegeben werden. Dabei steigt die Priorität mit sinkendem Wert, 0 ist also die höchste Priorität im System.

Mit ***timeslice*** wird die Dauer einer Zeitscheibe angegeben. Der Wert wird dann benutzt, wenn weitere Tasks mit der gleichen Priorität wie dieser Task ***READY*** sind. Die Dauer der Zeitscheibe kann zwischen 1 und 255 gewählt werden. Sie gibt die Anzahl von Timer-Interrupts an, für die dieser Task den Prozessor bekommt. Nähere Erläuterungen dazu befinden sich im Kapitel über den Systemtimer auf Seite 11.

■ Return Wert

sta_tsk gibt einen Fehlercode zurück. Für eine Beschreibung der Fehlercodes siehe das Kapitel "Fehlercodes" ab Seite 70.

■ Siehe auch

trm_tsk (Seite 56), ***sus_tsk*** (Seite 52), ***exi_tsk*** (Seite 27), ***rsm_tsk*** (Seite 44)

■ Beispiel

Assembler:

```
        include misc.mac        ;Miscellaneous macros
        include unimos.inc      ;UniMOS includes
        include unimosif.inc    ;UniMOS interface
        .....
        ini_tsk config_table    ;initialize UniMOS
        dis_pre                 ;prevent tasks from being started
        sta_tsk task1,DGROUP,stack0,0,0,0
        sta_tsk task2,DGROUP,stack1,1,1,0
        sta_tsk task3,DGROUP,stack2,2,1,0
        ena_pre                 ;start task with highest priority
```

C:

```
#include "unimos.h"
#define STACKSIZE 128       /* number of words for the stack */
extern WORD _seg *dgroup ; /* Segment for DGROUP */
        .....
        ini_tsk (&ct) ;
        dis_pre () ;
        sta_tsk (task1,dgroup,&stack1[STACKSIZE],1,1,5) ;
        sta_tsk (task2,dgroup,&stack2[STACKSIZE],2,1,5) ;
        sta_tsk (task3,dgroup,&stack3[STACKSIZE],3,1,5) ;
        ena_pre () ;
```

Beide Beispiele zeigen die gleiche Startsequenz für UniMOS. Nach dem Aufruf von ini_tsk wird dieser Code zum Hintergrundtask mit der niedrigsten Priorität. Sobald dann ein weiterer Task mit höherer Priorität gestartet wird, bekommt dieser den Prozessor. Um die Initialisierung ungestört ablaufen zu lassen, wird hier die Taskumschaltung verhindert.

unsigned int far sus_tsk(ticks)

unsigned int ticks; Anzahl Timer ticks

Aufruf: far call

■ Beschreibung

sus_tsk führt den Task vom ***RUN*** in den ***SUSPEND***-Zustand über. Der Parameter *ticks* gibt die Zeitdauer in Timer-ticks an, bis der Task automatisch wieder in den ***READY***-Zustand überführt wird (siehe Kapitel ***Der Systemtimer*** auf Seite 11). Wenn ticks=0 ist, dann bleibt der Task suspended, bis er durch einen anderen Task mit der Funktion ***rsm_tsk*** (siehe Seite 44) wieder in den ***READY***-Zustand überführt wird.

■ Return Wert

sus_tsk gibt einen Fehlercode zurück, wenn momentan die Taskumschaltung durch ***dis_pre*** (siehe Seite 24) verhindert ist. Ist der Timer abgelaufen oder wurde der Task mit ***rsm_tsk*** weitergeführt, so wird ein entsprechender Fehlercode zurückgegeben.

■ Siehe auch

rsm_tsk (Seite 44)

■ Beispiel

Siehe Beispiel zu ***rsm_tsk*** auf Seite 44

void far tim_int(void)

keine Parameter

Aufruf: far call

■ Beschreibung

Ein ***tim_int*** Aufruf signalisiert an UniMOS, daß ein Timer-Interrupt aufgetreten ist. UniMOS wird nun die Zeitscheibe des momentanen Tasks dekrementieren und bei Erreichen von null den nächsten ***READY*** Task mit der gleichen Priorität auswählen. Außerdem werden die Timer der wartenden Tasks dekrementiert und bei Erreichen von null wird der entsprechende Task ***READY*** gesetzt.

Wichtiger Hinweis: Der Timer Interrupt-Handler hat möglicherweise sehr viele Aufgaben zu erledigen, die auch je nach Systemauslastung recht lange dauern können. Innerhalb von tim_int wird sichergestellt, daß nicht während der gesamten Laufzeit die Interrupts disabled sind. Sie werden regelmäßig für kurze Zeit freigegeben, so daß externe Ereignisse behandelt werden können. Es wird jedoch kein Task ausgeführt bevor ***tim_int*** beendet ist, auch dann nicht, wenn eine Interrupt-Routine z.B. ***sig_int*** aufgerufen hat. Wenn ein weiterer ***tim_int*** Aufruf erfolgt bevor der vorherige abgeschlossen ist, so wird der zweite Aufruf ignoriert. Das verhindert unvorhersehbares Verhalten für den Fall, daß die Zeitdauer zwischen zwei Aufrufen zu kurz wird. Es führt aber dazu, daß Timer ticks verloren gehen können. Zu einer hohen Belastung des Timer Interrupt-handlers führt insbesondere eine hohe Anzahl von suspendierten und wartenden Tasks mit timeout. Besonders zeitkritisch wird die Situation dann, wenn viele Tasks zur selben Zeit aufgrund eines gleichzeitig abgelaufenen Timers wieder in den READY Zustand gesetzt werden müssen. Hinweise im Kapitel "Leistungsdaten" ab Seite 73 helfen bei der Abschätzung der benötigten Zeit.

Da durch ***tim_int*** eine Taskumschaltung erfolgen kann, ist der Zeitpunkt der Rückkehr aus dieser Funktion vollkommen unbestimmt. Daher sollte der Aufruf von ***tim_int*** am Ende des Interrupt-Handlers erfolgen. Falls ein "End of Interrupt"-Kommando benötigt wird, dann sollte es auf jeden Fall vorher erfolgen.

Hinweis: ***tim_int*** wird normalerweise aus einem Interrupt-Handler heraus aufgerufen. Ein Aufruf von einem Task ist jedoch nicht ausgeschlossen.

■ Return Wert

Der zurückgelieferte Wert hat keine Bedeutung.

■ Siehe auch

sig_int (Seite 47)

■ Beispiel

Assembler:

```
        include misc.mac        ;Miscellaneous macros
        include unimos.inc      ;UniMOS includes
        include unimosif.inc    ;UniMOS interface
```

timer interrupt handler:

```
        push    regs    ;save registers
        .....           ;process interrupt
        pop     regs    ;restore registers
        fint            ;end of interrupt (V25)
        tim_int         ;signal a timer interrupt
```

C:

```
#include "unimos.h"
```

interrupt handler:

```
void interrupt timint(void)
{
        ...                     /* process interrupt */
        fint ()                 /* finish interrupt (V25 only) */
        tim_int() ;
}
```

void far trm_slc(void)

keine Parameter

Aufruf: far call

■ Beschreibung

Mit der Funktion ***trm_slc*** wird die momentane Zeitscheibe beendet. Der gerade ausführende Task möchte kurz warten und einem anderen Task mit gleicher Priorität die Chance geben, den Prozessor zu erhalten. Der momentane Task bleibt im ***READY***-Zustand, wird aber an das Ende der READY-Liste gehängt. Damit erhält er den Prozessor erst wieder, wenn die Zeitscheiben aller nun vor ihm liegenden Tasks abgelaufen sind. Wenn nur dieser eine Task in der READY-Liste steht, dann wird er sofort wieder weiterlaufen. Ein Task mit niedrigerer Priorität wird den Prozessor also nicht erhalten, da der momentane Task immer ***READY*** bleibt. Wenn auch niedriger priorisierte Tasks den Prozessor erhalten sollen oder eine Mindest-Wartezeit garantiert werden muß, dann muß die Funktion ***sus_tsk*** (Seite 52) mit einem Parameter ungleich null verwendet werden.

■ Return Wert

trm_slc gibt einen Fehlercode zurück, wenn momentan die Taskumschaltung durch ***dis_pre*** (siehe Seite 24) verhindert ist.

■ Siehe auch

sus_tsk (Seite 52)

■ Beispiel

Assembler:

```
include misc.mac        ;Miscellaneous macros
include unimos.inc      ;UniMOS includes
include unimosif.inc    ;UniMOS interface
.....
trm_slc                 ;terminate the current time slice
```

C:

```
#include "unimos.h"
    ...
    trm_slc ()          /* terminate the current time slice */
```

unsigned int far trm_tsk(tasknumber)

unsigned int tasknumber; Nummer des Tasks

Aufruf: far call

■ Beschreibung

Die Funktion ***trm_tsk*** führt den angegebenen Task in den DORMANT Zustand über. Sie hat damit denselben Effekt wie die Funktion ***exi_tsk***, die jedoch nur für den eigenen Task gilt. Mit ***trm_tsk*** können auch solche Tasks beendet werden, die WAITING oder SUSPENDED sind. Auch wenn für den Task ein Timeout spezifiziert wurde, kann er terminiert werden. Konflikte können jedoch dann entstehen, wenn der zu beendende Task Speicherblöcke oder Resourcen hat. Diese werden nicht automatisch freigegeben, da nirgends abgespeichert ist, wem diese Objekte gerade gehören. Sie sind in diesem Fall bis zum Neustart des Systems verloren.

■ Return Wert

trm_tsk gibt einen Fehlercode zurück, wenn eine ungültige Tasknummer angegeben wurde oder die Taskumschaltung mit ***dis_pre*** verhindert ist und der Task versucht sich selbst zu beenden.

■ Siehe auch

sta_tsk (Seite 50), ***sus_tsk*** (Seite 52), ***exi_tsk*** (Seite 27), ***rsm_tsk*** (Seite 44)

■ Beispiel

Assembler:

```
        include misc.mac        ;Miscellaneous macros
        include unimos.inc      ;UniMOS includes
        include unimosif.inc    ;UniMOS interface
        .....
Task 1:
        get_mem 0       ;get memory block
        wai_int 0       ;wait for interrupt. No timeout
        .....

Task 2:
        trm_tsk 1       ;terminate Task 1, memory block is lost
```

C:

```
#include "unimos.h"

Task 1:
        wai_int (0) ;   /* wait for interrupt. No timeout */
        .....

Task 2:
        trm_tsk (1) ;   /* terminate Task 1, memory block is lost*/
```

unsigned int wai_evt(list, timeout)

unsigned int list;	Bit-Liste der Ereignisse
unsigned int timeout;	Timeout

Aufruf: far call

■ Beschreibung

wai_evt wartet auf ein oder mehrere Ereignisse. Im Gegensatz zu ***pol_evt*** wartet diese Funktion, falls noch nicht alle Ereignisse eingetreten sind. Wenn für den Parameter *timeout* ein Wert ungleich null eingegeben wird, dann wird das Warten nach Ablauf dieser Zeit abgebrochen. Es wird eine "Und"-Verknüpfung zwischen Wert des Parameters *list* und dem "*events*"-Eintrag im TCB ausgeführt. Das Ergebnis wird im TCB gespeichert. Ist das Ergebnis null, so wird der Task sofort weitergeführt, anderenfalls wartet er. Damit können mit dem Parameter *list* von der von ***ini_evt*** angelegten Liste Ereignisse entfernt, jedoch keine hinzugefügt werden.

■ Return Wert

wai_evt gibt normalerweise den Wert "kein Fehler" zurück. Falls das Warten von einem anderen Task oder aufgrund eines Timeouts abgebrochen wurde, wird ein entsprechender Fehlercode zurückgegeben.

■ Siehe auch

pol_evt (Seite 36), ***sig_evt*** (Seite 46), ***ini_evt*** (Seite 30)

■ Beispiel

Siehe Beispiel zu ini_evt auf Seite 30

unsigned int far wai_int(ticks)

unsigned int ticks; Anzahl Timer ticks

Aufruf: far call

■ Beschreibung

wai_int setzt den momentanen Task ***WAITING***. In diesem Zustand bleibt der Task, bis ein Interrupt-Handler ihn mit der Funktion ***sig_int*** wieder nach ***READY*** überführt. Mit ***wai_int*** wartet das System gewöhnlich auf externe Ereignisse. Damit auch bei Nichteintreten des Ereignisses der Task wieder weiterlaufen kann, kann ein anderer Task die Funktion ***rsm_tsk*** benutzen, um den wartenden Task wieder loslaufen zu lassen. Damit der wartende Task erkennen kann, ob das Ereignis eintrat oder nicht, wird ein unterschiedlicher Wert zurückgegeben.

■ Return Wert

wai_int gibt den Wert "kein Fehler" zurück, wenn der wartende Task durch die Funktion ***sig_int*** in den ***READY***-Zustand gesetzt wurde. Wurde der Task mit ***rsm_tsk*** nach ***READY*** überführt, dann wird der Fehler "WAIT erzwungenermaßen beendet" zurückgegeben. Auch bei abgeschalteter Taskumschaltung wird ***wai_int*** sofort mit dem entsprechenden Fehler beendet (siehe das Kapitel ***Fehlercodes*** auf Seite 70). Wird der Task nach Ablauf der Timer-ticks wieder gestartet, so wird der Wert "Wait durch Timer beendet" zurückgegeben.

■ Siehe auch

sig_int (Seite 47), ***tim_int*** (Seite 53)

■ Beispiel

Siehe Beispiel zu sig_int auf Seite 48

3.1. Hilfs- und Testfunktionen

UniMOS stellt einige Hilfs- und Testfunktionen zur Verfügung, die bei der Entwicklung des Systems benutzt wurden und möglicherweise auch für den Anwender interessant sein können. Die Funktionen werden im folgenden kurz beschrieben.

unsigned int get_tn(tcb)

struct tcbs _seg *tcb; Segment Zeiger zum TCB

Aufruf: near oder far call

■ Beschreibung

get_tn errechnet aus einer gegebenen TCB Adresse die Tasknummer.

■ Return Wert

Wenn kein Fehler auftrat wird die Tasknummer zurückgegeben. Ist der Abstand des TCB vom ersten TCB nicht ganzzahlig durch die Länge eines TCB teilbar ist, dann wird zur Fehleranzeige der Wert "-1" zurückgegeben.

■ Siehe auch

■ Beispiel

```
extern scb far *pscb ;    /* far pointer to System Control Block (SCB) */
...
task_number = get_tn (pscb->scb_ptcbcurr) ;
```

3.2. Debugfunktionen

Außer den im vorigen Kapitel beschriebenen Hilfs- und Testfunktionen, existieren einige Funktionen zum Debuggen des UniMOS Anwendungsprogramms. Diese Debug-Funktionen setzen UniMOS Code voraus, der nur in den mit einem „D“ gekennzeichneten Bibliotheken verfügbar ist (also z.B. UM86**D**S). Dieser zusätzliche Code kostet etwas Rechenzeit und Speicherplatz. Daher sollte das Anwendungsprogramm nach der Debugphase mit den normalen Bibliotheken gelinkt werden. Zu beachten ist außerdem, daß in einigen Kontrollblökken zusätzliche Variable eingefügt wurden und daher die Offsets unterschiedlich sein können.

*BYTE *get_inf(BYTE *info, WORD item)*

BYTE *info;	Zeiger zum Info Datenbereich
WORD item;	Gibt an, welche Information gewünscht wird

Aufruf: near oder far call

■ Beschreibung

get_inf liefert die durch den Parameter ***item*** angefragte Information zurück. Die Information wird in den Datenbereich an der Adresse ***info*** geschrieben, den das aufrufende Programm in ausreichender Größe zur Verfügung stellen muß. Folgende Informationsarten sind definiert:

item = 1: Memory Pool Information

returns: WORD info[2*n] = pcb_count[n]
WORD info[2*n + 1] = pcb_mincount[n]

Dabei hat n einen Wert zwischen 0 und der Anzahl verwendeter Memory-Pools. ***info*** muß also pro Memory-Pool vier Bytes reservieren. pcb_count gibt die Anzahl der momentan noch verfügbaren Speicherblöcke in diesem Pool an, während pcb_mincount die seit dem Systemstart minimal verfügbare Anzahl angibt. Mit diesen Informationen kann man also feststellen, ob die Anzahl der Speicherblöcke ausreicht. Außerdem kann man beispielsweise nach beenden von UniMOS feststellen, ob auch alle Speicherblöcke wieder zurückgegeben wurden. Das ist dann der Fall, wenn pcb_count am Ende gleich dem Anfangswert ist. ***get_inf*** mit item=1darf erst nach der Funktion ***ini_tsk*** aufgerufen werden. Sinnvolle Rückgabewerte kann man aber erst erwarten, nachdem die Speicherblöcke mit ***ini_mem*** initalisiert wurden. Auch nach dem beenden aller Tasks durch ***trm_tsk*** liefert ***get_inf*** noch gültige Ergebnisse.

■ Return Wert

Es wird immer ein Zeiger auf ***info*** zurückgegeben.

■ Siehe auch

■ Beispiel

```
WORD poolinfo[2*N_MEMPOOL] ;
...
get_inf((BYTE *) &poolinfo, 1) ;
printf ("POOL -Start: %5.1u -End: %5.1u -Minimum: %5.1u -Used: %5.1u\n",
          POOL_COUNT, poolinfo[0], poolinfo[1], POOL_COUNT-poolinfo[1]) ;
```

4. Anhang

In diesem Kapitel werden die Kontrollblöcke von UniMOS sowie weitere wichtige Details beschrieben. Für alle Kontrollblöcke gilt generell, daß man nur mit Hilfe der STRUCs (in der Datei UNIMOS.INC für Assembler-Programme bzw. UNIMOS.H für C-Programme) auf sie zugreifen sollte. Die absoluten Offsets können sich in zukünftigen Versionen ändern. Das gilt auch für die Länge der Strukturen, die aufgrund von Erweiterungen größer werden können. Verlassen Sie sich auch möglichst nicht auf Datentypen. Falls es notwendig sein sollte, können z.B. leicht die Prioritäten und die Dauer von Zeitscheiben im TCB von einem Byte auf ein Wort erweitert werden.

4.1. UniMOS Publics und Externals

UniMOS hat einige wenige ***public*** und ***extrn*** Variablen definiert, die im Anwenderprogramm benutzt werden können beziehungsweise als PUBLIC definiert sein müssen. Diese Variablen werden im folgenden beschrieben.

Externe Variablen:

```
extrn   pswinit:abs   ;Program Status Wort ;(initial)
extrn   tcbsize:abs   ;Größe eines TCB in Paragraphen
```

pswinit gibt den Wert des Programm Status Wortes (Flag Register) beim Starten eines Task an. Das wird normalerweise ein fester Wert sein, jedoch kann es zu Testzwecken interessant sein diesen Wert zu verändern.

Die Variable ***tcbsize*** legt die Größe des Task Kontroll Blockes (TCB) fest. UniMOS benötigt hier für seine Variablen eine Mindestgröße. Es ist jedoch möglich, den TCB für benutzerspezifische Anwendungen zu vergrößern. Der zusätzliche Bereich kann dann z.B. genutzt werden um lokale Variablen für jeden einzelnen Task abzulegen. Diese lokalen Variablen können Registerinhalte von externen Hardwaredevices sein, wie beispielsweise die Register eines Fließpunkt-Prozessors oder auch eines Floppy-Disk Controllers. Durch die Definition einer "Dispatcher-Hook"-Funktion kann man die eigene Software zum Ein- und Auslagern dieser Werte zur Verfügung stellen.

Um die externen Referenzen beim Linken aufzulößen, müssen im Anwenderprogramm folgende Anweisungen enthalten sein:

```
        public    pswinit
        public    tcbsize
```

Die zugehörigen Definitionen können z.B. folgendermaßen aussehen:

```
tcbsize   equ   (size tcbs)/16
pswinit   equ   0F202h
```

Damit wird die Größe eines TCB automatisch auf die von UniMOS benötigte Größe festgelegt. Wenn keine eigenen Erweiterungen im TCB benötigt werden, dann ist das die optimale Lösung. Ein kleinerer Wert als dieser darf nicht angegeben werden, da dann die TCBs nicht angelegt werden können. Die Länge eines TCB ist nach oben nur durch den vorhandenen Systemspeicher begrenzt. Es ist zu beachten, daß jeder Task genau einen TCB benötigt. Hat man also beispielsweise 128 Tasks und gibt tcbsize=40h an (entsprechend 1024 Bytes TCB pro Task), so werden allein für TCBs bereits 128 kB Speicher belegt.

Public Variablen:

```
        public  version ;
version equ     0200h   ;Versionsnummer
```

Diese Public Anweisung gibt dem Benutzer Zugriff auf die UniMOS-Versionsnummer. Sie kann benutzt werden um verschiedene Versionen voneinander zu unterscheiden. Um auf diesen Wert zugreifen zu können, muß folgende Anweisung in einen Assembler-Quelltext eingefügt werden:

```
        extrn   version:abs
```

4.2. Speicherbelegung und Speicherbedarf

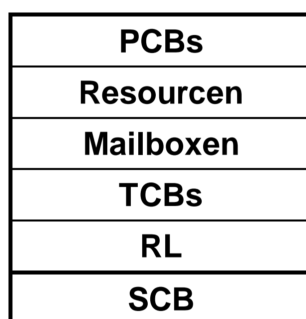
Im UniMOS Quellcode ist ein Segment mit dem Namen "unimos" mit folgender Anweisung definiert:

```
unimos    segment para public
unimos    ends
```

An der Startadresse dieses Segmentes beginnen die UniMOS Kontrollblöcke, deren Größe von der Konfiguration des Systems abhängt, also der Anzahl der Tasks, der Mailboxen, der Prioritäten u.s.w. Diese Segmentdefinition stellt für UniMOS nur die Spezifikation dieser Startadresse sicher. Ab hier muß genügend RAM-Bereich zur Verfügung stehen, um alle UniMOS Kontrollblöcke aufzunehmen. Das kann theoretisch bis zu 1 MB sein, also weit mehr als die 64 kB eines Segmentes.

Im UniMOS Quelltext finden sich deshalb keine Anweisungen zur Speicherreservierung (DB, DW, ...), sondern es ist dem Benutzer überlassen, genügend RAM-Bereich zur Verfügung zu stellen. Zur Laufzeit des Programmes muß dann das unimos-Segment zu einem genügend großen unbenutzten RAM-Bereich zeigen. Mit einem EXE nach HEX Konverter (wie z.B. UniLOC) kann man dies für ein eigenständiges Controllersystem leicht erreichen. Man kann damit jedes Segment an eine beliebige Adresse "relokieren".

Etwas komplizierter wird die Sache unter DOS. Hier darf man nicht ungestraft beliebige Speicherbereiche frei benutzen. Daher muß der Bereich für diese Kontrollblöcke unter Aufsicht von DOS belegt werden. Das bedeutet, daß das Benutzerprogramm eine unimos-Segment-Definition enthalten muß, die jedoch jetzt den Speicherbereich definiert. Da die von UniMOS verwendeten Datenstrukturen in der Include-Datei "UNIMOS.INC" definiert sind, kann man auch die Fähigkeiten des Assemblers benutzen um die Größe jeweils zu errechnen. Dazu soll zunächst beschrieben werden, welche Datenblöcke UniMOS beim Initialisieren für sich reserviert. Die folgende Skizze stellt den Aufbau des unimos-RAM-Bereiches dar:



Der unimos-RAM-Bereich beginnt mit dem ***System Control Block***, dem SCB, der auf Seite 65 im Detail beschrieben ist. Seine Länge ist zwar unabhängig von der Konfiguration, sie variiert jedoch mit dem verwendeten Speichermodell, da *near* oder *far*-pointer verwendet werden.

Der nächste Paragraph hinter dem SCB wird von der ***READY-Liste*** belegt (siehe Seite 68), deren Länge direkt von der Anzahl der Prioritäten abhängt.

Hinter der READY-Liste stehen die ***Task Kontroll Blöcke*** (TCB). Jeder Task hat einen eigenen TCB, der alle notwendigen Daten des Task speichert (siehe Seite 66). Die minimale Länge eines TCB ist von UniMOS vorgegeben, sie kann jedoch vom Benutzer erweitert werden (siehe "UniMOS Publics und Externals" auf Seite 61).

An die TCBs schließen sich die Mailboxen an. Die Länge einer Mailbox ist abhängig vom verwendeten Speichermodell, da auch hier ein Zeiger verwendet wird.

Hinter den Mailboxen werden die Resourcen angelegt. Jede Resource is gleich lang, jedoch ist die Anzahl der Resourcen von der Konfiguration abhängig.

Am Ende des unimos-Speicherbereiches finden sich die Pool Kontrollblöcke (PCB), jeweils einer pro angelegtem Speicherpool.

Für die Größe des benötigten RAM-Bereiches läßt sich nun eine einfache Formel angeben:

G = scb + prio*2 + tasks*tcb + m*mbx + r*rsc + p*pcb

G : ....benötigter Speicherbereich in Bytes
scb:........Länge des SCB
prio: ....Anzahl der Prioritäten
tasks: .....Anzahl der Tasks
tcb:.........Länge eines TCB
m: ....Anzahl der Mailboxen
mbx: ......Länge einer Mailbox
r: ....Anzahl der Resourcen
rsc:.........Länge einer Resource
p: ....Anzahl der Speicherpools
pcb:........Länge eines PCB

Wichtiger Hinweis: Die Kontrollblöcke sind jeweils auf Paragraphengrenze ausgerichtet. Damit muß man zu jedem Term in der Gleichung noch das jeweilige "Alignment" hinzuaddieren. Das folgende Beispiel zeigt, wie man ein unimos-Segment der passenden Größe reserviert:

```
unimos segment para public
        db      (size scbs) dup(?)                  ;reserve space for SCB
        align   16
        db      n_prio*2 dup(?)                     ;reserve space for ready list
        align   16
        db      n_tasks*(size tcbs) dup(?)          ;reserve TCB space
        align   16
        db      n_mailboxes*(size mailboxs) dup(?)  ;mailbox space
        align   16
        db      n_resources*(size rscs) dup(?)      ;resource space
        align   16
        db      n_pool*(size pcbs) dup(?)           ;resource space
unimos ends
```

4.3. Die Konfigurationstabelle

Über die Einträge in der Konfigurationstabelle wird UniMOS zur Laufzeit initialisiert. Die Adresse der Konfigurationstabelle wird als Parameter zur Funktion *ini_tsk* (siehe Seite 34) angegeben. Die Konfigurationstabelle muß also vom Benutzer angelegt und initialisiert werden. Der folgende Abschnitt beschreibt die Einträge dieser Tabelle.

```
offset  conf_tab_s     struc
     0  ct_taskcount   dw      ?  ;maximum number of tasks
     2  ct_priocount   dw      ?  ;maximum number of priorities.
                                  ;Priorities go from 0 to ct_priocount-1,
                                  ;with 0 being the highest priority
     4  ct_mbxcount    dw      ?  ;maximum number of mailboxes.
     6  ct_rsccount    dw      ?  ;maximum number of resources.
     8  ct_poolcount   dw      ?  ;maximum number of memory pools
    10  ct_fatal       dd      ?  ;offset and segment of fatal error
                                  ;handler
    14  ct_disphook    dd      ?  ;offset and segment of dispatcher hook routine
        conf_tab_s     ends
```

4.3.1. Die Elemente der Konfigurationstabelle

In **ct_taskcount** muß vom Benutzer die maximale Anzahl von Tasks eingetragen werden. UniMOS reserviert für jeden Task einen TCB im unimos-Segment. Gibt man hier mehr Tasks an als wirklich notwendig, so kostet das zwar keine Rechenleistung, jedoch Speicherplatz.

ct_priocount gibt die maximale Anzahl unterschiedlicher Prioritäten an. Für jede Priorität wird ein Eintrag in der READY-Liste reserviert. Die maximale Anzahl der verschiedenen Prioritäten wurde auf 256 festgelegt. Der Speicherbedarf pro Priorität ist sehr gering (zwei Bytes), jedoch sollten nur soviele Prioritäten vergeben werden, wie auch wirklich gebraucht werden. Anderenfalls steigt die Suchzeit für den nächsten READY-Task unnötig an, falls unbenutzte Prioritäten vorhanden sind.

ct_mbxcount gibt die maximale Anzahl der Mailboxen an. Eine Mailbox belegt weder viel Speicher (vier oder sechs Bytes), noch wird durch die Anzahl der Mailboxen die Leistung des Systems beeinflußt. Dasselbe gilt für Resourcen, deren maximale Anzahl mit **ct_rsccount** festgelegt wird.

ct_poolcount gibt an, wieviele Speicherpools vom System verwaltet werden sollen. Für jeden Pool wird ein Pool Control Block (PCB) angelegt, der einige Bytes an Speicher kostet, jedoch keine Systemleistung.

ct_fatal spezifiziert eine Funktion, die dann aufgerufen wird, wenn ein fataler UniMOS Fehler auftritt. Diese Funktion wird z.B. dann benötigt, wenn der Benutzer versucht den Hintergrund-Task in einen anderen Zustand als ***RUNNING*** oder ***READY*** zu überführen. Dies mag zwar in der Testphase passieren, darf jedoch im fertigen System nicht auftreten. Es kann deshalb empfohlen werden, hier z.B. in die Kaltstart-Routine zu laufen.

ct_disphook ist die Adresse einer Hilfs-Funktion (oder NULL, falls diese Funktion nicht benötigt wird), die nach jeder Task-Umschaltung unmittelbar vor der Fortführung des nächsten Tasks aufgerufen wird. Diese Funktion kann benutzt werden, um weitere Register zu sichern oder wiederherzustellen, die nicht prozessorintern sind. Das sind beispielsweise die Register eines Fließpunktprozessors oder eines anderen externen Controllers. In den Beispielprogrammen wird diese Funktion benutzt um die Task-Zustände am Bildschirm darzustellen. Die Hilfs-Funktion erhält im AX-Register einen Zeiger auf den TCB des jetzigen Tasks und im DX-Register die Adresse des vorherigen Tasks. Damit kann auf die (möglicherweise auch erweiterten) Task Kontroll Blöcke zugegriffen werden. Alle Register außer SS und SP dürfen verändert werden. Es wird das Stack-frame des neuen Tasks benutzt. Die Rückkehr zum Dispatcher erfolgt mit einem Far Return, retf. Der Dispatcher holt dann nur noch die gesicherten Registerwerte vom Stack und führt dann den Task fort.

4.4. Der System Control Block (SCB)

Der SCB enthält die Basis-Informationen des Systems. Er ist der erste Block im unimos-Segment, dessen Adresse vom Benutzer angegeben wird.

```
offset  scbs            struc
     0  scb_taskcount   dw      ?    ;number of tasks
     2  scb_priocount   dw      ?    ;number of different priorities
     4  scb_mbxcount    dw      ?    ;number of mailboxes
     6  scb_rsccount    dw      ?    ;number of resources
     8  scb_poolcount   dw      ?    ;number of memory pools
    10  scb_ptcb        dw      ?    ;pointer to TCBs
    12  scb_pmbx        dw      ?    ;pointer to mailboxes
    14  scb_prsc        dw      ?    ;pointer to resources
    16  scb_ppcb        dw      ?    ;pointer to pool control blocks
    18  scb_ptcbcurr    dw      ?    ;pointer to current TCB
    20  scb_ptcbprev    dw      ?    ;pointer to previous TCB
    22  scb_readylist   dw      ?    ;pointer to ready list
    24  scb_firstwait   dw      ?    ;points to TCB of first task with
                                     ;running TIMER
    26  scb_preempt     dw      ?    ;0: preemption enabled, <>0: disabled
    28  scb_nextfree    dw      ?    ;next free paragraph after system
                                     ;blocks. Memory from this block
                                     ;onward is not used by UniMOS
    30  scb_fatal       dw      ?,?  ;offset and segment of handler for
                                     ;fatal errors
    34  scb_disphook    dw      ?,?  ;offset and segment of dispatcher
                                     ;hook handler
    38  scb_timint      db      ?    ;timer interrupt handler busy if <> 0
        scbs            ends
```

Die Zahl ganz links gibt den jeweiligen Abstand zum Anfang des Kontrollblockes an.

Über die Einträge im SCB sind alle anderen Kontrollblöcke erreichbar. Der SCB kann auch vom Benutzer verwendet werden, um Systeminformationen zu erhalten. Er darf jedoch nicht verändert werden! Insbesondere kann **scb_nextfree** benutzt werden, um den ersten freien Speicherblock hinter den Systemdatenblöcken festzustellen. Mitunter ist es auch wichtig festzustellen, welches die momentane Tasknummer ist. Sie ist errechenbar aus der Differenz des momentanen TCB Zeigers zum Anfang der TCBs dividiert durch die Länge eines TCB in Paragraphen (scb_ptcbcurr-scb_ptcb)/tcbsize. Dabei ist *tcbsize* die Länge eines TCB in Paragraphen, so wie sie vom Benutzerprogramm für UniMOS spezifiziert werden muß (siehe "UniMOS Publics und Externals" auf Seite 61).

Zum Berechnen der Tasknummer aus dem TCB-pointer stellt UniMOS die Funktion ***get_tn*** (siehe Seite 59) zur Verfügung.

4.4.1. Die Elemente des SCB

Die SCB-Elemente **scb_taskcount**, **scb_priocount**, **scb_mbxcount**, **scb_rsccount**, **scb_poolcount**, **scb_fatal** und **scb_disphook** werden beim Initialisieren des Systems (***ini_tsk***) mit den entsprechenden Werten aus der Konfigurationstabelle geladen. Zur ihrer Beschreibung sei daher auf das Kapitel "Die Konfigurationstabelle" auf Seite 64 verwiesen.

Die Elemente **scb_ptcb**, **scb_pmbx**, **scb_prsc**, **scb_ppcb** und **scb_readylist** werden während der Initialisierung jeweils mit Segment-Zeigern auf die Liste der TCBs, der Mailboxen, Resourcen, PCBs und der READY-Tasks geladen. Die Adresse eines bestimmten TCB errechnet sich dann aus scb_ptcb plus der Tasknummer multipliziert mit der Länge eines TCB. Die Länge eines TCB wird in Paragraphen angegeben und kann somit direkt zum Segment-Zeiger auf die TCBs addiert werden. Mit dem Offset 0 erhält man dann einen *far pointer*

auf den TCB. Die Adresse einer bestimmten Mailbox, Resource oder READY-Priorität wird etwas anders berechnet. Hier wird der Segmentwert aus scb_pmbx, scb_prsc oder scb_readylist beibehalten und der Offset zu dem jeweiligen Element errechnet. Er ergibt sich aus seiner Nummer multipliziert mit der Länge eines Elementes. Diese Art der Adressierung bedingt, daß alle Elemente in ein 64 kB großes Segment passen. Dadurch ist die Anzahl der Mailboxen auf 16384 (10922 bei *far* Datenzeigern) und die Anzahl der Resourcen auf 10922 (unabhängig vom Speichermodell) begrenzt. Die Anzahl der Prioritäten ist auf 32768 begrenzt (aufgrund von tcb_priority im TCB ist die maximal mögliche Anzahl der Prioritäten jedoch nur 256). Für alle praktisch vorkommenden Einsatzfälle liegen diese Werte weit oberhalb dessen was notwendig sein kann. Die Funktion ***ini_tsk*** kehrt mit einer Fehlermeldung zurück, falls dennoch versucht wird größere Werte anzugeben.

scb_ptcbcurr und **scb_ptcbprev** sind Segment-Zeiger auf den momentanen und den vorherigen TCB. Diese Werte werden vom Scheduler eingetragen und sind bereits gültig, wenn die "Hook-Routine" aufgerufen wird. Damit können dann weiter Parameter des gerade gestoppten Tasks gerettet und solche des gerade gestarteten Tasks wiederhergestellt werden. Außerdem kann scb_ptcbcurr jederzeit benutzt werden um auf den Kontrollblock des momentanen Tasks zuzugreifen oder auch um die Tasknummer zu errechnen.

scb_firstwait hält die TCB Adresse der ersten Task, der auf einen Timeout wartet. Hier wird 0 eingetragen, falls kein Task einen laufenden Timer hat. Weitere Tasks mit laufendem Timer werden durch die TCBs miteinander verkettet.

scb_preempt gibt an, ob die Taskumschaltung freigegeben ist oder nicht. Ist Taskumschaltung erlaubt, so ist dieser Wert 0, anderenfalls gibt er die Anzahl geschachtelter ***dis_pre***-Aufrufe an.

In **scb_nextfree** wird von der Initialisierungsfunktion ***ini_tsk*** diejenige Segment-Adresse eingetragen, ab der keine UniMOS Speicherbereiche mehr angelegt sind. RAM-Bereiche oberhalb dieser Adresse können anderweitig frei benutzt werden.

scb_timint wird ungleich null gesetzt, wenn gerade der Timer-Interrupt Handler ausgeführt wird. Damit wird ein geschachtelter Aufruf dieser Funktion vermieden.

4.5. Der Task Control Block (TCB)

```
    tcbs            struc
 0  tcb_status      db     ?  ;task status
                              ;four lower bits give the status
                              ;four upper bits give reason for status
 1  tcb_priority    db     ?  ;prio of this task: 0=highest, 255=lowest
 2  tcb_prev        dw     ?  ;for linked lists: previous TCB
 4  tcb_next        dw     ?  ;for linked lists: next TCB
 6  tcb_prevtimer   dw     ?  ;previous TCB with timer
 8  tcb_nexttimer   dw     ?  ;next TCB with timer
10  tcb_stackptr    dw     ?  ;stack pointer (SP)
12  tcb_stackseg    dw     ?  ;stack segment (SS)
14  tcb_timer       dw     ?  ;timer value for timeouts
16  tcb_events      dw     ?  ;stores up to 16 events
                    union
18  tcb_rscoffset   dw     ?  ;offset to the resource that this task is
                              ;waiting for (offset within scb_prsc).
18  tcb_mbxoffset   dw     ?  ;offset to the mailbox that this task is
                              ;waiting for (offset within scb_pmbx).
                    ends
                    union
20  tcb_rscprio     db     ?  ;the task is waiting with this priority
                              ;for a resource
20  tcb_error       dw     ?  ;extended error code
                    ends
22  tcb_timeslice   db     ?  ;duration of a time slice in timer ticks.
23  tcb_remaining   db     ?  ;remaining timer ticks.
    tcbs            ends
```

Die Zahl ganz links gibt den jeweiligen Abstand zum Anfang des Kontrollblockes an.

Der Task Kontroll Block enthält alle für einen Task relevanten Informationen. Seine Länge wird beim Initialisieren des Systems festgelegt und sie ist für jeden Task gleich. Die Länge ist immer ein ganzzahliges Vielfaches eines Paragraphen. Dabei legt die hier gezeigte Struktur die Mindestlänge fest. Der Anwender kann einen erweiterten TCB spezifizieren, der dann für eigene Zwecke benutzt wird (siehe auch "**Externe Variablen**" ab Seite 61).

4.5.1. Die Elemente des TCB

Der momentane Status eines Task wird in **tcb_status** gespeichert. Dieses Byte besteht aus zwei Bitfeldern. Die vier niederwertigsten Bits geben den Status an, während die vier höherwertigen Bits diesen Status näher spezifizieren. Das ist insbesondere für einen wartenden Task von Bedeutung, da hier die genaue Ursache des Wartens angezeigt wird. Dabei sind folgende Werte für den Task Status möglich:

```
state_dormant   equ  0   ;This task is in the "DORMANT" state
state_suspend   equ  1   ;This task is in the "SUSPEND" state
state_wait      equ  2   ;This task is in the "WAIT" state
state_ready     equ  3   ;This task is in the "READY" state
state_run       equ  4   ;This task is in the "RUN" state
```

WAIT- und SUSPEND-Zustände sind durch die folgenden Codes in den höherwertigen vier Bit des Status näher spezifiziert:

```
wait_for_signal  equ  10h  ;wait for signal from interrupt handler
wait_for_event   equ  20h  ;wait for events
wait_for_timer   equ  30h  ;wait for timer
wait_for_mail    equ  40h  ;wait for mail
wait_for_rsc     equ  50h  ;wait for a resource
```

tcb_priority gibt die Priorität des Tasks an. Sie kann zwischen 0 und 255 liegen, wobei 0 die höchste und 255 die niedrigste Priorität ist.

tcb_prev und **tcb_next** zeigen auf einen vorherigen und einen nächsten TCB falls TCBs verkettet werden müssen. Besteht die Liste aus nur diesem einen TCB, so zeigen beide Zeiger auf diesen TCB. Verkettete TCBs werden z.B. beim Warten auf Nachrichten für eine Mailbox oder beim Warten auf eine Resource benötigt. Wenn mehrere Tasks mit einem Timeout auf ein Ereignis warten, so sind die TCBs über die Zeiger **tcb_prevtimer** und **tcb_nexttimer** verkettet. Somit können also mehrere Tasks z.B. auf Nachrichten warten und gleichzeitig auch auf einen Timeout. In **tcb_timer** wird der timeout-Wert gespeichert, der dann bei jedem Timer-Interrupt dekrementiert wird. Beim Erreichen von null wird der Task wieder in den READY-Zustand gesetzt.

Wenn ein Task vom Scheduler aus dem RUN- in den READY-Zustand überführt wird, dann wird sein gesamter Status (alle Registerinhalte des Prozessors) auf dem Stack gespeichert und der Stackpointer und das Stacksegment werden im TCB in **tcb_stackptr** und **tcb_stackseg** gesichert. Dieselben Elemente des neuen TCB werden benutzt um Stackpointer und Stacksegment des neuen Tasks zu laden und von diesem Stack den neuen Taskstatus wiederherzustellen. Die Register werden in der folgenden Reihenfolge auf dem Stack gesichert:

CS, IP, PSW, AX, CX, DX, BX, SP, BP, SI, DI, DS, ES

Da der Stack nach unten wächst, findet sich CS an der höheren Speicheradresse und ES an der niedrigeren. **tcb_stackseg:tcb_stackptr** zeigen auf den zuletzt gesicherten Wert, also auf ES.

tcb_events ist ein Feld von 16 Bit, von denen jedes einem Ereignis zugeordnet werden kann. Der Task kann auf eines oder mehrere dieser Ereignisse warten. Er wird erst dann weitergeführt, wenn alle Ereignisse eingetreten sind, also alle Bits gesetzt sind.

tcb_mbxoffset und **tcb_rscoffset** geben diejenige Mailbox oder Resource an, auf die dieser Task wartet. Sie spezifizieren einen Offset innerhalb der Mailboxen beziehungsweise der Resourcen. Dieses Feld wird doppelt benutzt, da ein Task nicht gleichzeitig auf eine Nachricht und auf eine Resource warten kann. UniMOS benötigt diese Informationen um einen Task bei abgelaufenem timeout aus der Warteschlange herauszunehmen.

Wenn ein Task auf eine Resource wartet, so kann man dafür eine Priorität angeben. Damit kann er gegenüber anderen Tasks bevorzugt werden, die die Resource nicht so dringend benötigen. Die Priorität, mit der ein Task die Resource anfordert, wird in **tcb_rscprio** gespeichert. Auf eine Resource wartende Tasks werden in der Reihenfolge ihrer Prioritäten miteinander verkettet. Einige Systemfunktionen können mit einem erweiterten Fehlercode beendet werden, der nicht direkt zurückgegeben werden kann. Das ist insbesondere bei solchen Funktionen der Fall, die normalerweise einen Zeiger zurückgeben. Im Fehlerfall geben sie dann einen NULL-Zeiger zurück und speichern den Fehlercode in **tcb_error**. Dieser Wert kann dann von der aufrufenden Funktion analysiert werden. Da während des Wartens auf eine Resource kein Fehlercode benötigt wird, belegen tcb_rscprio und tcb_error denselben Speicherbereich.

In **tcb_timeslice** wird die beim Aufruf von ***sta_tsk*** angegebene maximale Dauer einer Zeitscheibe abgespeichert. Wenn der Scheduler den Task zur Ausführung auswählt, dann wird dieser Wert nach **tcb_remaining** kopiert. Bei jedem Timer-Interrupt wird tcb_remaining dekrementiert und wenn null erreicht wird, wenn also die Zeitscheibe abgelaufen ist, dann wird der nächste Task gleicher Priorität ausgewählt.

4.6. Die READY-Liste

Die READY-Liste ist auch ein Kontrollblock, für den jedoch aufgrund seiner Einfachheit kein STRUC definiert ist. Es handelt sich um ein eindimensionales Feld, dessen Einträge jeweils ein Wort (zwei Byte) lang sind. Die Länge der Liste ist durch die Anzahl der Prioritäten gegeben, da für jede Priorität ein Eintrag vorgesehen ist. Der erste Eintrag gilt für die Priorität null, der zweite für eins usw. Wenn kein Task mit der entsprechenden Priorität ***READY*** ist, dann wird in der READY-Liste ein NULL-Zeiger eingetragen. Sonst enthält sie den Zeiger auf denjenigen TCB, der entweder ***RUNNING*** ist, oder als nächster Task ausgewählt werden soll. Die READY-Liste wird von den Systemfunktionen gelesen und modifiziert. Ein Benutzerprogramm braucht sie normalerweise nicht zu lesen und sollte sie auf keinen Fall ändern.

4.7. Der Nachrichtenblock

UniMOS gestattet die Übertragung von Nachrichten zwischen verschiedenen Tasks. Als Hilfsmittel für die Übertragung werden dabei Mailboxen (Briefkästen) verwendet, in die man eine Nachricht ablegen und sie auch wieder von dort abholen kann. Sender und Empfänger sind dabei in der Regel verschiedene Tasks. Bei der Initialisierung des Systems wird die maximal gewünschte Anzahl Mailboxen angegeben. Mailboxen werden über ihre Nummer angesprochen, die von 0 bis zur maximalen Anzahl minus 1 reicht (also z.B. 0 .. 9 bei insgesamt zehn Mailboxen). Für jede Mailbox wird eine Struktur der folgenden Form angelegt:

```
mailboxs        struc
mailbox_first   dw/dd   ?  ;pointer to first mail or zero
mailbox_tcb     dw      ?  ;pointer to TCB of first task
                           ;waiting for a message
mailboxs        ends
```

Die Struktur enthält also lediglich einen Zeiger (*near* oder *far*) auf die erste Nachricht (oder NULL, falls keine Nachricht vorhanden ist), sowie einen Segment-Zeiger auf den TCB des ersten Tasks, der auf eine Nachricht für diese Mailbox wartet. Auch der TCB-Eintrag kann NULL sein, nämlich dann, wenn kein Task auf eine Nachricht wartet.

Alle Nachrichten haben prinzipiell die gleiche Struktur, wenn auch ihre Länge unterschiedlich sein kann. Der allen Nachrichten gemeinsame Anfangsteil ist hier dargestellt:

```
  mails          struc
0 mail_function  db     ? ;function code
1 mail_prio      db     ? ;priority: 0=highest, 255=lowest
2 mail_source    dw     ? ;task number of originating task
4 mail_next      dw     ? ;pointer to next mail or zero
6 mail_prev      dw     ? ;pointer to previous mail or zero
8 mail_data      db     ? ;data area (immediate data or pointer)
  mails          ends
```

Die Zahl ganz links gibt den jeweiligen Abstand zum Anfang des Kontrollblockes an.

mail_function ist der Teil, der dem Empfänger der Nachricht mitteilt, was er mit dieser Mitteilung machen soll. Die Benutzung dieses Feldes ist dem Anwender vollkommen freigestellt. **mail_data** ist der ebenfalls benutzerspezifische Datenteil der Nachricht. Hier können z.B. direkt Daten stehen oder aber auch ein Zeiger auf einen Datenblock. Da die Länge einer Nachricht nicht von vorneherein festgelegt ist, kann hier auch beispielsweise eine Zeichenkette abgespeichert sein, die von einem anderen Task auf einem Display dargestellt werden soll. In diesem Fall könnte dann mail_function die Darstellungsart (Farbe, invertiert, blinkend, mit Piepston u.s.w.) angeben.

mail_next und **mail_prev** werden von UniMOS zum Verketten der Nachrichten benötigt. Diese Zeiger (*near* oder *far*) dürfen nicht von anderer Stelle verändert werden. UniMOS trägt außerdem in **mail_prio** die Priorität dieser Nachricht ein und stellt dem Empfänger in **mail_source** die Task-Nummer des Absenders zur Verfügung. Aufgrund dieser Variablen kann man keine Nachricht verschicken, die in einem Nur-Lese-Speicher (ROM, EPROM) steht. Solche Nachrichten müssen dann mit Zeigern vermittelt werden.

4.8. Der Resourcen-Kontrollblock

Resourcen werden benötigt um den Zugriff auf Objekte zu koordinieren, die nur in begrenzter Anzahl verfügbar sind. Jede dieser Resourcen kann eines oder mehrere Elemente vom gleichen Typ verwalten. Die Anzahl der benötigten Resourcen wird beim Initialisieren des Systems in der Konfigurationstabelle angegeben. Für jede Resource wird ein Kontrollblock mit der folgenden Struktur angelegt:

```
  rscs          struc
0 rsc_count     dw     ? ;number of resources left
2 rsc_maxcount  dw     ? ;maximum number of resources
4 rsc_tcb       dw     ? ;root pointer to TCBs waiting for
                         ;this resource
  rscs          ends
```

Die Zahl ganz links gibt den jeweiligen Abstand zum Anfang des Kontrollblockes an.

rsc_count gibt die Anzahl noch verfügbarer Elemente dieser Resource an. Die gesamte Anzahl steht in **rsc_maxcount**. Das ist der Wert, der beim Aufruf von ***ini_rsc*** angegeben wurde. Wenn ein Task durch ***req_rsc*** ein Element anfordert ohne daß noch eines verfügbar ist, so wird er wartend gesetzt und die Adresse seines TCB wird in **rsc_tcb** eingetragen. Mehrere wartende Tasks können in der Reihenfolge ihrer Prioritäten verkettet werden.

4.9. Der Memory-Pool Kontrollblock

Für jeden Speicherpool legt UniMOS beim Initialisieren mit ***ini_tsk*** einen Pool Kontrollblock (PCB) an. Er hat normalerweise folgenden Aufbau:

```
offs  pcbs           struc
   0  pcb_next       dw    ?,?  ;points to the next unused record
   4  pcb_first      dw    ?,?  ;points to the first record
   8  pcb_last       dw    ?,?  ;points to the last record
  12  pcb_size       dw    ?    ;number of bytes per record (0 = 64 kB)
  14  pcb_count      dw    ?    ;number of records
  16  pcb_type       db    ?    ;memory type of this pool
  17  pcb_reserve    db    ?    ;reserved
      pcbs           ends
```

In der Debug-Version von UniMOS (also z.B. UM86**D**S) wird im Memory-Pool Kontrollblock ein weiterer Eintrag definiert, so daß dann die folgende Struktur gilt:

```
offs  pcbs           struc
   0  pcb_next       dw    ?,?  ;points to the next unused record
   4  pcb_first      dw    ?,?  ;points to the first record
   8  pcb_last       dw    ?,?  ;points to the last record
  12  pcb_size       dw    ?    ;number of bytes per record (0 = 64 kB)
  14  pcb_count      dw    ?    ;number of records
  16  pcb_mincount   dw    ?    ;minimum number of records
  18  pcb_type       db    ?    ;memory type of this pool
  19  pcb_reserve    db    ?    ;reserved
      pcbs           ends
```

Alle Blöcke in einem Pool sind miteinander verkettet. **pcb_next** zeigt immer zum nächsten freien Block. Dieser Zeiger wird beim nächsten ***get_mem*** Aufruf zurückgegeben. Der Zeiger zum nächsten freien Block, der in dem dann zurückgegebenen Block gefunden wurde, wird anschließend hier eingetragen.

pcb_first und **pcb_last** zeigen zum physikalisch ersten bzw. letzten Block dieses Speicherpools. Diese Einträge werden u.a. benutzt, um die Zugehörigkeit eines Blockes zu einem Pool zu bestimmen.

pcb_size und **pcb_count** geben die Größe eines Blockes und die Anzahl der noch verbleibenden Blöcke in diesem Pool an. **pcb_count** wird also bei jedem ***get_mem*** und ***rel_mem*** Aufruf angepasst.

pcb_mincount ist nur in der Debug-Version vorhanden. Hier wird die minimal verfügbare Anzahl Speicherblöcke gespeichert.

pcb_type gibt den Typ dieses Speicherpools an. Es werden z.Zt zwei Typen unterstützt, nämlich ein near-Pool (**pcb_type**=1) und ein far-Pool (**pcb_type**=2).

pcb_reserve stellt lediglich sicher, daß der nächste PCB auf Wortgrenze ausgerichtet ist.

4.10. Fehlercodes

Die UniMOS Funktionen können Fehlercodes zurückgeben. Dieser Code ist immer ein Wort (zwei Bytes) und er wird im AX-Register übergeben. Dies entspricht der C-Konvention. Die Bedeutung der Codes ist im folgenden erklärt.

0 - kein Fehler

Die Funktion wurde fehlerfrei ausgeführt.

1 - ungültige Task-Nummer

Es wurde eine Task-Nummer spezifiziert, die außerhalb des gültigen Bereiches liegt. Bei der Initialisierung des Systems mit der Funktion ***ini_tsk*** wird eine maximale Anzahl von Tasks (*ct_maxtask*) in der Konfigurationstabelle angegeben. Die Task-Nummern können damit zwischen 0 und *ct_maxtask*-1 (einschließlich) liegen.

2 - Task ist nicht in einem für diese Funktion gültigen Zustand

Um einige Systemfunktionen auszuführen, muß der Task einen dafür zulässigen Zustand haben. Die zulässigen Zustände und die erlaubten Übergänge sind im Zustandsdiagramm, Bild 1 auf Seite 7 dargestellt. Damit ein Task z.B. über die Funktion ***sta_tsk*** gestartet werden kann, muß er zwangsläufig vorher im Zustand ***DORMANT*** sein.

3 - WAIT erzwungenermaßen beendet

Wenn ein wartender Task mit ***rsm_tsk*** vorzeitig von ***WAITING*** nach ***READY*** überführt wird, dann wird dieser Fehlercode zurückgegeben. Der Task der ***WAITING*** war, kann damit die Ursachen für sein Weiterlaufen unterscheiden.

4 - Ungültige Priorität

Es wurde eine Priorität gewählt, die außerhalb des zulässigen Bereiches liegt. Die maximal mögliche Priorität wird beim initialisieren des Systems mit ***ini_sys*** angegeben.

5 - Taskumschaltung ist verhindert

Es wurde eine Funktion aufgerufen, die den momentanen Task anhalten würde, obwohl der Taskwechsel ausgeschaltet ist. Beispiel: Aufruf von ***dis_pre*** und anschließend ***exi_tsk***.

6 - Wait durch Timer beendet

Das Warten auf ein Ereignis wurde durch einen Timeout beendet ohne daß das Ereignis eingetreten ist.

7 - Falsche Mailbox Nummer

Es wurde eine ungültige Mailbox-Nummer angegeben. Gültige Nummern gehen von 0 bis zu ct_mbxcount-1 (siehe "Die Konfigurationstabelle" auf Seite 64).

8 - Ungültige Nachricht

Es wurde ein NULL-Zeiger als Zeiger zu einer Nachricht angegeben.

9 - Ungültige Resource

Es wurde eine ungültige Resourcen-Nummer angegeben. Gültige Nummern gehen von 0 bis zu ct_rsccount-1 (siehe "Die Konfigurationstabelle" auf Seite 64).

10 - Keine Resourcen mehr verfügbar

Alle vorhandenen Elemente dieser Resource sind momentan belegt.

11 - Zu viele Resourcen freigegeben

Es wurden mehr Elemente einer Resource freigegeben als überhaupt verfügbar waren. Dies ist ein Fehler im Anwendungsprogramm. ***rel_rsc*** darf höchstens so oft aufgerufen werden, wie ***pol_rsc*** (ohne Fehlercode beendet) und ***req_rsc*** zusammen.

12 - Zu viele Mailboxen definiert

Alle Mailboxen müssen in ein 64 kB großes Segment passen. Wenn in der Konfigurationstabelle eine zu große Anzahl Mailboxen angegeben wird, so gibt ***ini_tsk*** diese Fehlermeldung zurück.

13 - Zu viele Resourcen definiert

Alle Resourcen müssen in ein 64 kB großes Segment passen. Wenn in der Konfigurationstabelle eine zu große Anzahl Resourcen angegeben wird, so gibt ***ini_tsk*** diese Fehlermeldung zurück.

14 - Zu viele Speicherpools definiert

Alle PCBs müssen in ein 64 kB großes Segment passen. Wenn in der Konfigurationstabelle eine zu große Anzahl Speicherpools angegeben wird, so gibt ***ini_tsk*** diese Fehlermeldung zurück.

15 - Zu viele Blöcke in diesem Pool

Es wurde eine ungültige Block-Anzahl angegeben. Die Anzahl muß ungleich null sein und die Gesamtlänge des Pools (Blockgröße mal Anzahl) darf bei einem near-Pool nicht größer als 64 kB, bei einem far-Pool nicht größer als 1 MB sein.

16 - Ungültiger Speicherpool

Beim Aufruf der Funktion wurde eine Pool-Nummer angegeben, die größer war, als die bei der Konfiguration definierte Anzahl der Pools minus eins.

17 - Ungültige Blockgröße

Die Größe eines Speicherblockes muß mindestens vier Bytes betragen, andernfalls liefert ***ini_mem*** diesen Fehlercode zurück.

18 - Ungültige Pool-Typ

Normalerweise müssen je nach Pool-Typ (near oder far) die entsprechenden Speichermanagement-Funktionen benutzt werden. Versucht man z.B. mit ***get_nmem*** einen Block aus einem far-Pool zu erhalten, so wird diese Fehlermeldung zurückgegeben.

19 - Ungültige Blockadresse

Wird beim Freigeben eines Speicherblockes mit ***rel_mem*** eine Adresse angegeben, die zu keinem der Speicherpools gehört, dann wird dieser Fehlercode zurückgegeben.

20 - Keine weiteren Speicherblöcke verfügbar

Die Funktion get_mem hat versucht einen Speicherblock zu erhalten. Es ist jedoch kein freier Block mehr vorhanden.

4.11. Leistungsdaten

Dieses Kapitel beschreibt die Möglichkeiten und die Leistungsgrenzen des UniMOS Echtzeit-Kernes. Es werden zunächst nocheinmal die Systemanforderungen aufgelistet, so wie sie bereits in den vorherigen Kapiteln dieses Handbuches beschrieben wurden. Dabei ist angenommen, daß maximal 1 MB Systemspeicher zur Verfügung stehen. Einige Prozessoren (V53, V55, 80286) können aufgrund spezieller Zugriffsmechanismen mehr Speicher adressieren, was jedoch nicht unterstützt wird. Eine Tabelle zeigt abschließend die Ausführungszeiten, die Interrupt-Sperrzeiten und die Größe der einzelnen UniMOS-Unterprogramme.

Wieviele Tasks sind möglich?

Die Länge eines TCB und der für jeden Task notwendige Stackbereich begrenzt letztendlich die maximale Anzahl Tasks. Wenn man sich mit dem Stackbereich einschränkt, so kommt man vielleicht mit 96 Bytes aus. Das sind für Stack und TCB zusammen 128 Bytes pro Task (bei 32 Bytes pro TCB). Damit liegt die maximal mögliche Anzahl Tasks bei etwa 8000. Praktisch wird diese Anzahl niedriger sein, da auch Speicher für die Programme sowie weitere Daten und UniMOS Kontrollblöcke benötigt wird. Eine große Anzahl von Tasks kostet also Speicherplatz im RAM. Solange die Tasks im Zustand DORMANT, SUSPENDED oder WAITING sind, beeinträchtigen sie aber nicht die Leistung des Systems, es sei denn es wurde ein timeout angegeben. Dann muß bei jedem Timer-Interrupt der Timer im TCB dekrementiert werden.

Wieviele Prioritäten sind möglich?

Die Anzahl unterschiedlicher Prioritäten wird durch den Datentyp **tcb_priority** im TCB begrenzt. Dort ist momentan (aus historischen Gründen) ein Byte vorgesehen, was 256 verschiedene Prioritäten erlaubt. Dieser Datentyp könnte zukünftig in ein Wort umgewandelt werden, wodurch die Anzahl der Prioritäten auf 32768 ansteigen würde. Dennoch wird die jetzige Obergrenze von 256 Prioritäten nicht als echte Beschränkung angesehen. Für normale Anwendungen reichen zehn unterschiedliche Prioritäten vollkommen aus. Da jede Priorität in der READY-Liste nur ein Wort belegt, kostet eine Priorität praktisch keinen Speicherplatz. Unbenutzte Prioritäten kosten jedoch Systemleistung, da bei einem Taskwechsel mitunter diese READY-Liste durchsucht werden muß. Das ist normalerweise kein Problem, denn diese Suche ist relativ schnell, da immer nur getestet werden muß, ob der Eintrag ungleich null ist. Durch Verwendung intelligenter Algorithmen braucht diese Liste außerdem nur selten ganz durchsucht zu werden. Oft genügt die Suche ab der momentanen Priorität abwärts, da ein höherpriorisierter Task als der momentane nicht READY sein kann (denn sonst hätte er ja den Prozessor). Bei Verwendung vieler ungenutzter Prioritäten kann es jedoch zu pathologischen Fällen kommen, die extrem lange brauchen. Das ist z.B. dann der Fall, wenn ein Task mit der Priorität 255 die Taskumschaltung wieder freigibt (***ena_pre***) und kein anderer Task mit höherer Priorität READY ist. Dann muß ab Priorität 0 die gesamte READY-Liste durchsucht werden. Das ist sicher kein großer Schaden, denn der Task mit der niedrigsten Priorität hat wohl keine Aufgabe, die nicht etwas warten könnte. Außerdem dürfte dieser Fall in der Praxis leicht vermeidbar sein. Er soll jedoch verdeutlichen, daß dem Programmierer durchaus die Aufgabe des gewissenhaften Programmierens zufällt.

Wieviele Mailboxen sind möglich?

Die Anzahl der Mailboxen ist dadurch beschränkt, daß alle Mailboxen in ein 64 kB großes Segment passen müssen. Eine Mailbox ist vier oder sechs Bytes lang, je nachdem ob *near* oder *far* Datenzeiger verwendet werden. Damit liegt die Maximalzahl weit über den normalerweise notwendigen fünf bis zehn Mailboxen. Unbenutzte Mailboxen kosten keine Systemleistung und praktisch keinen Speicherplatz.

Wieviele Resourcen sind möglich?

Auch für die Anzahl der Resourcen gilt das für Mailboxen gesagte. Eine Resource ist immer sechs Bytes lang, so daß theoretisch über 10000 Resourcen möglich sind (und jede davon kann bis zu 32767 Elemente verwalten).

Wieviele Tasks können mit einem Timeout warten?

Die Anzahl der Tasks, die mit laufendem Timer warten, ist nicht von vorneherein beschränkt, da in einer verketteten Liste beliebig viele Elemente vorhanden sein können. Jeder weitere Timer kostet jedoch Systemleistung, da er bei jedem Timer-Tick heruntergezählt werden muß. Wenn der Timer-Interrupt z.B. jede Millisekunde auftritt, dann kann es bei mittlerer Prozessorleistung (8088, 80188, V25, V40) und hundert Tasks, die mit Timeout warten, schon zu knapp werden, um vor dem nächsten Timer-Interrupt fertig zu sein. Wenn außerdem gerade der Timer eines Tasks abgelaufen ist und deshalb der Task in den READY-Zustand gesetzt werden muß, so wird zusätzliche Prozessorzeit benötigt. Passiert das außerdem auch noch bei allen hundert Tasks gleichzeitig, so reicht die Zeit zwischen zwei Timer-Interrupts sicher nicht mehr aus. Ein interner Mechanismus verhindert eine Katastrophe, indem ein Timer-Interrupt nicht behandelt wird, bevor der vorherige abgeschlossen ist. Dadurch können dann jedoch Timer-Interrupts verloren gehen. Die Interrupt-Sperrzeiten stellen kein Problem dar, da während der Ausführung der Timer-Interrupt-Routine (***tim_int***) regelmäßig die Interrupts freigegeben werden.

Wie bei der Vergabe der Prioritäten muß also auch hier versucht werden, gewisse ungünstige Konstellationen zu vermeiden. Hat das System nur wenige Tasks (bis etwa 32), so darf sicherlich jede Millisekunde ein Timer-Interrupt auftreten und die Tasks dürfen auch alle gleichzeitig mit Timeout warten. Bei größeren Systemen muß man jedoch andere Wege gehen. Zum Beispiel sollte überlegt werden, ob man die Timer-Interrupts nur alle 10 oder 100 Millisekunden generiert. Das ist oft ausreichend. Kommt es wirklich vor, daß soviele Tasks gleichzeitig mit Timeout warten müssen? Wenn das nicht der Fall ist, dann können diese Überlegungen getrost ignoriert werden. Wenn alles nichts hilft, dann muß auch die Verwendung eines Prozessors einer höheren Leistungsklasse in Betracht gezogen werden (80286, 80386, V33, V53, V55).

Welches Zeitverhalten gilt beim Verschicken von Nachrichten mit Prioritäten und beim Warten auf Resourcen mit Prioritäten?

Bei ***snd_msg*** und ***get_rsc*** werden jeweils Warteschlangen unterstützt, in denen jeder Eintrag eine Priorität hat. Es handelt sich dabei um eine eindimensionale Liste, d.h. es existiert ein Zeiger auf das erste Element und jedes Element hat einen Zeiger auf das nächste und das vorherige Element. Damit kann man sich innerhalb dieser Liste vorwärts und rückwärts bewegen. Beim Einketten eines neuen Elementes in die Liste, prüft die entsprechende Systemfunktion zunächst, ob die Priorität gleich oder niedriger als die Priorität des letzten Listenelementes ist. Wenn ja, dann wird das neue Element hinten angehängt. Ist das nicht der Fall, dann wird geprüft, ob die Priorität größer ist als die des ersten Elementes der Liste. Dann braucht das neue Element nur am Anfang der Liste eingefügt zu werden. Liegt die Priorität des neuen Elementes zwischen der des ersten und der des letzten Elementes, dann muß die Liste linear von vorne bis zur passenden Position durchsucht werden. Das ist ein Vorgang, der Rechenzeit benötigt und wiederum zu extrem ungünstigen Fällen führen kann, die aber in der Regel vermeidbar sein dürften.

Generell kann man empfehlen, eine niedrige Priorität für den Normalfall zu verwenden und nur für die wirklich dringenden Fälle eine höhere Priorität zu benutzen. In aller Regel werden die dringenden Fälle weitaus seltener eintreten als die Normalfälle. Damit müssen dann beim Einketten eines hochpriorisierten Elementes nur wenige höher oder gleichhoch priorisierte andere Elemente übersprungen werden.

Leistungsmessungen

Hier sind die Ergebnisse der Performance-Messungen aufgeführt. Die Meßergebnisse wurden mit einem V25-MINI-IE-P (Fa. AdTec oder Fa. NEC) mit normalem V25 Prozessor erzielt. Die interne Taktfrequenz betrug dabei 8 MHz. Es wurde das interne RAM (Registerbänke) disabled, keine Wartezyklen eingefügt und kein Refresh zugelassen. Es wurde die UniMOS Bibliothek für das SMALL Memory Modell verwendet und es wurden die Instruktionen des 80188 zugelassen. Die folgende Tabelle listet die Ausführungszeiten sowie die Interrupt-disable Zeiten einschließlich dem Speichern der Parameter auf dem Stack jeweils in Mikrosekunden.

Es ist zu beachten, daß die Ausführungszeiten etwas von der Systemkonfiguration abhängen. Insbesondere muß bei der Taskumschaltung jeweils die Ready-Liste nach dem nächten READY-Task durchsucht werden. Jede ungenutzte Priorität kostet dabei einige Mikrosekunden Ausführungszeit und zwar normalerweise bei gesperrten Interrupts. Bei den hier gezeigten Meßergebnissen ist beim Umschalten von Task A mit Priorität n jeweils Task B mit Priorität n+1 READY. Task A wird aus dem *RUNNING* Zustand genommen, während Task B nach *RUNNING* überführt wird. Bei einigen Systemfunktionen kann nicht davon ausgegangen werden, daß der mo-

mentane Task der am höchsten priorisierte Task ist, der den Prozessor benötigt. In diesem Fall muß die gesamte READY-Liste ausgehend von der Priorität null durchsucht werden. Der nächste Task wird dabei umso schneller gefunden, je höher seine Priorität ist (0 = höchste Priorität).

Funktion	ohne TW	mit TW	disable	Länge
def_int				35
dis_pre	13,5	-		15
ena_pre	22,5	80 *1		70
exi_tsk	-	158		97
get_int				23
get_mem				198-200
ini_evt				25-26
ini_mem				412-425
ini_rsc				50-51
ini_tsk	493 *2	-		453-463
pol_evt				23-24
pol_msg				62-71
pol_rsc				57-58
rcv_msg				230-257
rel_mem				214-216
rel_rsc				150
req_rsc				314
rsm_tsk		135		259-262
sig_evt				142
sig_int				152
snd_msg				188-199
sta_tsk	125	171		233-237
sus_tsk	-	145		170
tim_int				342-347
trm_slc				74
trm_tsk	56 *3			257
wai_evt				150
wai_int	-	143		159

Anmerkungen:

*1 Pro Priorität, für die kein Task READY ist, kommen etwa 5,5 µsek dazu. Wenn also kein Task mit Priorität kleiner 10 READY ist, so ist die Ausführungszeit etwa 135 µsek.

*2 Die Ausführungszeit ist sehr Parameterabhängig. Hier: 64 Tasks, 10 Prioritäten, 5 Mailboxen.

*3 Abhängig vom Status des terminierten Tasks. Hier: Terminieren eines suspendierten Tasks ohne Timer.

Alle Werte wurden ohne Dispatcher-Hook Routine gemessen, deren Laufzeit sich bei einem Taskwechsel hinzuaddiert.

Die Spalte "o TW" gibt die Ausführungszeit der Funktion an, wenn kein Taskwechsel notwendig ist. "m TW" hingegen berücksichtigt eine notwendige Taskumschaltung. Die Spalte "disable" gibt die Maximalzeit an, während der die Interrupts gesperrt sind.

Die Länge der Module ist in der letzten Spalte in Bytes angegeben. Da die Länge vom Speichermodell und vom verwendeten Prozessor (8088/80188) abhängen kann, ist hier jeweils eine minimale und eine maximale Länge angegeben. Die anderen Code-Längen liegen dazwischen. Es ist zu beachten, daß einige Module weitere Module dazulinken, wodurch die effektive Länge etwas erhöht werden kann. Ein Beispiel ist der Dispatcher, der zwischen 87 und 96 Bytes lang ist und fast von jedem Modul benötigt wird. Er wird dann allerdings auch nur

einmal dazugebunden und kann so von jedem Modul benutzt werden. Wenn alle Module zusammengelinkt werden, so benötigt UniMOS etwa 5 kB Codespeicher.

Die in dieser Tabelle genannten Werte sind nur als ungefähre Werte zu verstehen. Selbst geringste Modifikationen an den Modulen werden die Ausführungszeit und die Größe der Module beeinflussen. Außerdem variieren die Ausführungszeiten mit der Konfiguration, wie bereits vorher in diesem Kapitel erläutert wurde.

Stichwortverzeichnis

Datentypen 22
deadlock 49
Debugfunktionen 62
  get_inf 62
Debug-Version 72
DGROUP 22, 52
Diskettenformate 6
Dispatcher 7, 36
Dispatcher-Hook 63
disphook 66
DORMANT 7
Entwicklungswerkzeuge 9
Ereignis 12, 14
ereignisgesteuert 5
erweiterter TCB 63
event 12, 14
FAR-pointer 11
Fehlercodes 72
Funktionen
  def_int 23
  dis_pre 25
  ena_pre 27
  exi_tsk 28
  get_fmem 30
  get_int 29
  get_mem 30
  get_nmem 30
  ini_evt 31
  ini_fmem 33
  ini_mem 33
  ini_nmem 33
  ini_rsc 35
  ini_tsk 36
  pol_evt 38
  pol_msg 39
  pol_rsc 41
  rcv_msg 42
  rel_fmem 43
  rel_mem 43
  rel_nmem 43
  rel_rsc 44
  req_rsc 45
  rsm_tsk 46
  sig_evt 48
  sig_int 49
  snd_msg 51
  sta_tsk 52
  sus_tsk 54
  tim_int 55
  trm_slc 57
  trm_tsk 58
  wai_evt 59
  wai_int 60
Hilfsfunktionen
  get_tn 61
Hintergrund-Task 8, 36
Idle Code 9
initial PSW 63
Initialisierung 22
Interrupt 7, 12, 13
Konfigurationstabelle ....
.... 36, 52, 66
Kontrollblöcke 63
lokale Variablen 63
Mail 15
Mailbox 15
Multitasking-Kern 5
Nachricht 12
Nachrichten 15
NEAR-pointer 11
Offset 11
Preemption-Zähler 25, 27
Priorität 6, 25
PSW
  Startwert 63
READY 7
READY-Liste . 36, 57, 64, 70
Reentrancy 19
Register
  sichern 22
relokieren 64
residente Programme 21
Resource 13
Round Robin 7
Rückgabewerte 11
RUNNING 7, 46
SCB 64, 67
Segment 11
Sichern der Registerinhalte
.... 22
Speichermodelle 11
SUSPEND 54
SUSPENDED 7
System Control Block 64
Systeminformation 67
Systemtimer 12
System-Timer 6
Systemvoraussetzungen 9
Task Kontroll Block 36
Task Kontroll Blöcke 65
Tasknummer 67
Task-Zustände 7
TCB 36, 63, 65
  Größe 63
timeout 7, 8, 13
Timer 46, 55
Timer-Interrupt 55
TSR 21
UniLOC 64
UniMOS-Funktionen 22
unimos-Segment 64
Versionsnummer 64
WAITING 7, 60
Zeitscheibe 6, 57
zeitscheibengesteuert 5