Technische Universität Ilmenau
Fakultät für Informatik und Automatisierung
Fachgebiet Rechnerarchitekturen

Studienjahresarbeit

Zusammenführen der eRTOS Projekte zur eRTOS Version 2.1

Bearbeiter:	Andreas Pillekeit, 25950
Betreuer:	Dr.-Ing. Vesselka Duridanova
	Dr.-Ing. Bernd Däne
Verantwortlicher Hochschullehrer:	Prof. Dr.-Ing. habil. Wolfgang Fengler
Bearbeitungszeitraum:	1.06.02-01.12.02

1 Einführung

1.1 Thema

Im Verbundprojekt „Mehrkoordinaten Mess- und Positioniersystem“ soll Messtechnik für die Mehrkoordinatenmessung und Positionierung im Micro- und Nanometerbereich geschaffen werden. Um die anfallenden großen Datenmengen in Form von Sensordaten und komplexen Filterberechnungen in Echtzeit verarbeiten zu können soll ein Rechnerkern mit mehreren DSPs der TMS320c6x Baureihe der Firma Texas Instruments entwickelt werden. Die speziellen Anforderungen die dieses Projekt auch an die Software stellt machten es erforderlich ein eigenes Multitasking Echtzeitbetriebssystem für diesen DSP, das eRTOS, zu entwickeln. Sein modularer Aufbau und die Möglichkeit periodische Tasks mit determinierter Ausführungszeit für die Messwerterfassung und Berechnung, interruptabhängige Tasks sowie Hintergrund Tasks für Bedien- und andere nicht zeitkritische Prozesse zu erzeugen, ermöglicht der Steuerungssoftware die nötige Flexibilität die an sie gestellten Aufgaben zu erfüllen.

Mehrere Diplom- und Studienjahresarbeiten beschäftigten sich mit der Weiterentwicklung des eRTOS, dabei wurden Verbesserungen am Systemkern vorgenommen oder neue Module entwickelt. Durch diese Form der Weiterentwicklung entstanden mehrere separate Dokumentationen und Softwareprojekte. Meine Aufgabe war es die einzelnen Softwareprojekte in ein Gesamtsystem eRTOS 2.1 zu integrieren und dazu eine Dokumentation zu schaffen, die dem Anwender ein einfaches einarbeiten in das Betriebsystem ermöglicht und somit auch die Weiterentwicklung durch zukünftige Projekte erleichtert.

1.2 Ist Analyse

Zum Zeitpunkt dieser Studienjahresarbeit existierten die Projektarbeit zum „eRTOS 1.4“, [SchLev01], die Studienjahresarbeiten „DMA Unterstützung im eRTOS“, [Kra02] und „eRTOS Analyse“, [Lev00] sowie die Diplomarbeiten „Konzeption und Realisierung eines dynamisch konfigurierbaren Echtzeitbetriebsystems für den Einsatz in einem Messwerterfassungssystem“, [Sch01] und „Konzeption und Realisierung von Betriebssystemsoftware für ein Dual-Port-DSP“, [Lev02].

Die Studienjahresarbeit [Lev00] beschäftigte sich mit Scheduling des eRTOS , den Einflüssen des Systemticks auf das Laufzeitverhalten des Systems und mögliche Testverfahren. Die Projektarbeit „eRTOS 1.4“ beschäftigte sich mit der Version 1.4 des Betriebsystems die hauptsächlich statische Komponenten hatte. Darauf aufbauend beschäftigte sich die Diplomarbeit [Sch01] mit der Erweiterung dieses Systems, um dynamische Komponenten wie dynamische Speicher-, Interrupt- und Taskverwaltung sowie mit der Implementierung eines Messwertspeichers und einer Betriebsmittelverwaltung. Mit Abschluss dieser Diplomarbeit existierte die Version eRTOS 2.0. Aufbauend auf dieser Version beschäftigte sich die Studienjahresarbeit [Kra02] mit der Schaffung eines Moduls zur Unterstüzung der DMA Fähigkeiten des TMS320c6x für das eRTOS und die Diplomarbeit [Lev02] beschäftigte sich mit der Erweiterung des Nachrichtensystems des eRTOS um die Fähigkeit Nachrichten an

andere Prozessoren zu schicken. Als Hardwaregrundlage dazu diente das Daytona Board der Firma Spectrum.

Als Hardwaregrundlage für das eRTOS dienen derzeit 3 Plattformen.

Ein experimentelles Basisboard mit dem D-Modul der Firma dSignT

- Ausgabefunktion auf 8-stellige LED-Siebensegment-Anzeige
- Ausgabefunktion auf 8er LED-Block
- Eingabefunktion über 32-Tasten Tastatur
- schneller 2-Kanal A/D-Wandler mit jeweils 6 Bit Datenbreite
- + 3.3 V / + 5 V / - 5 V abgesicherte Spannungsversorgung mit Anzeige
- 8 frei zur Verfügung stehende User-LEDs mit Treiber
- Setup- und Resettaste
- RS 232 - serielle Schnittstelle

Quelle: [RAweb]

Die momentan genutzte Version des D-Moduls ist mit 512 KB SSRAM und 512 KB Flash Memory ausgerüstet.

Ein weiteres kompakteres Board mit dem D-Modul der Firma dSignT mit 2 Tastern und 2 Ausgabe LEDs sowie einer RS 232 Schnittstelle [DModweb] und ein Multiprozessorboard Daytona der Firma Spectrum. [Specweb]

2 Hardware Grundlagen

2.1 Die TMS320c6x Baureihe

Die Prozessoren der TMS320c6x Baureihe sind DSP die von Texas Instruments produziert werden. Sie sind untereinander vollständig Pinkompatibel und Befehlskompatibel zum ersten DSP der Baureihe dem TMS320c6201. Dieser DSP kann Festpunktoperationen mit einer Geschwindigkeit von 1600 MIPS berechnen. Seine Zykluszeit beträgt 5 ns, pro Zyklus kann er acht 32Bit Instruktionen ausführen. Er besitzt 6 Festpunkt ALUs und zwei 16 Bit Multiplizierer. Genauere Informationen unter [TIweb]

Neben weiteren TMS320c62x DSP Varianten bis 2400 MIPS existieren auch TMS32064x DSPs. Sie haben eine ähnliche Hardwarearchitektur und beherrschen auch nur Festpunktoperationen, aber bis zu einer Geschwindigkeit von 4800 MIPS, bei einer Zykluszeit von 1,67 ns und 28 Operationen pro Zyklus.

Die DSP der TMS320c67x Baureihe beherrschen zusätzlich zu den Festpunktoperationen noch Fließkommaoperationen. Der TMS320c6701 kann 1 GFLOP berechnen, bei einer Zykluszeit von 6 ns. Er besitzt 4 Fließkomma/Festpunkt ALUs, 2 Festpunkt ALUs und 2 Fließkomma/Festpunkt Multiplizierer. Genauere Informationen unter [TIweb]

In diesem Projekt wurde bisher der TMS320c6201 und der TMS320c6701 eingesetzt. Einige der optimierten Speicherzugriffsbefehle des TMS320c6701 werden vom eRTOS unterstützt (Siehe 3.4. Implementierung)

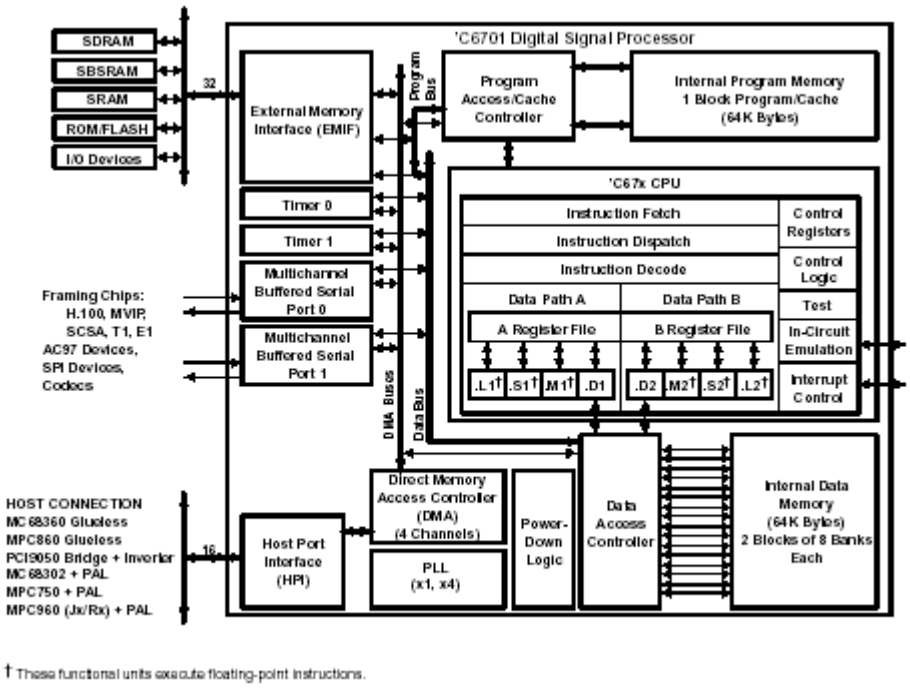


Abb. 1 Blockschaltbild des TMS320c6701, Quelle: [TIweb]

2.2 DMA Hardware des TMS320C6x

Der TMS320C6x unterstützt das Senden von DMA Nachrichten über bis zu vier DMA Kanäle. Der hardwaremäßig existierende fünfte Kanal ist nicht benutzbar, da er vom Host Interface zum Zugriff auf den Systemspeicher genutzt wird.

Jeder der vier benutzbaren DMA Kanäle kann Daten zwischen internen und externen Daten- und Programmspeichern sowie zu externen E/A – Einheiten transferieren.
Für jeden Kanal existieren fünf 32 Bit Kontrollregister:

- Primary Control Register
  - Hier können grundlegende Einstellungen zur DMA Nachrichtenübertragung vorgenommen werden. (z.B. Behandlung der Quell- und Zieladressen, Prioritätssteuerung, Elementgröße, Interruptaktivierung und Synchronisierung)
- Secondary Control Register
  - Hier wird der aktuelle Status des DMA Kanals und die aufgetretenen Ereignisse für jeden DMA Kanal angezeigt (z.B. Kopiervorgang beendet, Synchonisationsereignis aufgetreten)
- Transfer Control Register
  - Hier wird festgelegt wie viel Elemente kopiert werden sollen.
- Source Address Register
  - Quelladresse
- Destination Address Register
  - Zieladresse

Nähere Erläuterungen zu den Registern in [Kra02] S. 5ff.

2.3 Das Daytona Board

Das PCI-Board Daytona der Firma Spectrum ermöglicht es zwei Prozessoren der TMS320C6x Baureihe Parallel zu betreiben. Jedem Prozessor stehen dabei 512KB synchroner SRAM (Adressbereich: 0x0040 0000-0x0047 FFFF) und 16MB SDRAM (Adressbereich: 0x0200 0000-0x02FF FFFF) zur Verfügung. Auf die 32KB Dual-Port Ram SDRAM (Adressbereich: 0x0174 0000-0x0174 7FFF) haben beide Prozessoren Zugriff.

Der Dual-Port Ram ist von beiden Prozessoren direkt adressierbar (in 32 Bit Blöcken) und ermöglicht somit einen einfachen Austausch von Daten zwischen den Prozessoren. Die Konsistenz der Daten bei gleichzeitigem Zugriff auf den Dual-Port Ram muss aber von den Schreib-Lese Algorithmen selbst gewährleistet werden, da es keine in die Hardware integrierte Sperrmöglichkeit gibt.

Der Kommunikationsweg einer Kommunikation über Dual-Port Ram ist in der Abb. 2 dargestellt.

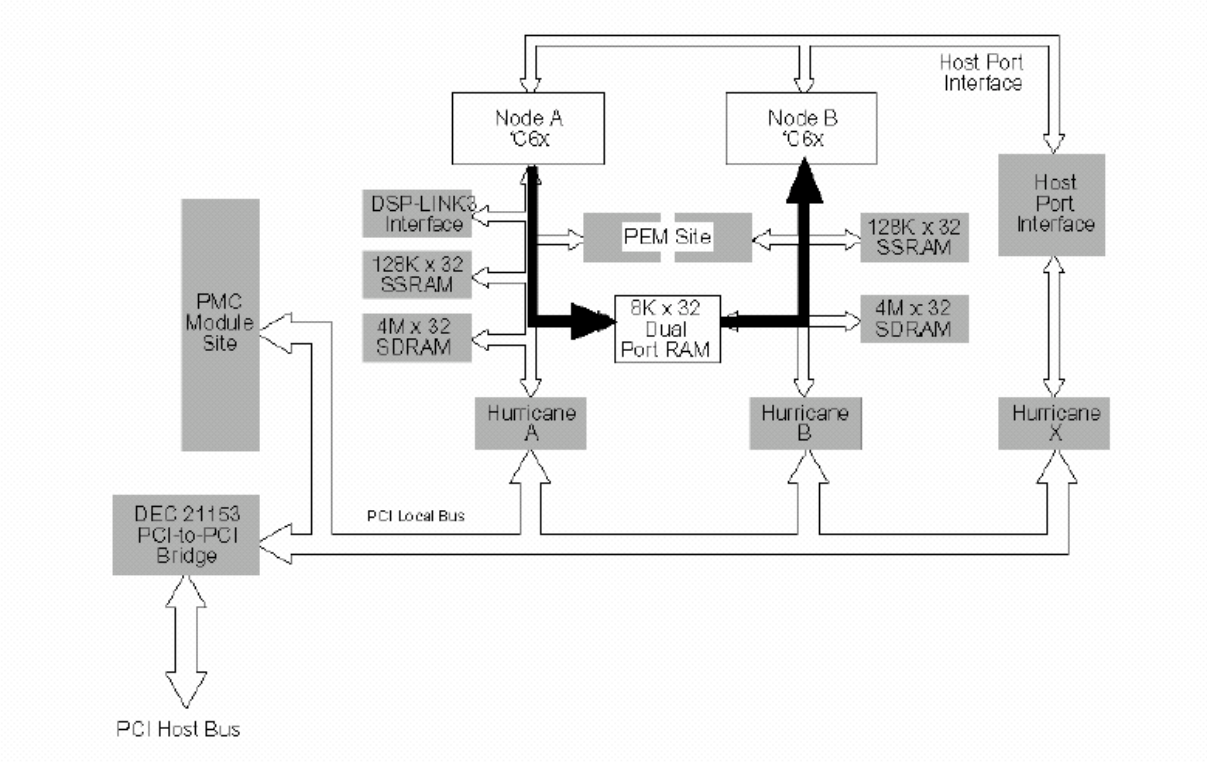


Abb. 2 Datentransfer über Dual-Port Ram, Quelle: [Lev02] S.33

Die Möglichkeit des Datenaustausches über den Dual- Port Ram wird von der Dual-Prozessor Nachrichtensystemerweiterung genutzt.

Um größere Datenmengen zu übertragen, besteht weiterhin auch noch die Möglichkeit der Kommunikation von SSRAM zu SSRAM oder von SSRAM zur CPU über den Hurricane DMA Controllerchip des Daytona Boards. Diese Kommunikationsmöglichkeiten werden vom eRTOS nicht speziell unterstützt. Weitere Informationen unter [Specweb]

Die Kommunikationswege unter Einbeziehung des Hurricane Controllers sind nachfolgend dargestellt.

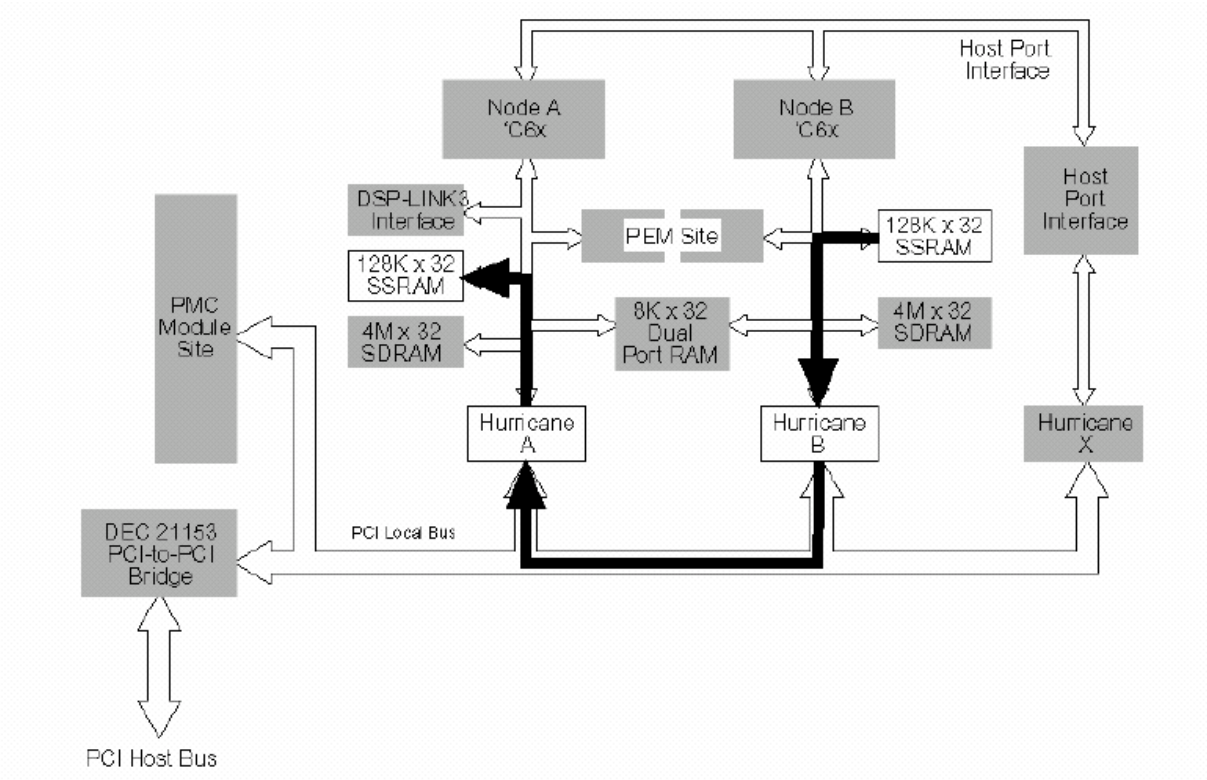


Abb. 3 Datentransfer über Hurricane Controller von SSRAM zu SSRAM,
Quelle: [Lev02] S.34

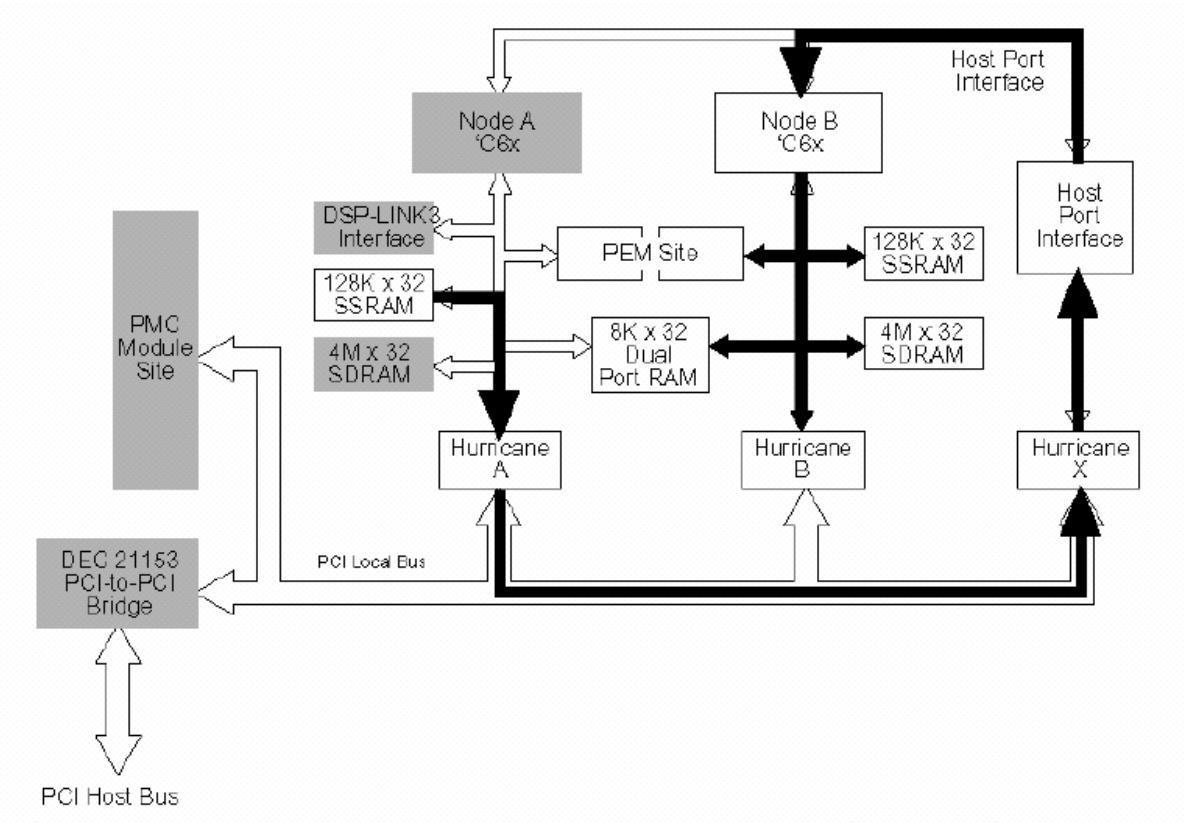


Abb. 4 Datentransfer über Hurricane Controller von SSRAM zur benachbarten CPU,
Quelle: [Lev02] S.35

3 Das eRTOS

3.1 Merkmale des eRTOS 2.1.

Das eRTOS 2.1 ist ein Multitasking Echtzeitbetriebsystem für den Einsatz in eingebetteten Systemen bei Verwendung eines TMS320C6x Prozessors von Texas Instruments auf einem Einzel- oder Multiprozessorboard.
Es wurde mit dem Code Composer Studio V 1.20 von Texas Instruments unter der Verwendung der Programmiersprache C und Assembler entwickelt.

Es verfügt momentan über folgende Eigenschaften und Komponenten.

- Dynamisches Scheduling:
    - Kooperierende und nicht kooperierende Tasks können zur Laufzeit des Systems erstellt, aktiviert, deaktiviert und gelöscht werden.
    - Periodendauer und Startzeitpunkte sind ebenfalls veränderbar
- Dynamische Speicherverwaltung
    - Möglichkeit der Nutzung von unterschiedlichen Speicherbereichen und Speicherarten
    - Reservierung und Freigabe von Speicher zur Laufzeit des Systems.
- Dynamische Interruptverwaltung
    - Den nichtkooperierenden Tasks können zur Laufzeit Interruptabhängigkeiten zugeordnet werden
- Messwertspeicher
    - Spezielle Routinen für die einfache Erfassung und Speicherung von Messwerten
- Nachrichtensystem
    - Nachrichten (maximale Größe 96 Bit) können an einzelne Tasks versand werden. Auch im Dualprozessorbetrieb zwischen den Tasks einzelner Prozessoren.
- Betriebsmittelverwaltung
    - Spezielle Funktionen die Mechanismen zur Steuerung und Kontrolle von exklusiv nutzbaren Ressourcen zur Verfügung stellen.
- DMA Unterstützung
    - Daten können mit Hilfe der DMA Funktionalitäten des TMS320C6x übertragen werden

Die Konfiguration der einzelnen Elemente des Betriebssystems erfolgt zentral zur Compilezeit. Das bedeutet, dass das Nutzerprogramm und das Betriebssystem zur gleichen Zeit übersetzt werden.

3.2 Aufbau und Funktionsweise des eRTOS 2.1.

3.2.1 Dynamisches Scheduling

Die kleinste Einheit der internen Zeitmessung die das Betriebssystem kennt, ist ein „Tick“. Dieser Tick wird mit jedem Timer 0 Interrupt um eins erhöht. Die Periodenzeit der Timer 0 Intervalle wird in der Config.h über die Ticklänge angegeben (siehe Implementierung). Das Betriebssystem verwendet den Tick für die Verwaltung und Steuerung der Ausführungszeiten der einzelnen Tasks. Mit jedem Tick wird die Zeitverwaltungsroutine des eRTOS gestartet, die das Scheduling steuert.

Hinweis:
Die Systemzeit (Tick) wird in einer 32Bit Integervariable gespeichert. Die Routinen sind so gestaltet, das ein Überlaufen der Variable bei langen Systemlaufzeiten keine Probleme verursacht. Bei Änderungen der Taskeigenschaften kurz vor dem Überlaufen der Variable kann es aber zu einer Verschiebung der Startzeiten kommen. (Siehe [Sch01] S.88)

Das Betriebssystem verwaltet die Nutzeraufgaben über 3 verschiedene Taskarten.

- Kooperierende Tasks
    - Besitzen eine statisch festgelegte bekannte Periode, in der Sie zyklisch wiederholt werden
    - Haben eine höhere Priorität als nk-Tasks
    - Die Ausführung wird über das Prinzip des Ratenmonotonen Scheduling vom Betriebssystem gesteuert
    - Sind nicht unterbrechbar (nonpreemptiv) (außer durch SISR)
- Nichtkooperierende Tasks
    - Sie werden in der Zeit ausgeführt, in der keine k-Tasks laufen.
    - Besitzen keine festgelegte Periode
    - Die Ausführung wird über das Round-Robin-Prinzip vom Betriebssystem gesteuert
    - Sind unterbrechbar (preemptiv)
- Interruptabhängige Tasks
    - Sind eine spezielle Gruppe von nk-Tasks, die mit dem Eintreten des ihnen zugeordneten Interrupts ausgeführt werden.
    - Werden im Round-Robin-Scheduling System bevorzugt behandelt

Die Tasks werden durch Ihre Taskidentifikationsnummer (TaskID) voneinander unterschieden. Dabei existiert die Konvention das k-Tasks TaskIDs zwischen 0 und 99 erhalten und nk-Tasks TaskIDs ab 100. Diese TaskID muss im System einmalig sein. Im Mehrprozessorbetrieb ist darauf zu achten, dass keine Dopplungen der IDs auf den einzelnen Prozessorplattformen existiert.
Da k-Tasks die höchste Priorität besitzen und nicht unterbrechbar sind, sollten sie wichtige rechenintensive oder zeitkritische Aufgaben übernehmen (z.B. Messdaten einlesen, Filterberechnungen). Die maximale Ausführungszeit des Codes des k-Tasks darf dabei die zugewiesene Periodendauer nicht überschreiten. Aufgrund ihrer niedrigen Priorität und Unterbrechbarkeit eignen sich nk-Tasks für untergeordnete Aufgaben wie Bildschirmausgaben und Tastaturabfragen.

Für kTasks existiert folgendes Taskzustandsdiagramm

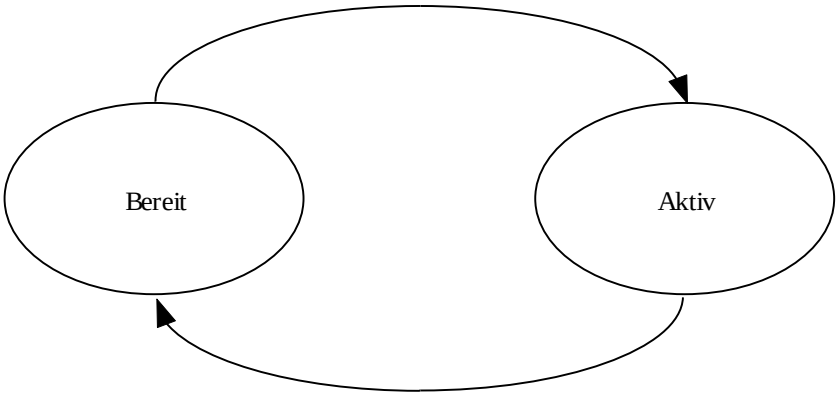


Abb. 5 Taskzustandsdiagramm für k-Tasks

Das eRTOS verwaltet die Ausführung der einzelnen k-Tasks nach dem Prinzip des Ratenmonotonen Scheduling.

In einem Array werden pro k-Task folgende Zustände gespeichert:

TaskID	Zeiger auf die Funktion	Periode	Start Tick	Letzte Aktivierung	Nächste Aktivierung

Wegen des Geschwindigkeitsgewinnes ist die Zeitverwaltungsroutine teilweise in Assembler geschrieben. Der Stackwechsel und das sichern der Registerinhalte wird durch Code, der bei Interrupt 4 (Timer 0) ausgeführt wird, in der Assem.asm Datei durchgeführt, die dann die zeitverwalt() C-Funktion aufruft.

Für das Ratenmonotone Scheduling im eRTOS 2.1. existiert folgender Ablaufplan.

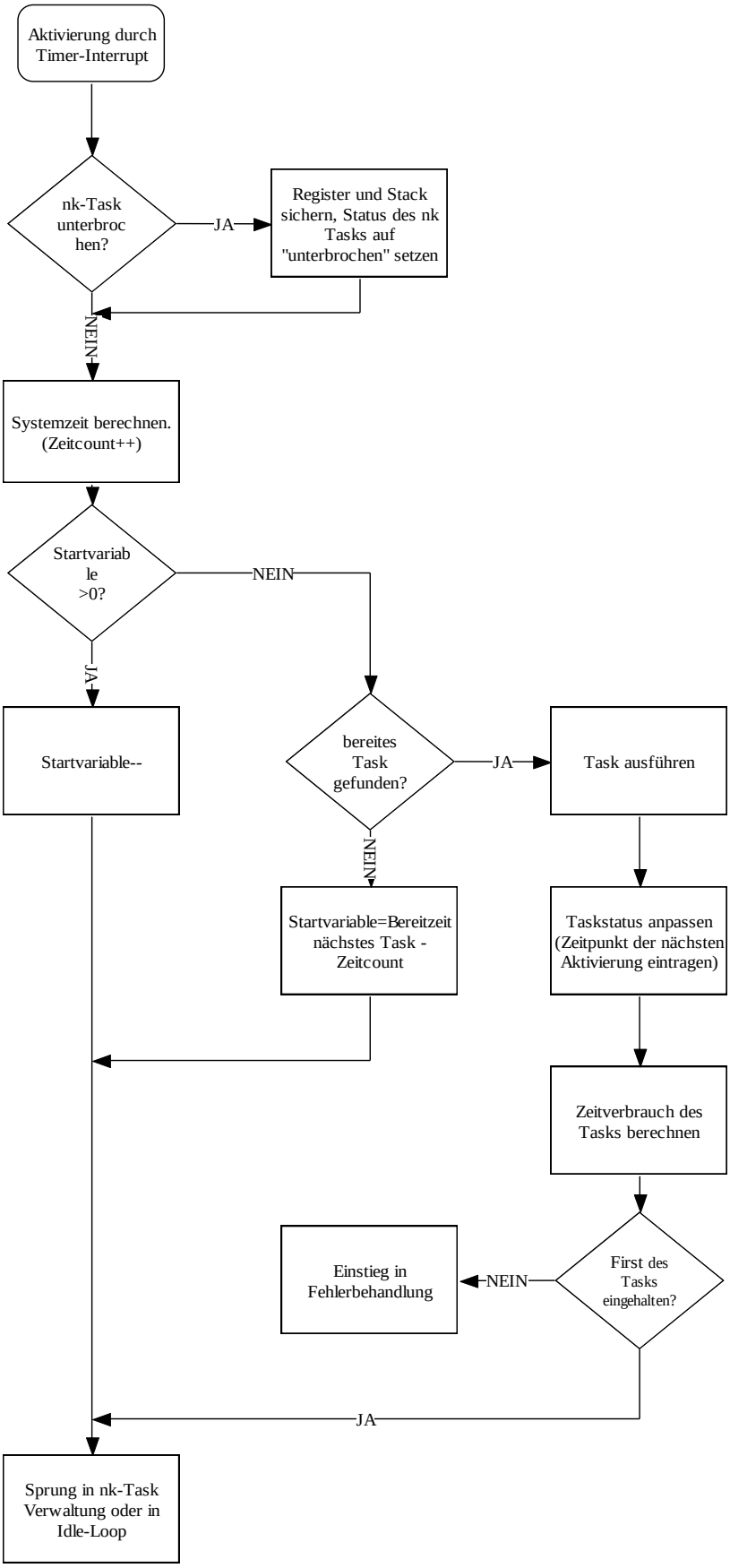


Abb. 6 k-Task Verwaltung, Quelle: [Sch01] S. 87.

Zeitverbrauch für das Ratenmonotone Scheduling:

	CPU Takte
Min. Zeitbedarf bei 2 Tasks	301
Max. Zeitbedarf bei 2 Tasks	366
Differenz	65
Anwachsen des Bedarfs pro weiteren Task	35

Quelle: [Sch01] S.89

Das flexiblere ratenmonotone Scheduling des eRTOS 2.1. benötigt mehr Zeit pro Task als in der Version 1.4. und steigt linear mit der Anzahl der Tasks.

Für nk-Tasks existiert folgendes Taskzustandsdiagramm

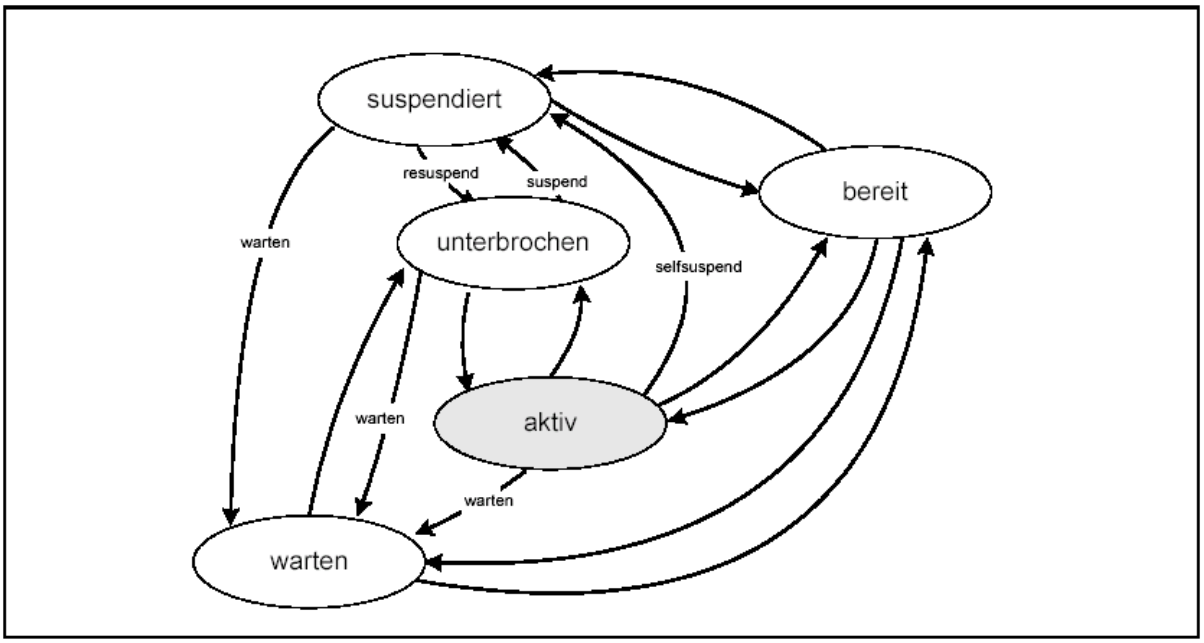


Abb. 7 Taskzustandsdiagramm für nk-Tasks, Quelle: [Lev02] S. 25

Das eRTOS verwaltet die Ausführung der einzelnen nk-Tasks nach dem Prinzip des Round Robin Scheduling. Die zur Verfügung stehende Ausführungszeit wird definiert durch die freien Zeitintervalle zwischen der Ausführung der einzelnen k-Tasks. Von diesem Zeitkonto muss aber noch die Ausführungszeit der nk-Task Verwaltung abgezogen werden.

In einem Array werden pro nk-Task folgende Informationen gespeichert.

TaskID
Status (z.B. Bereit)
Priorität
zugehöriger Interrupt
Wartegrund (z.B. Zeit)
Warteoption1
Warteoption2
Unterbrechung
Zeiger auf Funktion
Register (32 x 32 Bit)
Stack (80 x 32 Bit)
Isr
Suspendzeit (für warten auf Zeit)

Unterbrechung wird nur systemintern verwendet, um festzustellen, ob der Task unterbrochen wurde. ISR wird nicht mehr ausgewertet.

Das eRTOS kennt 3 Prioritätsstufen (Hintergrund, Mittel und Hoch) für nk-Tasks. Pro Prioritätsstufe existiert eine Warteschlange aller ausführbaren und momentan suspendierten nk-Tasks.

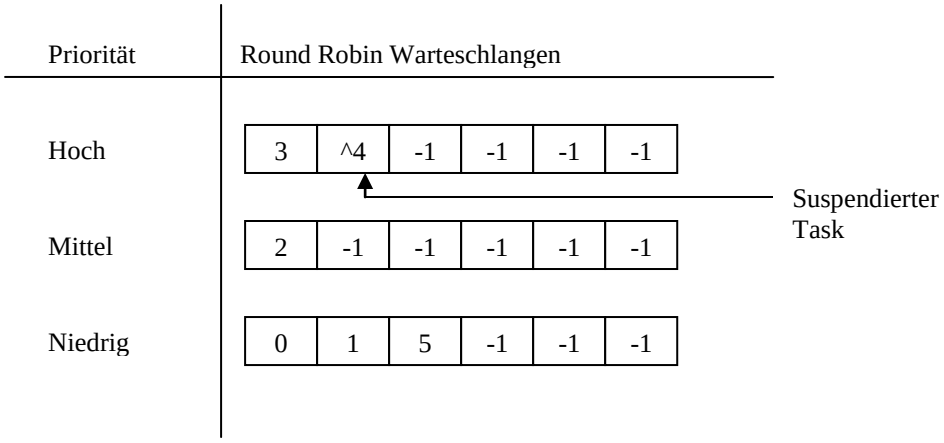


Abb. 8 Prioritätslisten im eRTOS 2.1. [Sch01] S. 92

Pro Verwaltungsdurchlauf wird geprüft ob die einzelnen Warteschlangen (beginnend mit der höchsten Priorität) ausführbare Tasks enthalten. Falls ja werden diese nach dem Round Robin Prinzip ausgeführt. Falls nein wird die Warteschlange mit der nächst niedrigeren Priorität geprüft. Für interruptabhängige nk-Tasks gilt, dass sie beim Eintreten des zugehörigen Interrupts vom Zustand „Suspendiert“ in den Zustand „Bereit“ wechseln.

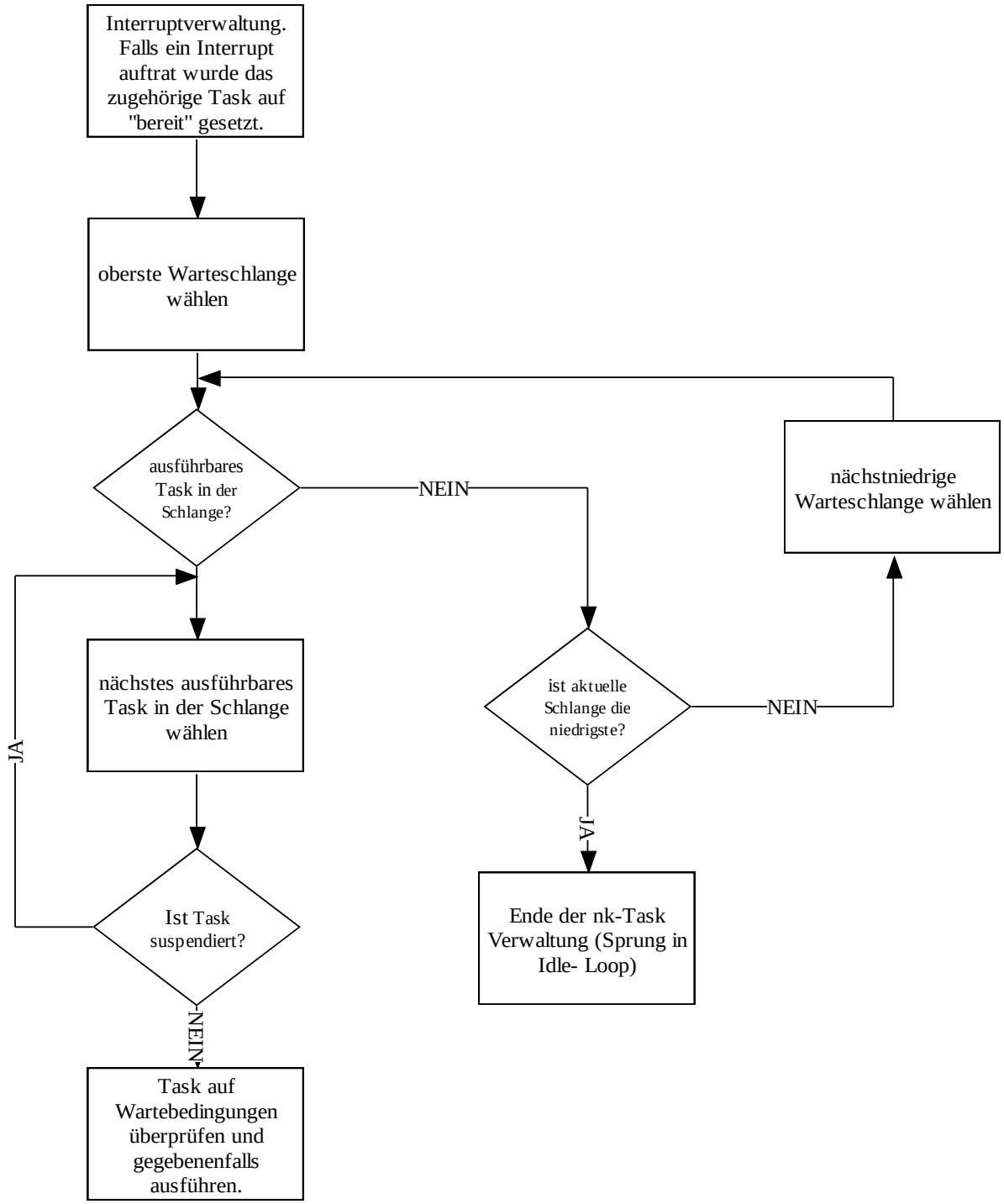


Abb. 9 Ablaufplan nk-Task Verwaltung, Quelle [Sch01] S.94

Nach einer Unterbrechung wird ein nk-Task nicht neu gestartet, sondern an der jeweiligen Stelle weiter ausgeführt. Es kann aber vereinzelt zu Neustarts kommen (Siehe 3.3 Fehler im eRTOS 2.0). Ein nk-Task kann auch in den Zustand „warten“ überführt werden. Der Task wird dann bis zum Eintreten eines bestimmten Ereignisses nicht mehr ausgeführt. Als Ereignis kann der Ablauf einer bestimmten Anzahl von Ticks, das Eintreffen einer Nachricht über das Nachrichtensystem oder das Freiwerden eines Betriebsmittels gewählt werden. Der Wartegrund „warten auf Betriebsmittel“ ist nicht sinnvoll ohne die Betriebsmittelverwaltung nutzbar und sollte deswegen auch nur von der Betriebsmittelverwaltung benutzt werden.

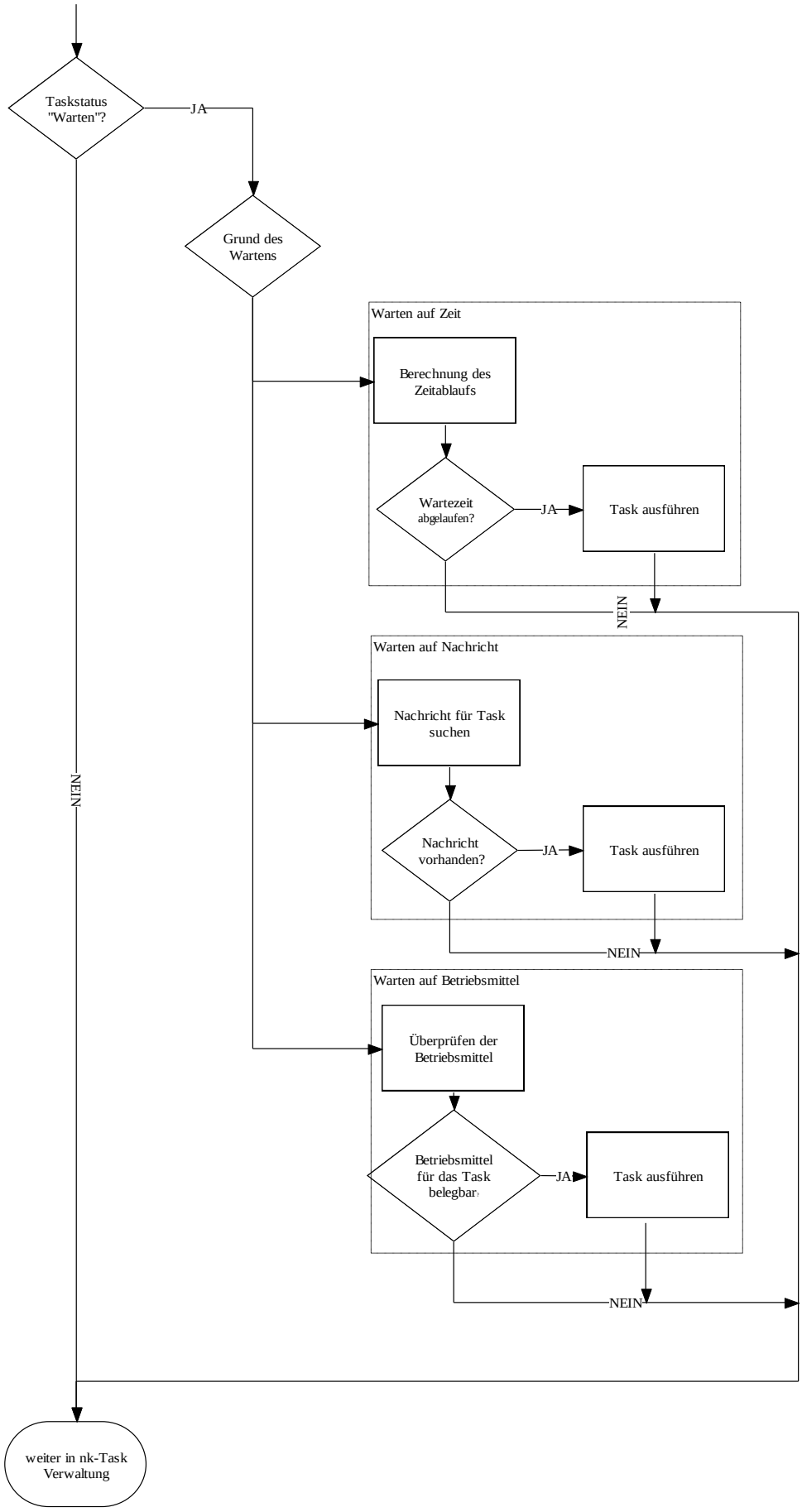


Abb. 10 „Warten Auf“ Ablaufplan in der nk-Taskverwaltung Quelle: [Sch01] S. 95

3.2.2 Dynamische Speicherverwaltung

Das dynamische Speicherverwaltungssystem ermittelt freie Speicherbereiche und ist in der Lage zur Laufzeit auf Speicheranforderungen der Nutzertasks zu reagieren. Dabei wird versucht die Fragmentierung des Speichers möglichst gering zu halten. Mit dem dynamischen Speicherverwaltungssystem können verschiedene Arten von Speicher verwaltet werden und gezielt Speicher eines bestimmten Typs zugewiesen werden.

Zurzeit wird der prozessorinterne Speicher für das Betriebssystem und dessen dynamischen Systembereich (Heap) sowie für Bios und Interupt Service Tabelle (IST) genutzt, der externe Speicher steht den Nutzertasks zur Verfügung.

Wie viel Speicher in den einzelnen Speicherbereichen für die Speicherverwaltung zur Verfügung steht, wird in der Datei Alloc.h festgelegt.

```
#define G_INTERN 1000    // groesse heap interner ram (Element á 32 Bit)
#define G_EXTERN 1000    // groesse heap externer ram (Element á 32 Bit)
#define G_SBS    1000    // groesse heap sbs-ram (Element á 32 Bit)
```

Dabei ist zu beachten, dass der tatsächlich zur Verfügung stehende physikalische Speicher nicht überschritten wird.

Als Verwaltungsmethode für den dynamischen Nutzerbereich wurde die Methode des Segregated Fits mit exakten Listen plus Binärbaum gewählt (Siehe [Sch01] S. 13 ff.). Da bis auf short und char alle Datentypen vielfache von 32 Bit sind, lassen sich sehr einfach Größenklassen für diese Methode finden, nämlich vielfache von 32 Bit.

Es existieren drei unterschiedliche Verwaltungsstrukturen für den Speicher. Diese Verwaltungsstrukturen existieren jeweils nebeneinander für jede Speicherart (die Anzahl der Speicherarten wird in SAnzahl in Alloc.h definiert). Für häufig auftretende Größenklassen existiert ein Array (karray), welches die Adresse des ersten freien Elementes jeder festen Speichergrößenklasse (es existieren momentan 32 feste Größenklassen) in jeder Speicherart speichert. Dieses Element enthält wiederum die Adresse des nächsten freien Elementes dieser Klasse. So entsteht eine Liste mit freien Elementen für jede Größenklasse in jedem Speicherbereich. Die einzelnen Elemente haben folgenden Aufbau:

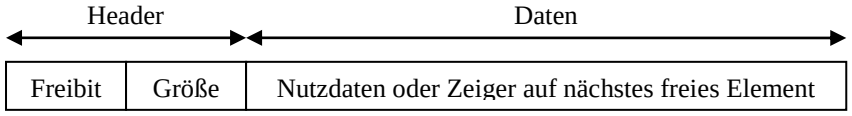


Abb. Aufbau eines Speicherblockes, Quelle [Sch01] S.70

Das „Freibit“ dient zur Kennzeichnung freier Blöcke, Größe gibt die Länge des Datenbereichs an. In diesem Datenbereich können sich Nutzdaten oder, falls der Block frei sein sollte, die Adresse des nächsten Elementes der Liste befinden.
Für Datentypen die kleiner gleich 32 Bit sind (also integer, float, short, char) ist diese Methode aber uneffektiv (wegen des Header- Nutzdatenverhältnisses) oder nicht einsetzbar (weil sie zu klein sind um die 32 Bit Adresse des Nachfolgers zu speichern). Deswegen wird für diese Datentypen das Simple Segregated Storage Verfahren angewandt. Für die

Datentypen werden hier Speicherseiten für die einzelnen Elemente eingerichtet, in die mehrere dieser kleinen Datentypen abgelegt werden können.

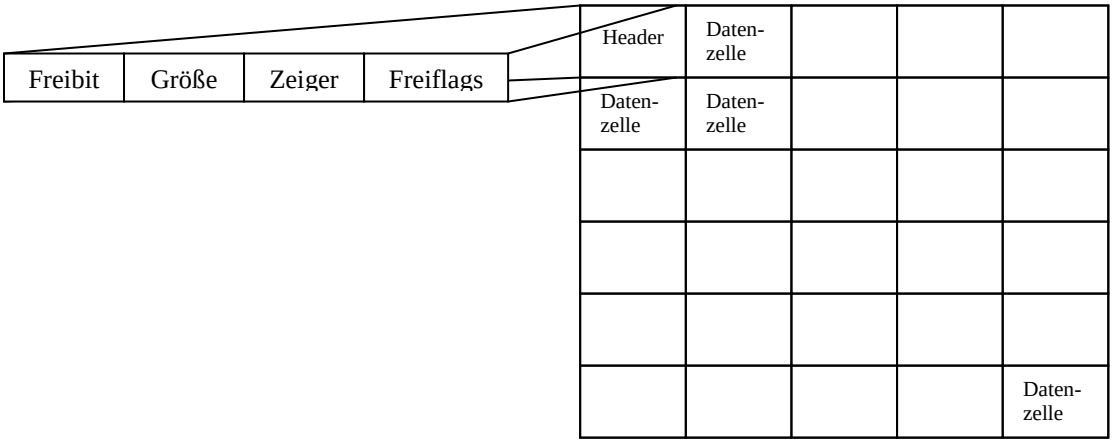


Abb. 11 Speicherseite für Simple Segregated Storage Verfahren, Quelle: [Sch01] S.70

Zusätzlich zu dem Freibit und der Größenangabe wird hier noch ein Zeiger auf die nächste Speicherseite und eine Angabe welche Elemente der Speicherseite frei sind (Freiflags) benötigt. Es existieren Speicherseiten für 16 und 32 Bit Datenfelder. Die Adresse der ersten Speicherseite jeden Typs wird in einer Zeigervariablen (rootpg) gespeichert.

Für die Verwaltung von Speicherbereichen, die nicht in den Größenklassen liegen, die durch das Größenklassen Array (karray) repräsentiert werden oder durch die Speicherseiten verwaltet werden, wird eine Baumstruktur erzeugt. Jeder Knoten im Baum hat folgenden Aufbau.

Klassengröße	Zeiger Linkes Blatt	Zeiger Rechtes Blatt	Zeiger Liste der Elemente

Abb. 12 Aufbau der Knoten für Baumstruktur, Quelle: [Sch01] S. 72

Im Feld Klassengröße wird die Größe des Speicherbereichs, der durch diesen Knoten repräsentiert wird, abgelegt. Danach werden die Zeiger auf die nachfolgenden Knoten in der Baumstruktur mit anderen Klassengrößen gespeichert. Am Ende befindet sich der Zeiger auf das erste freie Element der Liste der Elemente dieser Klassengröße (ähnlich karray). Die einzelnen Knoten werden generiert wenn ein entsprechend großer Speicherbereich freigegeben wird und wieder gelöscht, wenn ihre Elementliste keine freien Speicherbereiche mehr enthält.
Zum besseren Verständnis der dynamischen Speicherverwaltung nachfolgend die Ablaufpläne der Algorithmen. Quelle: [Sch01] S. 74 ff.

nAlloc

Dient der Reservierung von Speicher.

nFree

Der dem Zeiger zugeordnete reservierte Speicher wird freigegeben

Del Knoten

Entfernt einen nicht mehr benötigten Knoten (weil seine Elementliste leer ist) aus dem Baum und gibt den vom Knoten belegten Speicher frei.

Insert Tree

Diese Funktion fügt einen neuen Knoten mit gewünschter Größe dem Speicherbaum hinzu. Der Test ob diese Größe im Baum vorhanden ist wurde bereits in „im Tree“ durchgeführt und verneint.

Im Tree

Testet ob eine bestimmte Blockgröße im Speicherbaum bereits vorhanden ist. Wenn ja wird über den Zeiger „leer“ auch die Adresse des Elementes mit geliefert.

Iteratives Splitting

Zelle von größerem Block oder aus Heap splitten.

Simple Storage

Speichern eines Elementes in einem Speicherseitenfeld. (Für Datentypen <=32Bit)

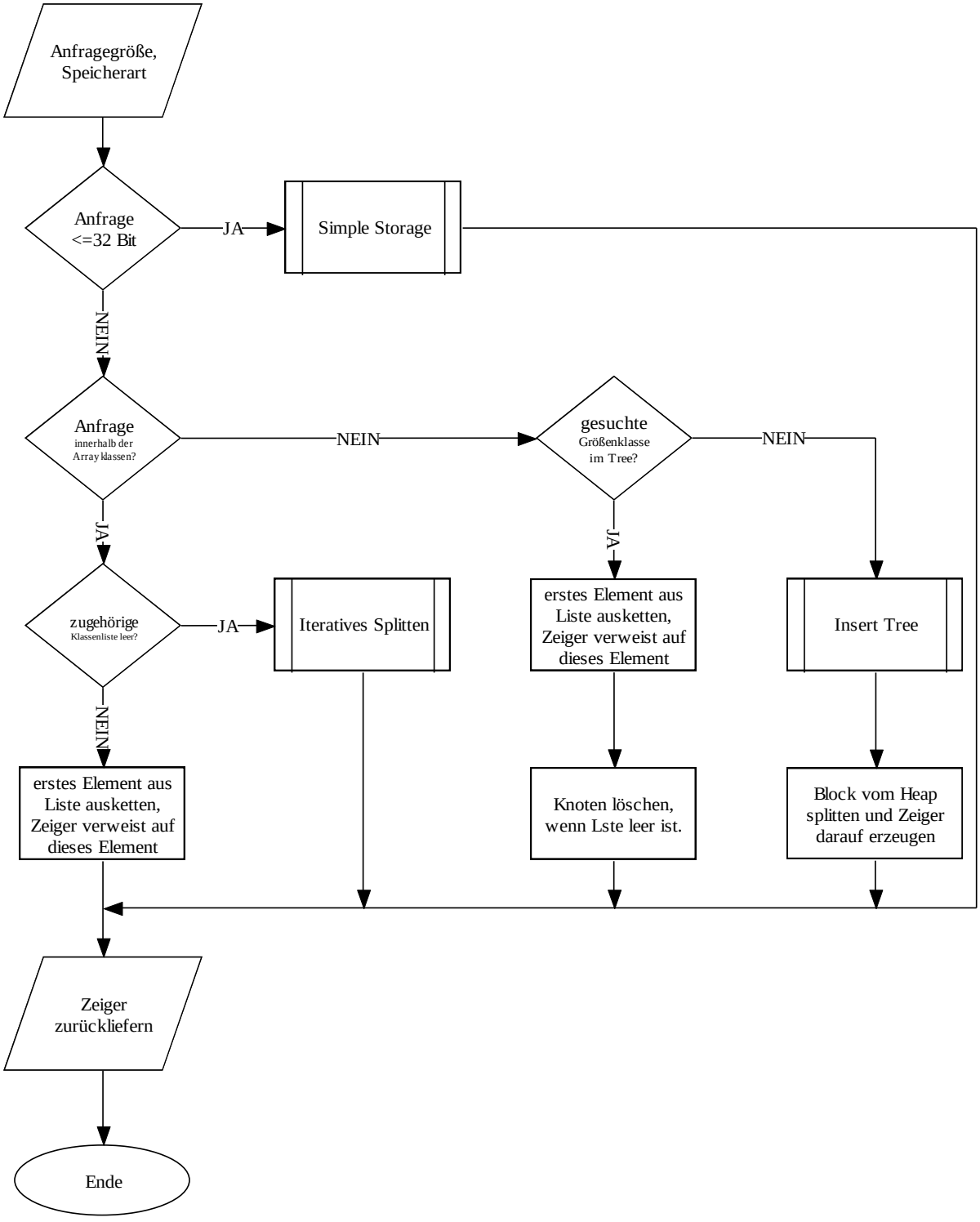


Abb. 13 nAlloc, Quelle: [Sch01]

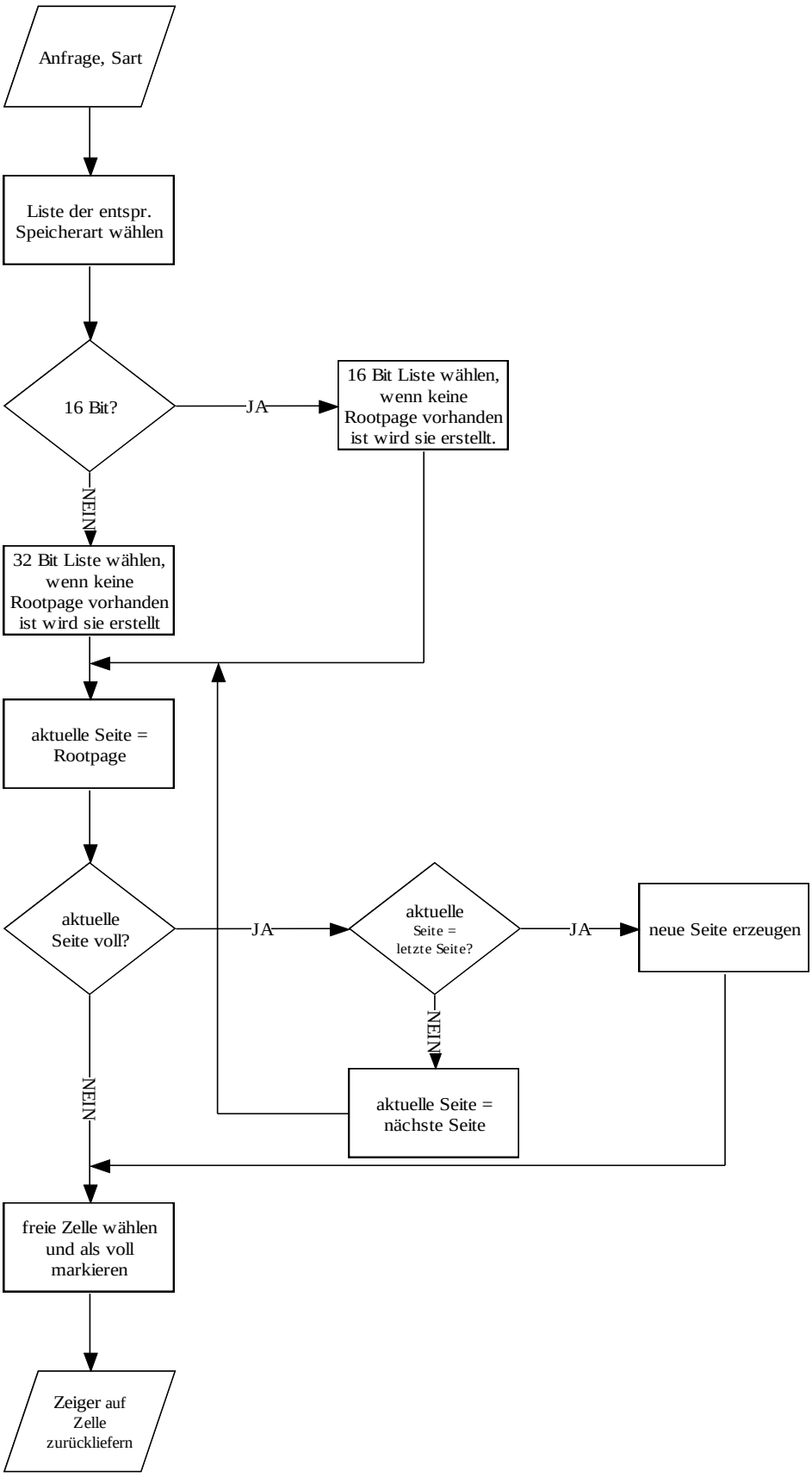


Abb.14 Simple Storage, Quelle: [Sch01]

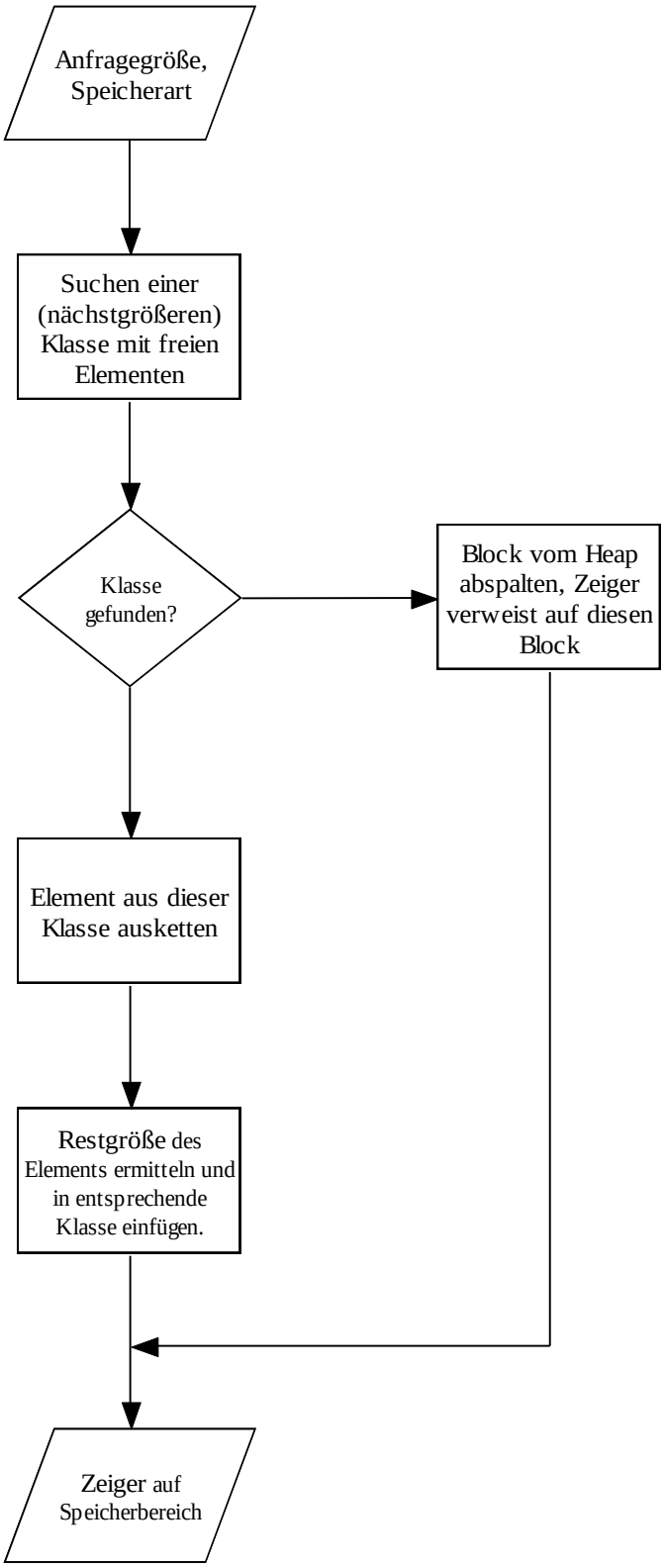


Abb. 15 Iteratives Splitting, Quelle: [Sch01]

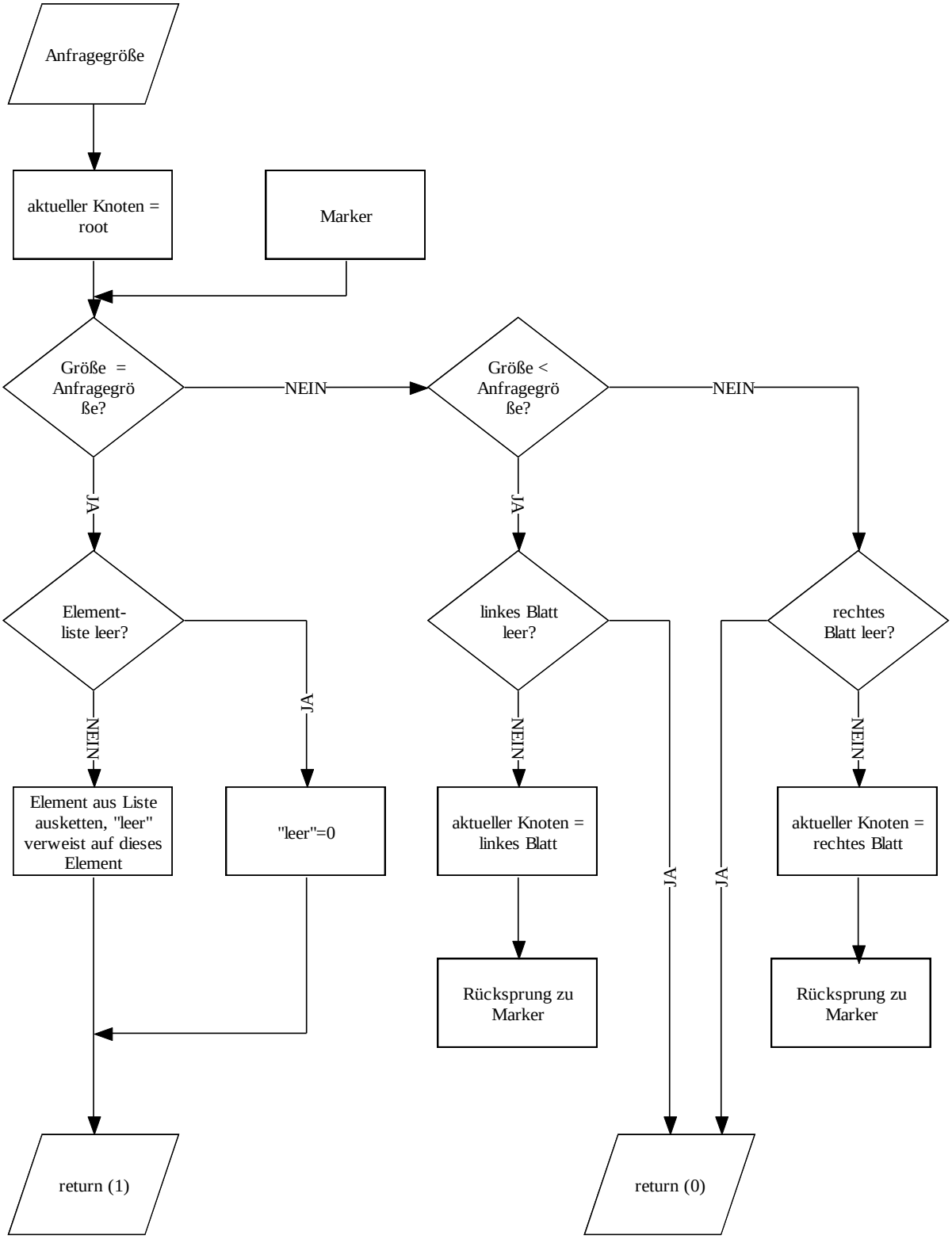


Abb. 16 Im Tree, Quelle: [Sch01]

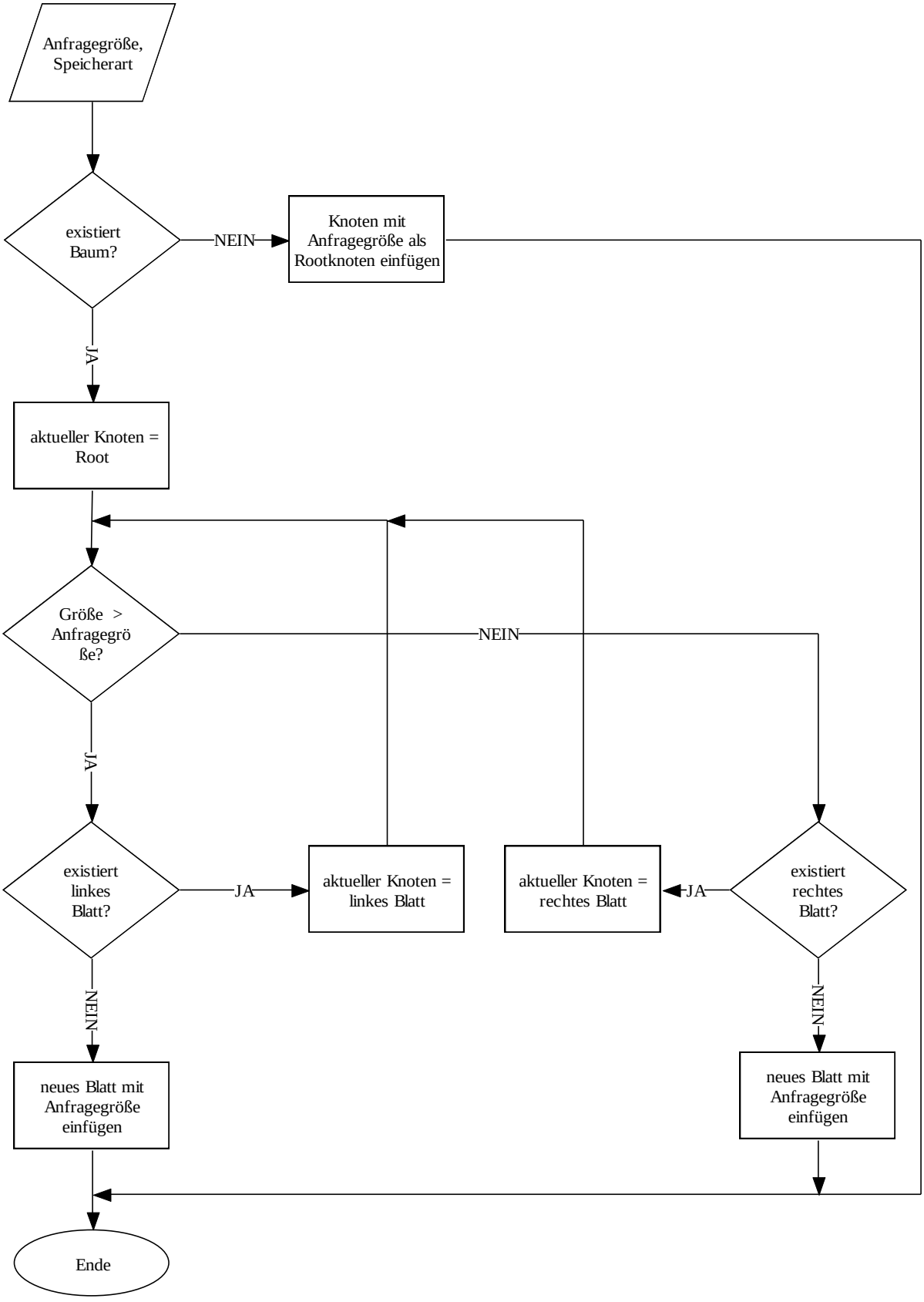


Abb. 17 Insert Tree, Quelle: [Sch01]

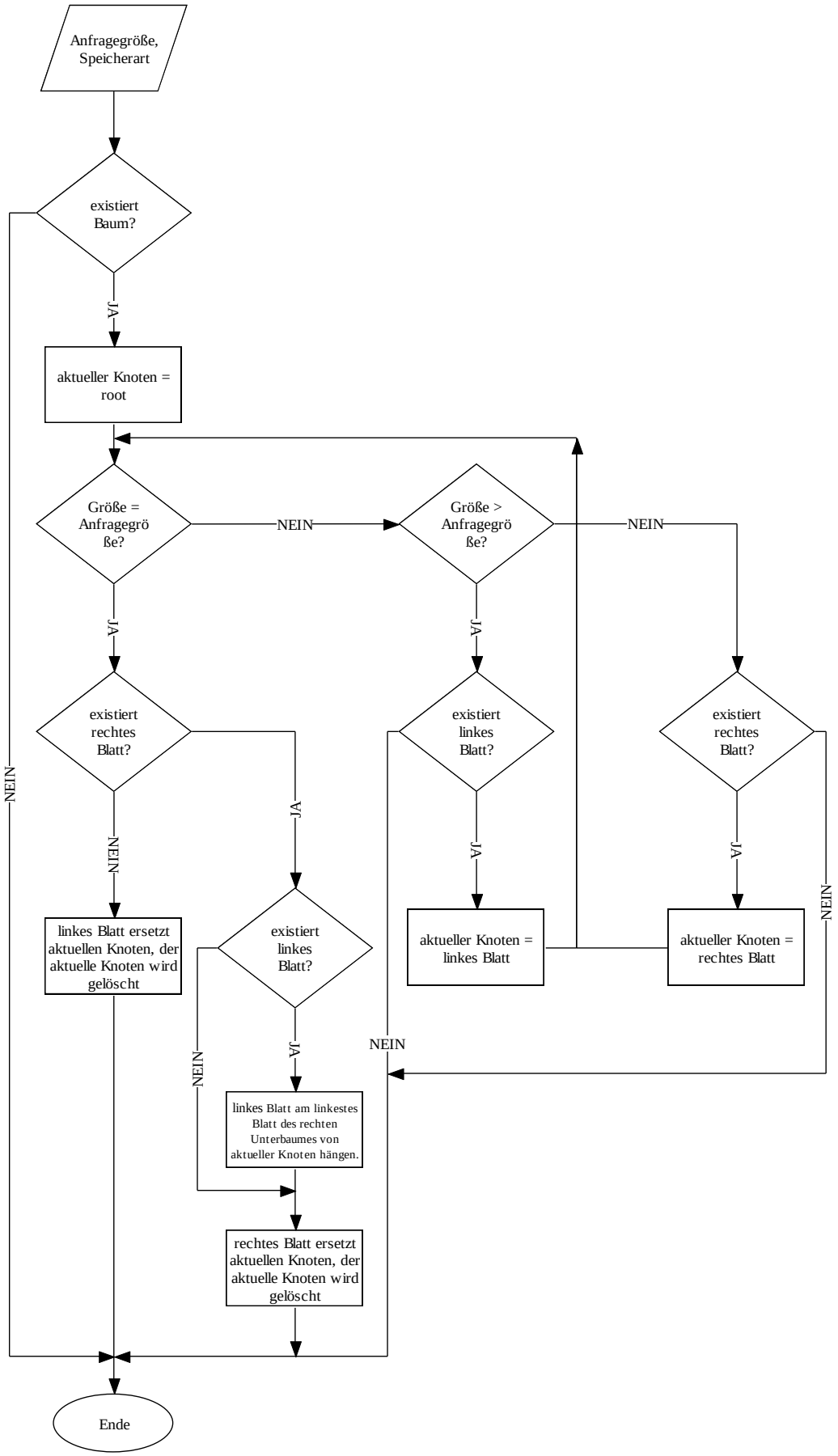


Abb. 18 Del Knoten, Quelle: [Sch01]

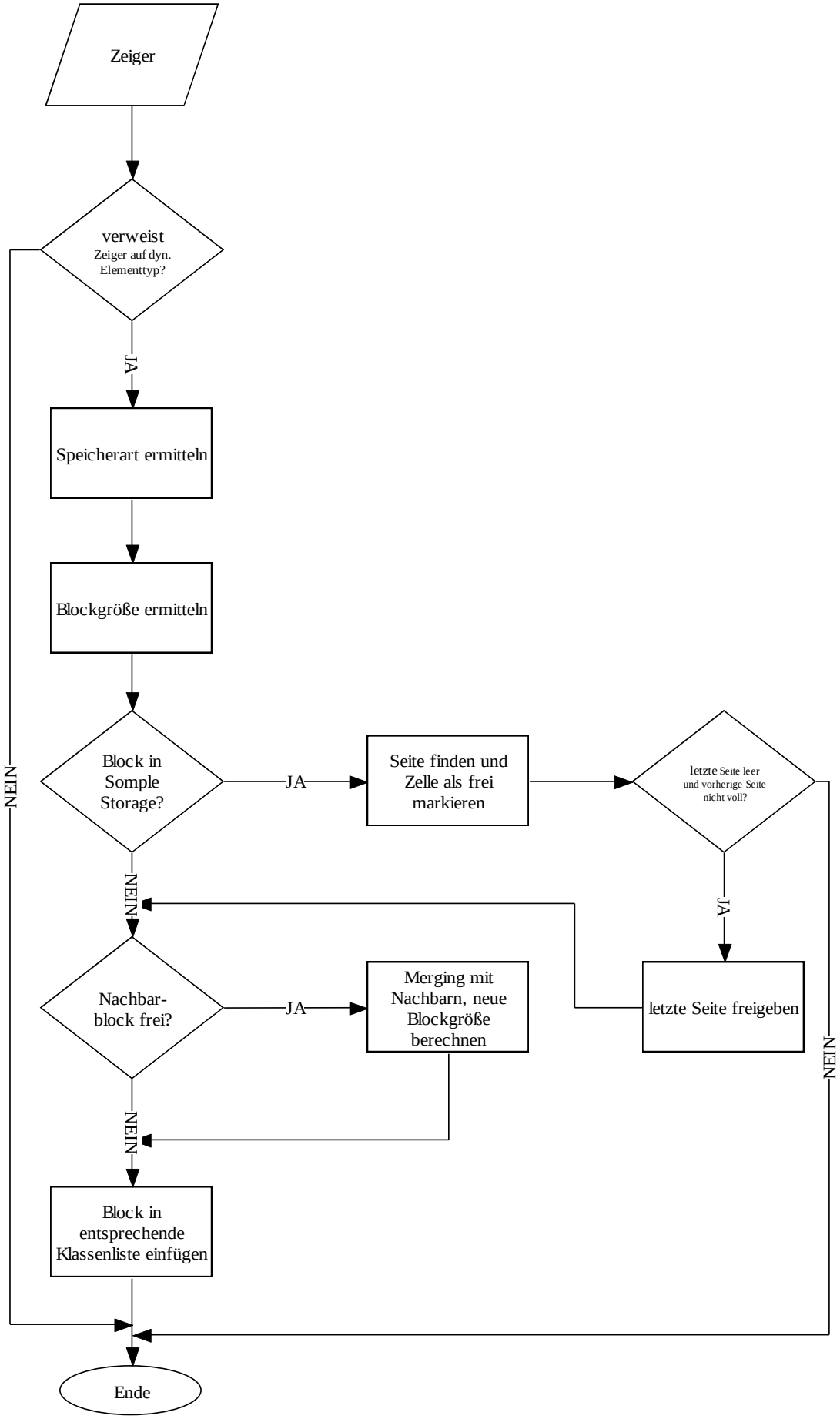
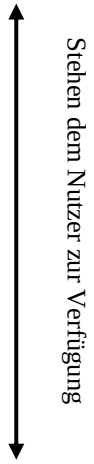


Abb. 19 nFree, Quelle: [Sch01]

3.2.3 Dynamische Interruptverwaltung

Die dynamische Interruptverwaltung stellt Funktionen bereit die es ermöglichen nk-Tasks in Abhängigkeit von bestimmten äußeren Ereignissen auszuführen, die über Interrupts gekennzeichnet werden.
Die Zuordnung der Interrupts wurde über das Interrupt Multiplexregister verändert, um eine höhere Priorisierung der Timerinterrupts zu erreichen (Siehe Interupt.c) . Es ergibt sich folgende Interruptzuteilung:

Int 0	Reset
Int 1	NMI
Int 2	reserviert
Int 3	reserviert
Int 4	Timer 0
Int 5	Timer 1
Int 6	External Interrupt Pin 6
Int 7	External Interrupt Pin 7
Int 8	DMA Channel 0 Int
Int 9	DMA Channel 1 Int
Int 10	Emif SDRAM Timer Int
Int 11	DMA Channel 2 Int
Int 12	DMA Channel 3 Int
Int 13	Host Port Host to DSP Int
Int 14	External Interrupt Pin 4
Int 15	External Interrupt Pin 5



Das Auftreten der Interrupts wird durch einen Flagcounter registriert, der in der nk-Task Verwaltung ausgewertet wird (warint). Sollen spezielle Interrupt Service Routinen (SISR) genutzt werden muss dies zur Compilezeit bekannt sein. Sie müssen entsprechend in die Interrupt Service Tabelle (IST) eingetragen werden. (Siehe 3.4.4 Die Datei Assem.asm). Da diese Routinen außerhalb der Zeitverwaltung des Betriebssystems laufen, ist darauf zu achten, dass sie nicht zu lange Laufzeiten haben. Das bedeutet, es muss gewährleistet sein, dass k-Tasks trotz Unterbrechung durch die SISR noch in ihrer Periode beendet werden können.

3.2.4 Nachrichtensystem

Die Funktionen des Nachrichtensystems ermöglichen eine Kommunikation zwischen den einzelnen Tasks. Mit der optional nutzbaren Dual-Prozessorerweiterung des Nachrichtensystems auch über die Prozessorgrenzen hinweg. Das Nachrichtensystem ist ein optionaler Bestandteil des Betriebssystems und kann deaktiviert werden. Zum Zwischenspeichern der Nachrichten wird ein spezieller Messagepuffer angelegt, dessen Größe der Nutzer festlegen kann (um eine schnelle Suche nach den Nachrichten zu ermöglichen, wird zusätzlich ein Nachrichtenindex Array angelegt). Der Puffer wird nach dem FIFO-Prinzip verwaltet. Sollte der Messagepuffer voll sein weil Nachrichten nicht abgeholt werden, ist keine weitere Kommunikation zwischen den Tasks mehr möglich. Jede ausgetauschte Nachricht besteht aus vier 32 Bit Blöcken, wobei davon 96Bit für die Nachrichtenübertragung zur Verfügung stehen und 32 Bit Headerinformationen (Message Control Block) sind.

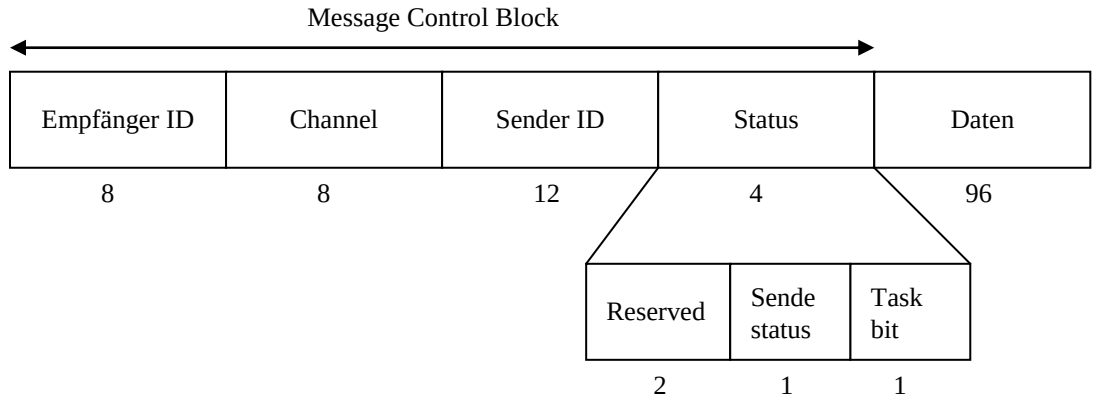


Abb. 20 Aufbau einer Nachricht im Nachrichtensystem, Quelle: [SchLev01] S.10

Die Sender und Empfänger IDs sind die entsprechenden Task IDs der Kommunikationspartner. Über die Kanäle, kann noch einmal eine feinere Adressierung des Ziels innerhalb der Tasks vorgenommen werden. Das Sendestatusbit wird gesetzt, wenn die Nachricht vollständig übertragen wurde, ist es nicht gesetzt, wird die Nachricht solange ignoriert bis es gesetzt wird.

Im Einzelprozessormodus läuft eine Kommunikation über das Nachrichtensystem grob folgend ab. Ein Task sendet über die Funktionen des Nachrichtensystems eine Nachricht an einen anderen Task über einen bestimmten Kanal. Diese Nachricht wird im Messagepuffer gespeichert. Die Tasks können mit den Funktionen des Nachrichtensystems (hier wird das Nachrichtenindexarray benutzt) prüfen, ob eine Nachricht für sie vorhanden ist, wenn Empfänger ID und Kanalnummer übereinstimmen wird die Nachricht abgeholt und danach aus dem Messagepuffer gelöscht. Es ist auch möglich, dass nk-Tasks bis zum Eintreffen einer Nachricht im Zustand „warten“ verbleiben.

Das Dualprozessornachrichtensystem baut auf das Einzelprozessornachrichtensystem auf. Das Betriebssystem unterscheidet selbständig intern zwischen internen Nachrichten, also Nachrichten die zwischen Tasks auf einem Prozessor ausgetauscht werden und externen Nachrichten, also Nachrichten die zwischen Tasks die auf verschiedenen Prozessoren laufen ausgetauscht werden. Für den Nutzertask ist die momentan genutzte Variante vollkommen transparent, er muss nicht wissen, wo der gewünschte Kommunikationspartner gerade läuft.

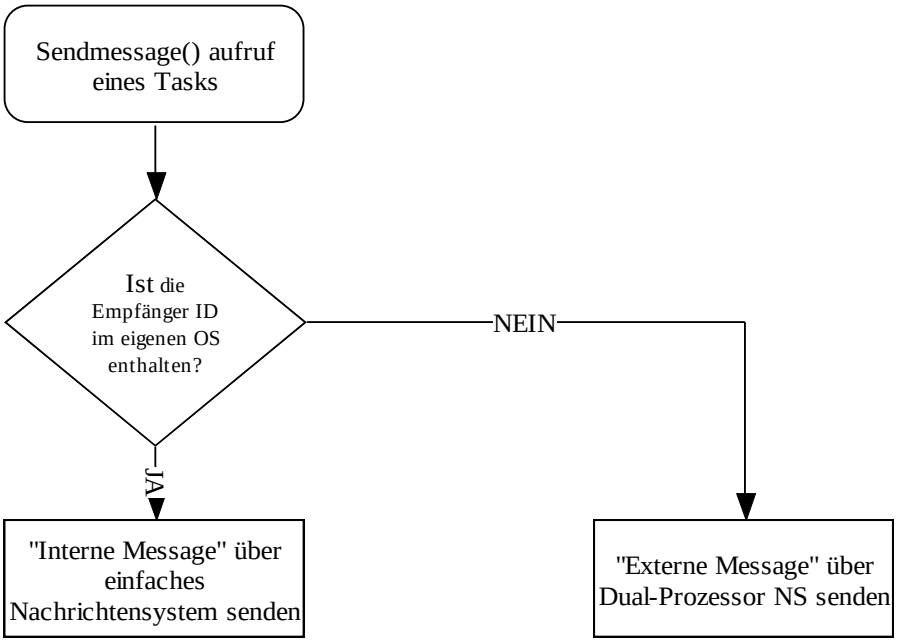


Abb. 21 Entscheidung zwischen interner oder externer Nachricht, Quelle: [Lev02]

Das Betriebssystem speichert alle TaskIDs der Tasks die auf seinem Prozessor laufen in einem Array (idpuffer). Befindet sich das Empfängertask in dieser Liste werden die Routinen der Einzelprozessorkommunikation benutzt. Ist dies nicht der Fall, handelt es sich um eine „externe Message“ die anders behandelt werden muss. Die Nachricht wird nicht im internen Nachrichtenpuffer abgelegt, sondern wird in dem Dual-Port Ram des Dualprozessorboards geschrieben. Ist dort kein Speicherplatz mehr vorhanden, wird die Nachricht verworfen. Wenn ein Task des Betriebsystems auf dem anderen Prozessor eine Nachricht abruft (askmessage()) wird der gesamte Inhalt Nachrichtenpuffer des Dual-Port Rams, der für Tasks des Betriebssystems bestimmt ist, in den internen Messagepuffer verschoben. Sollte der Platz im internen Messagepuffer dafür nicht aussreichen, verbleiben einige Nachrichten auf dem Dual-Port Ram (FIFO Prinzip) und werden beim nächsten askmessage() Aufruf verschoben, falls dann wieder Platz im internen Messagepuffer ist. Ist das integrieren in den internen Messagepuffer abgeschlossen, wird dort gesucht, ob eine Nachricht für das Task vorhanden ist. Eine Überprüfung des Dual-Port-Nachrichtenpuffers, welche Nachrichten benötigt werden findet nicht statt. Falls eine wichtige Nachricht, die das Abarbeiten der anderen Nachrichten im internen Messagepuffer voraussetzt, aus Platzmangel im Dual-Port Ram verbleibt, kann diese nie abgerufen werden und es kommt zu einem Stillstand der Kommunikation. Externe Nachrichten sind mit internen Nachrichten hinsichtlich ihres Aufbaus identisch (Vier mal 32 Bit mit identischem Headerbereich).

Um das Abrufen und Senden der Nachrichten effizient zu ermöglichen erhält der Dual-Port Ram eine spezielle Organisation. Er wird in zwei Bereiche eingeteilt. Ein Bereich in dem das eine Betriebssystem die Nachrichten ablegt die aus ihm gesendet wurden, dieser ist für das zweite Betriebssystem der Bereich, in dem sich die Nachrichten befinden, die an es gerichtet sind. Der zweite Bereich dient genau der entgegengesetzten Kommunikationsrichtung. Somit weiß jedes Betriebssystem automatisch, wo die zu sendenden Nachrichten hin verschoben werden müssen und wo sich die zu holenden Nachrichten befinden. Der Messagepuffer im Dual-Port Ram ist als Ringpuffer organisiert.
Um einen konsistenten Datenzugriff zu gewährleisten, ist der Zugriff auf die einzelnen Felder so geregelt, dass ein Betriebssystem einer CPU entweder nur Leserechte oder nur Schreibrechte auf einen bestimmten Speicherbereich hat. Das Betriebssystem der anderen

CPU hat genau entgegengesetzte Rechte (also da Leserechte, wo das andere Betriebssystem Schreibrechte hat, usw.). Ein gleichzeitiges Schreiben auf einen Speicherbereich ist somit nicht möglich und das Verwalten eines Sperrmechanismus entfällt. Dadurch, dass Anfang und Ende des Ringpuffers immer erst nach dem Schreiben/Löschen der eigentlichen Nachricht in den Speicher angepasst werden ist zum Zeitpunkt des Lesens eines Feldes immer ein gesicherter korrekter Inhalt gegeben.

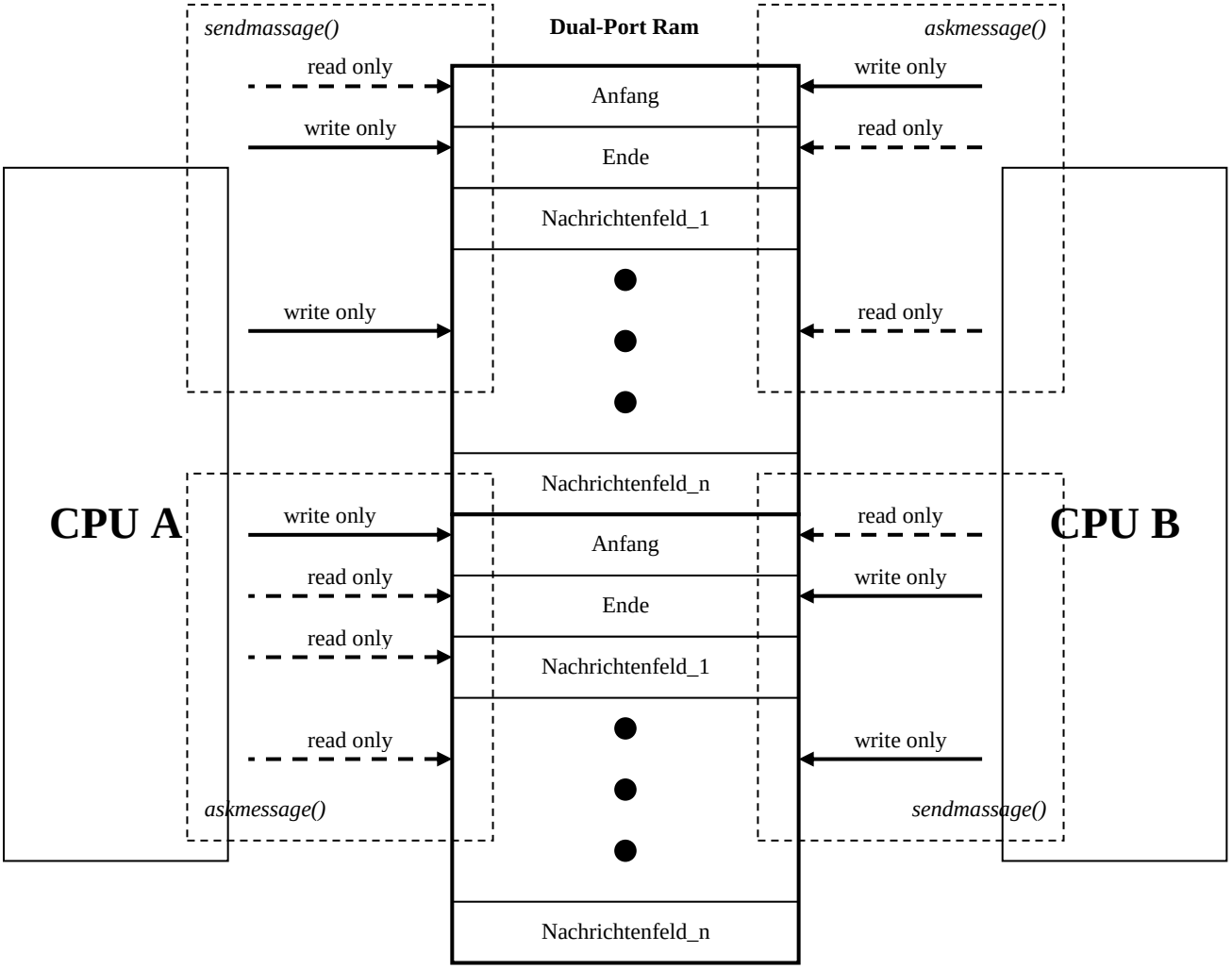


Abb. 22 Zuordnung von Lese- und Schreibrechten zur Konsistenzerhaltung, [Lev02] S.68

3.2.5 Messwertspeicher

Die Funktionen des optionalen Messwertspeichers dienen dem einfachen Erfassen anfallender Messwerte. Die Messwerte werden in der (sbsram) Ram-Erweiterung gespeichert. Dazu wird in diesem Speicher ein Ringpuffer erstellt auf den nach dem „First in First out“ Prinzip auf die einzelnen Elemente zugegriffen werden kann. Die Größe des Ringpuffers und die Größe der einzelnen Elemente kann vom Nutzer beim erstellen des Ringpuffers festgelegt werden und ist nachträglich nicht änderbar. Der Messwertspeicher ist ein optionaler Bestandteil des Betriebssystems.
Es existieren zwei unterschiedliche Implementierungen des Messwertspeichers. Einmal für das reine Erzeuger-Verbraucher-Problem, dass heißt, dass maximal ein erzeugender und ein verbrauchender Prozess für den Messwertspeicher gleichzeitig existiert und zum anderen eine Implementierung für mehrere parallele Schreib/Lese-Prozesse die Aufgrund des höheren Verwaltungsaufwandes zeitintensiver ist und mehr Speicher benötigt.

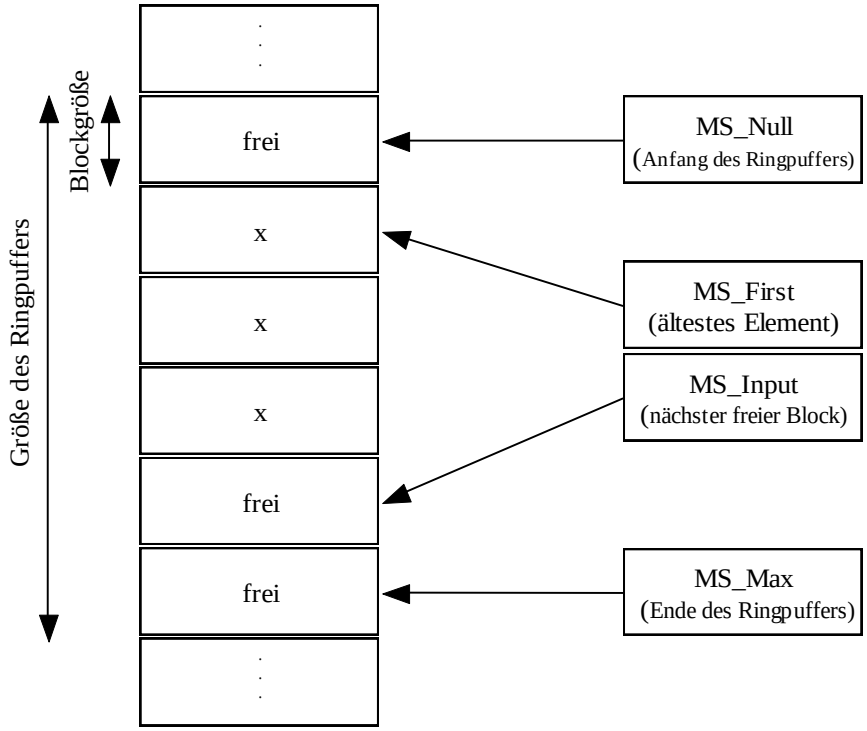


Abb. 23 Speicherstruktur des Messwertspeichers für reines Erzeuger-Verbraucher Problem.

Die Zeiger MS_Null und MS_Anfang geben die Grenzen des Ringpuffers an. Der Zeiger MS_Input zeigt auf die nächste freie Speicherzelle. Mit jedem neuen gespeicherten Element wird er um die Blockgröße erhöht und zeigt somit wieder auf das nächste freie Element. Falls er durch eine Erhöhung den Bereich des Ringpuffers verlassen würde, wird er auf MS_Null gesetzt., dadurch ergibt sich ein kontinuierliches Laufen von MS_Input durch den Ringpuffer. MS_First zeigt immer auf das älteste Element, das sich momentan im Ringpuffer befindet. Dies ist das Element das als nächstes aus dem Ringpuffer gelesen und entfernt wird (FIFO-Prinzip). Zeigt MS_Input auf das gleiche Element wie MS_First, ist der Ringpuffer vollständig gefüllt.

In der Abb. 23 wurden vier Messwerte im Messwertspeicher gespeichert, der erste Messwert wurde bereits wieder abgerufen.

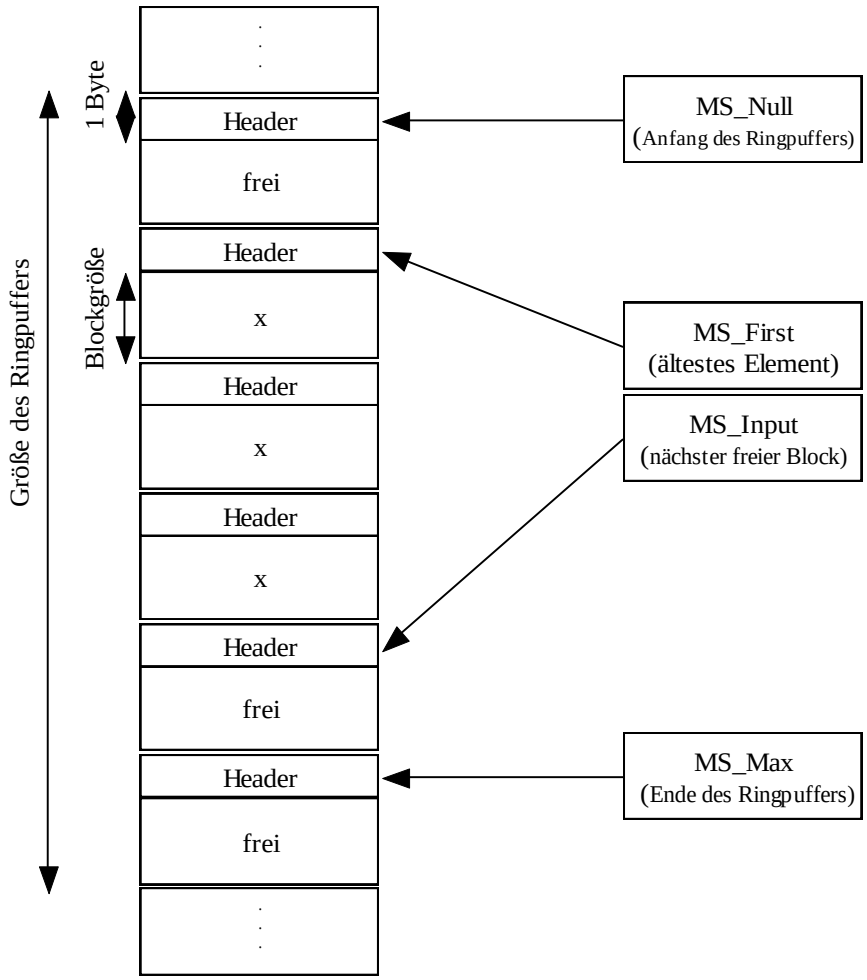


Abb. 24 Speicherstruktur des Messwertspeichers für mehrere Schreib- und Leseprozesse.

Der Messwertspeichers für mehrere Schreib- und Leseprozesse hat einen ähnlichen Aufbau wie der Messwertspeicher für nur einen Schreib und Leseprozess. Zusätzlich bekommt jeder Block einen 8 Bit Header, in dem Verwaltungsinformationen abgespeichert werden, über die der Zugriff auf das Element geregelt wird.
Kodierung der Verwaltungsinformationen (niedrigste 2 Bit):

00 - Zelle leer
01 - Zelle voll
10 - Zelle wird gerade beschrieben
11 - Zelle voll und wird gerade gelesen

Die restlichen Bits sind unbelegt.

3.2.6 Betriebsmittelverwaltung

Sie stellt Funktionen bereit, die ein einfaches Verwalten des Zugriffs auf exklusive Ressourcen (ein- oder mehrinstanzig) ermöglichen. Die Betriebsmittelverwaltung ist ein optionaler Bestandteil des Betriebssystems. Es wird nicht die Ressource selbst verwaltet, sondern eine sie stellvertretende Identifikationsnummer (BM_ID). Aus diesem Grund kann ein direktes belegen der Ressource unter Umgehung der Betriebsmittelverwaltung nicht ausgeschlossen werden. Es hat aber den Vorteil, dass beliebige Ressourcen (Speicher, Ports usw.) verwaltet werden können.

Es wurde ein Algorithmus verwendet der auf dem Peterson Algorithmus beruht (Siehe [Sch01] S. 53); Dieser hat eine sehr kurze kritische Region, einen geringen Speicherbedarf und eine kurze Laufzeit.

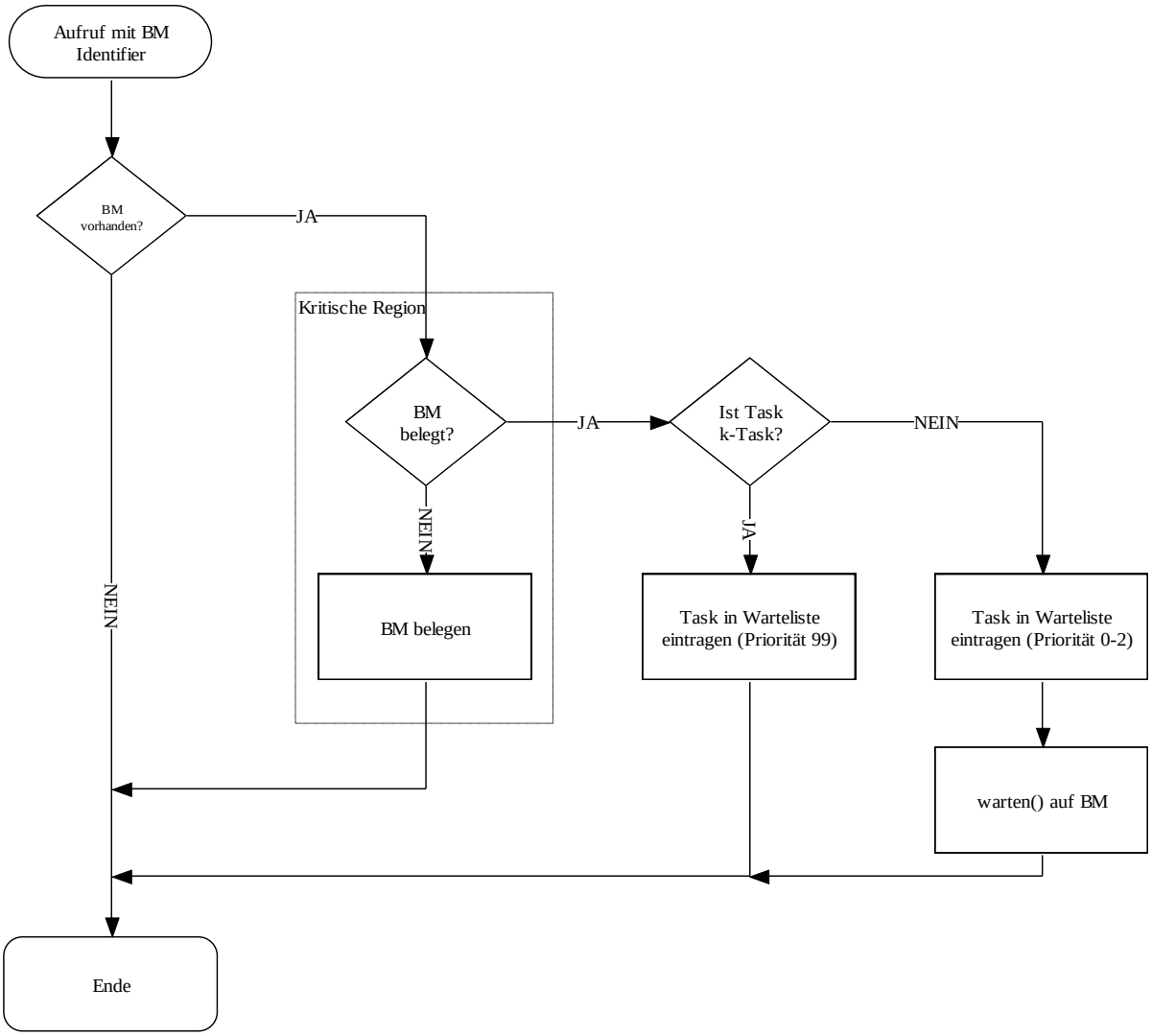


Abb. 25 Ablaufplan für die Belegung der Ressource, Quelle: [Sch01] S. 101

Der kritische Bereich besteht aus dem Laden der Sperrvariable, einem Vergleich und evtl. dem Speichern der Variable, was weniger als ca. 10 Takte benötigt.

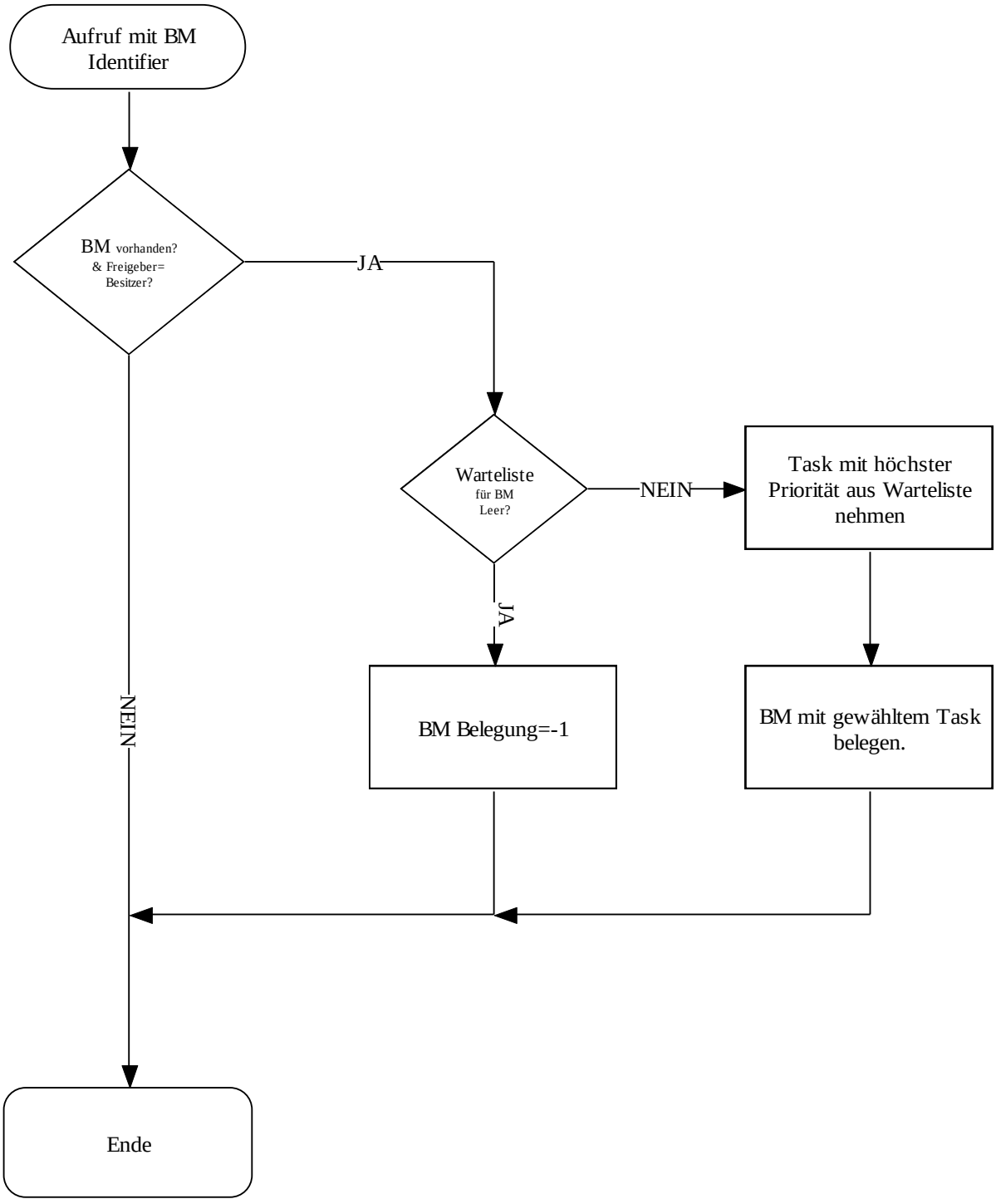


Abb. 26 Ablaufplan für Freigabe der Ressource. [Sch01] S.101

Da das Betriebsmittel entweder direkt von einem Eigentümer zum Nächsten übergeben wird oder freigegeben wird, kommt der Freigabealgorithmus ohne kritische Region aus.

Jedem Betriebmittel wird eine Warteschlange zugeordnet über die die Weitergabe der Ressource geregelt wird. Jede momentan wegen einer Belegung nicht erfüllbare Anfrage wird mit der Priorität des anfragenden Tasks in diese Warteschlange gespeichert. Beim Freiwerden der Ressource wird mit einer Maximumssuche über den Prioritäten der Task ermittelt, der die Ressource als nächstes erhält. Dabei wird die FIFO Ordnung unter gleichen Prioritäten eingehalten. Die Warteschlange ist als Liste organisiert, was ein einfaches Umordnen der Elemente bei Entfernen eines Elementes aus der Mitte ermöglicht.
Folgende Datenstruktur wird für jedes Betriebsmittel angelegt.

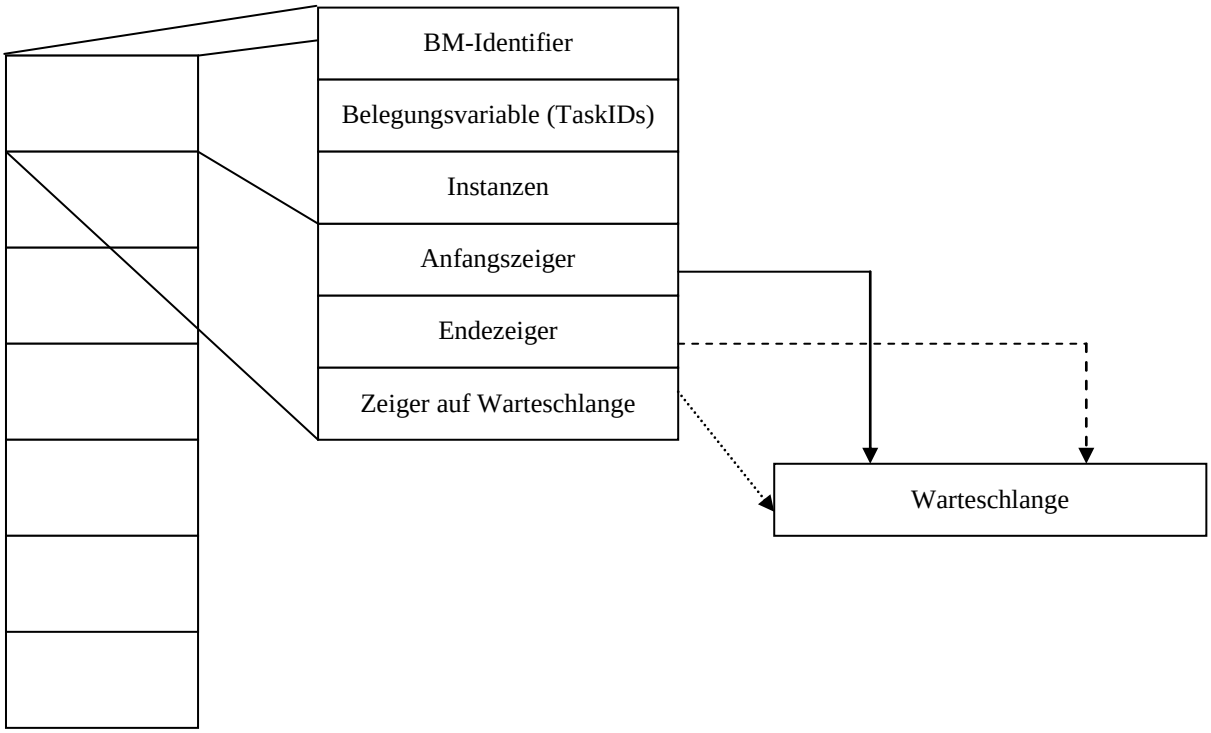


Abb. 27 Datenstruktur der Betriebsmittelverwaltung, Quelle: [Sch01] S.104

Falls ein angefordertes Betriebsmittel momentan belegt ist, wird der anfordernde nk-Task in den Zustand „Warten“ auf Betriebsmittel versetzt. Seine Ausführung wird fortgesetzt, wenn das gewünschte Betriebsmittel freigegeben wurde und er der Nächste in der Warteschlange ist. Da k-Tasks aufgrund ihrer festen maximalen Ausführungszeit und ihrer Ununterbrechbarkeit nicht auf ein Freigeben des Betriebsmittels warten können, muss der Nutzer selbst im Algorithmus des k-Tasks auf eine nicht Verfügbarkeit der Ressource reagieren. Die Anfrage wird aber in die Warteschlange für das Betriebsmittel eingetragen und die Ressource wird beim Freiwerden für das k-Task reserviert, so kann das k-Tasks beim nächsten Durchlauf darauf zugreifen. Durch ihre höhere Priorität werden K-Tasks gegenüber nk-Tasks bevorzugt bedient, deswegen kann es bei gemischten k/nk-Task Zugriffen auf eine Ressource unter Umständen zu einer langen Wartezeit für das nk-Task kommen.

3.2.7 DMA Unterstützung

Die Funktionen für die DMA Unterstützung sind ein optionaler Bestandteil des Betriebssystems. Die DMA Erweiterung des eRTOS 2.1 unterstützt das Senden von DMA Nachrichten über bis zu vier DMA Kanäle. Jeder der vier benutzbaren DMA Kanäle kann Daten zwischen internen und externen Daten- und Programmspeichern sowie zu externen E/A – Einheiten transferieren. Um den Nutzertasks das Senden von DMA Nachrichten zu ermöglichen wird ein DMA Server bereitgestellt, der die Anforderungen der Tasks entgegen nimmt und prioritätsabhängig an den DMA-Controller weiterleitet. Die Grundlage für die Kommunikation zwischen Task und DMA Server bildet das integrierte Nachrichtensystem des eRTOS. Da das interne Nachrichtensystem zur Kommunikation genutzt wird, können pro Nachricht an den DMA Server drei 32 Bit Datenworte als Nutzdaten übertragen werden. Die DMA - Nachricht an den Server hat deshalb folgenden Aufbau:

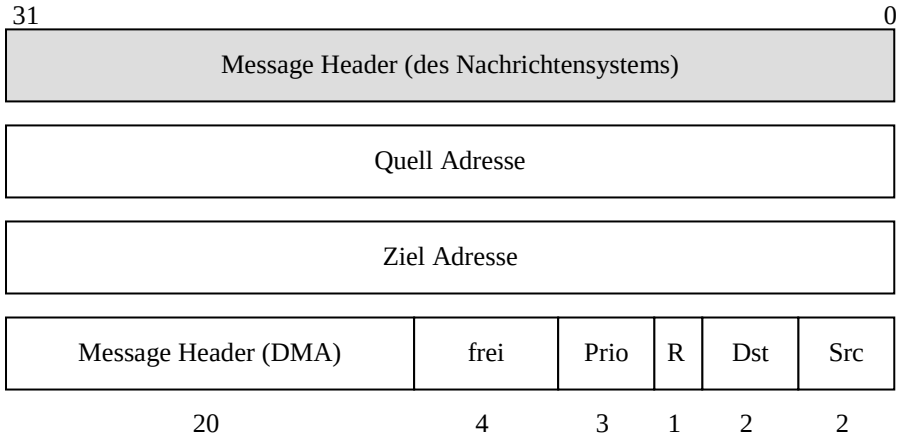


Abb. 28 Aufbau einer DMA Nachricht, Quelle: [Kra02] S. 10

Quell- und Zieladresse beschreiben die Speicherbereiche von denen gelesen, bzw. an die transferiert werden soll. Das 3. Nutzdatenwort dient der Konfiguration. Die ersten 2 Bit (Src) regeln die Behandlung der Quelladresse. Sie können folgende Werte annehmen:

Wert	Bedeutung
00	Quelladresse bleibt unverändert
01	Quelladresse wird um Blockgröße erhöht
10	Quelladresse wird um Blockgröße erniedrigt

Bit 2 und 3 (Dst) sind kodiert wie die ersten 2 Bit und regeln die Behandlung der Zieladresse. Bit 4 (R) legt fest ob der Task der die Nachricht versandt hat, nach Beenden der Übertragung eine Bestätigung (Reply) erhält. Die Priorität der Nachricht wird über Bit 5 bis 7 kodiert. Daraus ergibt sich die Möglichkeit die Priorität in 8 Stufen festzulegen. Bit 12 bis 31 kodieren die Größe des zu übertragenen Blockes, die maximale Blockgröße beträgt also 1 MByte.

Um Rechenzeit zu sparen wird der DMA-Server suspendiert, wenn kein DMA-Kanal frei ist. Dadurch ist ein zweiter „Resuspend“ Task erforderlich, der den DMA – Server wieder reaktiviert wenn ein DMA – Kanal frei wird. DMA Server und Resuspend Task sind jeweils nk-Tasks.

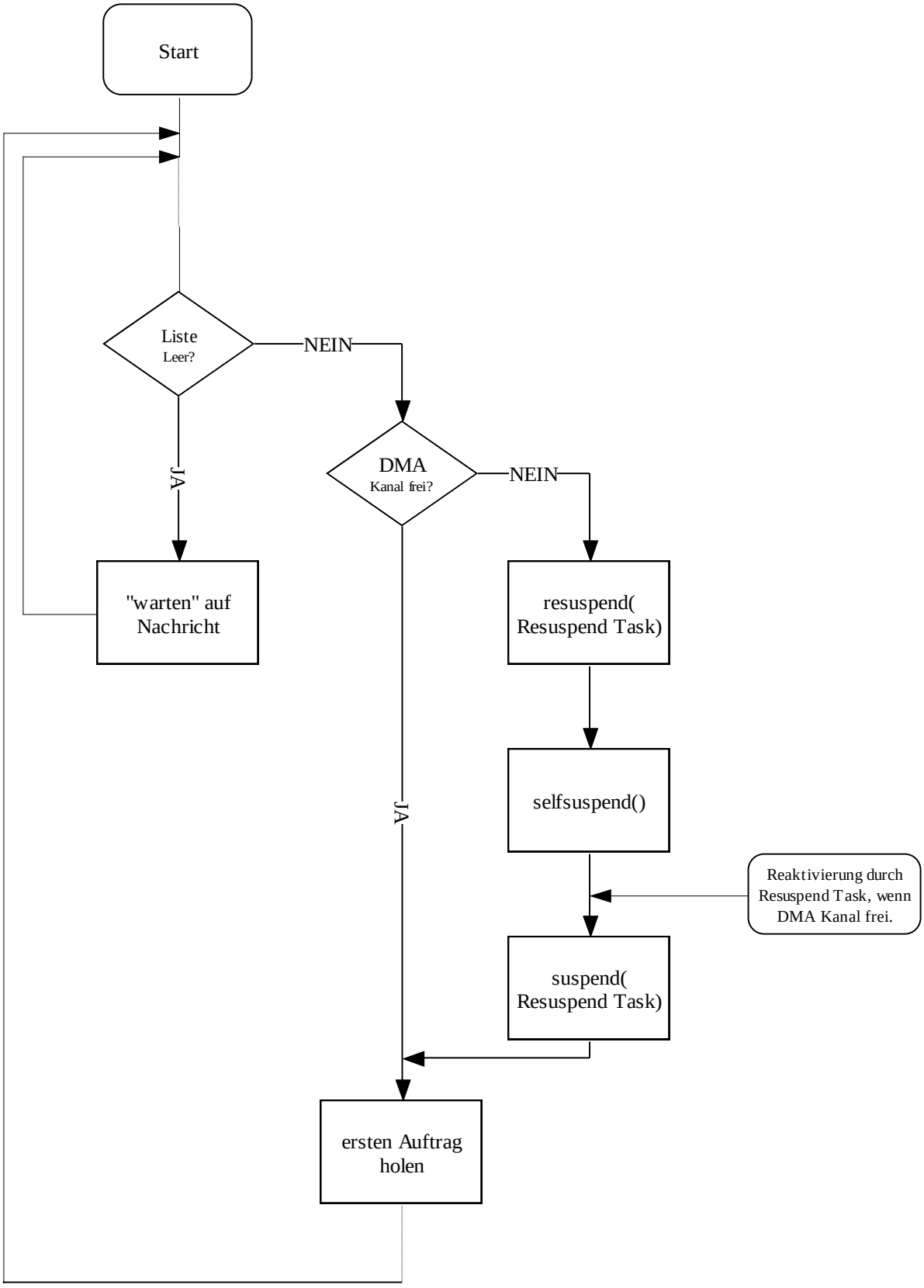


Abb. 29 Der Ablaufplan des Algorithmus für den DMA Server. [Kra02] S.18

Da ein Problem mit dem Interruptflagregister existiert, wird der Resuspendtask nicht direkt als interruptabhängiger Task erstellt. Vielmehr wird für jeden DMA Kanal eine Interrupt Service Routine (ISR) erstellt die ausgeführt wird, wenn der DMA-Controller den Interrupt für den entsprechenden DMA Kanal auslöst (wenn der Kanal frei ist). Diese Routine setzt dann die entsprechenden Flags, die vom Resuspendtask in einer Endlosschleife abgefragt werden.
Falls der DMA Kanal nun als frei gekennzeichnet ist, reaktiviert der Resuspend Task den DMA Server, der den Resuspendtask daraufhin suspendiert. Der Resuspendtask ist also indirekt von dem Interrupt abhängig. Bei den Operationen die durch die ISR durchgeführt werden, handelt es sich um wenige einfache Bitoperationen, die das Systemverhalten nicht beeinflussen sollten.

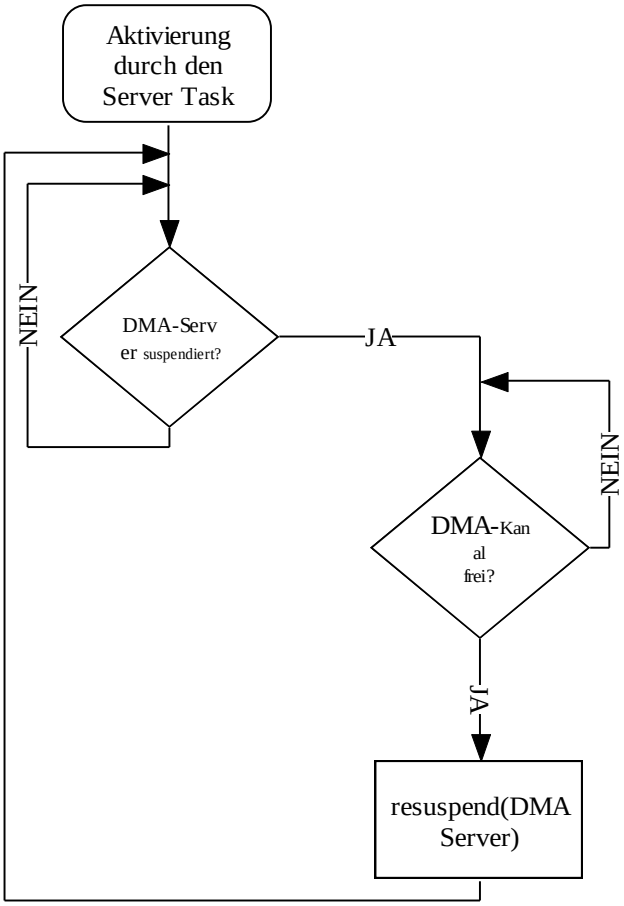


Abb. 30 Der Ablaufplan des Algorithmus für den Resuspend Task [Kra02] S.19

Weitere Informationen: [Kra02]

3.3 Fehler im eRTOS 2.0.

Im eRTOS 2.0 existierten auch einige Fehler. Wenn im weiteren eine Lösung dazu angegeben wurde, konnten diese behoben werden.

Fehlerhafte kTask-Verwaltung zeitverwalt() Quelle:[Rim02]

Fehler:
Peridode des k-Tasks ist um einen Tick länger als vom Nutzer gefordert.

Beschreibung:
Über

else startvariable=kt_table[n_task].n_aktiv-zeitcount;

wird festgelegt in wie viel Ticks der nächste k-Task bereit zur Ausführung ist.
Wenn der nächste Aktivierungszeitpunkt bei Tick 10 liegt und der aktuelle Tick 5 ist; Es müssten also noch 5 Ticks ablaufen bis zur Aktivierung. Die startvariable erhält also den Wert 5.

Über

```
if(startvariable>0)startvariable--;        // ist in diesem tick kein taskstart geplant?
        else
    {
            …
            altestes bereites Task ausführen
            …
    }
```

wird in jedem Tick die Startvariable verringert. Deswegen erhält sie erst im Tick 10 den Wert 0 der für die Ausführung des Tasks erforderlich ist (Test: startvariable>0) und im 11. Tick wird dann der k-Task ausgeführt. Somit ergibt sich für die Periode des k-Tasks immer Nutzerfestlegung+1.

Lösung:
Ändern der Berechnung der Startvariable. Um eins verkleinern.

else startvariable=kt_table[n_task].n_aktiv-zeitcount *-1;*

Fehler im Speicher-Allocator Quelle: Mail von Sebastian.Schmidt, 12 Feb 2002

Fehler:
Fehlerhafte Headerkonstruktion beim Erzeugen neuer Seiten in Funktion simple()

Lösung:
wurde von Sebastian Schmidt geliefert und ist eingefügt.

Fehler im Speicher-Allocator Quelle: [Rim02] S.97

Fehler:
Bei einer ungeraden Anzahl der über die Funktion nAlloc() angeforderten Bytes wird zu wenig Speicher reserviert, wenn mehr als 4 Byte angefordert werden.

Beschreibung:
Bei einer Anfrage über 4 Byte wird durch

anfrage=anfrage/4+1;

die Anzahl der 32 Bit Zellen berechnet die benötigt werden um die Anforderung zu erfüllen. Anfrage durch vier Datenzellen plus eine Headerzelle. Bei einer ungeraden Anforderung, z.B. fünf werden aber nur insgesamt 2 Bit Zellen reserviert wegen der Integer Division, also eine zu wenig (4 Byte Daten + 4 Byte Header). Die Größe des reservierten Speichers entspricht also nicht der Anforderung.

Lösung:
Änderung der Berechnung der Zellmenge:

```
if (anfrage%4!=0) hilf=1; else hilf=0;
anfrage=anfrage/4+1+hilf;
```

Fehler in warten Funktion Quelle: Bernd Däne, 29.05.02

Fehler:
In manchen Fällen startet ein wartendes nk-Task von vorn anstatt an der unterbrochenen Stelle fortzusetzen. Zusätzlich kann es bei häufigem Aufruf von Warten zu Fehlfunktionen im eRTOS kommen.

Lösung:
keine

Fehler in Funktion delete nk-Task. Quelle: [Lev02], Projektfile

Fehler:
Nk-Tasks werden nicht aus dem Scheduler entfernt

Lösung:
keine

Fehler in der Funktion suspend() Quelle: [Rim02]

Fehler:
Wenn suspend() auf einen bereits suspendierten nk-Task angewendet wird, kann es zu einer Beeinträchtigung der Ausführung anderer nk-Tasks kommen.

Beschreibung:

Die Variable anz_wl[] enthält die Anzahl der nichtsuspendierten Tasks für die einzelnen Prioritätsstufen. Mit jedem Aufruf von suspend() auf einen existierenden Task wird diese Anzahl für die Prioritätsstufe des Tasks verringert.

```
if (id==akttask) selfsuspend();
else
{
// Task finden
if (hilf==nktask){ …}
else
{
…
anz_wl[nkfunk[hilf].priority]--;
}
}
```

Sobald diese Anzahl den Wert 0 oder kleiner erreicht hat, wird die entsprechende Prioritätsstufe beim Scheduling nicht mehr berücksichtigt.

```
while((hilf<3)&&(g_task==(struct nkt *)-1))   // suche solange bis task gefunden oder listen abgesucht
{
        if(anz_wl[hilf]>0)
    {
    //Suche nach Ausführbaren nk-Task in der Prioritätsstufe
    }
…
}
```

Lösung:
In die suspend() Funktion wird eine zusätzliche Abfrage eingefügt.

```
if (hilf==nktask){ …}
else
if (nktfunc[hilf].status!=NKS_SUSPENDIERT)
{
…
anz_wl[nkfunk[hilf].priority]--;
}
```

Fehler im Messwertspeicher

Fehler:
Der Messwertspeicher für das reine Erzeuger-Verbraucher-Problem funktioniert aufgrund von einem Syntaxfehler nicht. Zusätzlich wird immer ein Element mehr im Messwertspeicher verwaltet als vom Nutzer angefordert obwohl nur für die angeforderte Menge Speicher reserviert wird. Der Messwertspeicher für mehrere Schreib- und Leseprozesse hat denselben Fehler. Zusätzlich überschreibt in diesen Modus der Header der Elemente die Inhalte anderer Elemente.

Beschreibung:
Das Problem, das ein Element zu viel verwaltet wird, liegt darin, dass ab Adresse ms_Null Elemente gespeichert werden und somit die Elementanzahl nicht dem Index entspricht, sondern korrekt mit Index+1 berechnet werden muss.

Dass der Header die anderen Elemente teilweise überschreibt liegt daran, dass in der Zeigerarithmetik nicht beachtet wurde, dass der Header mit verwaltet werden muss.

Lösung:
Für das Problem der inkorrekten Elementanzahl ist die Lösung relativ simpel, ms_Max muss einfach um ein Element verringert, werden um die korrekte Anzahl mit dem alten Algorithmus zu verwalten.

ms_Max=ms_Null+groesse-ms_BlkGroesse;

Das Problem des Headers ist etwas komplexer. Da die C-Zeigerarithmetik typabhängig ist. (Vgl. [Rim02] S. 99) Um die Headerinformation auf das 1. Byte zu beschränken ist wesentlich mehr Aufwand nötig als für die einfache Lösung, den Header einfach auf 32 Bit auszudehnen. Da dynamisch sowieso nur vielfache von 32 Bit reserviert werden können, wenn die Größe der Anforderung 32 Bit übersteigt, ist das kein Nachteil.
Um dies zu erreichen, muss die Menge des allocierten Speichers verändert werden.

ms_Null=(int *)nAlloc((groesse+ms_Zellen)*4, MW_SART);

Zusätzlich muss ms_Blkgroesse um eins erhöht werden wegen des integrierten Headers.

ms_BlkGroesse=blk_groesse+1;

der Aufruf der ms_copy() Funktion muss entsprechend angepasst werden, damit der Header vom Elementinhalt getrennt werden kann.

ms_copy(input_old+1, zeiger, ms_BlkGroesse-1)

Fehler in der Betriebsmittelverwaltung

Fehler:
Die Betriebsmittelverwaltungsfunktion funktioniert nicht. Occupy_bm() liefert immer eine freie Ressource zurück.

Beschreibung:
Eine ausführliche Beschreibung der Fehler und Änderungen wäre zu umfangreich, sie ist aber im Quellcode kommentiert.
Eine kurze Zusammenfassung:

- In nktaskver() wurden die RessoucenID mit der nk-TaskID verglichen, richtig ist zu vergleichen, ob die Ressource diejenige ist auf die das Task wartet.
- In nktaskver() wurden zwei unterschiedliche Hilfsvariablen verwendet, obwohl nur die Verwendung einer Sinn macht.
- In occupy_bm() wurde beim Test auf Eigentümer auf freie Instanzen getestet
- In occupy_bm() erfolgte die Zuweisung der Belegung nur für kTasks nicht aber für nk-Tasks
- In occupy_bm() wurde beim Zufügen von k-Tasks in die Warteliste für das Betriebsmittel versäumt den Endezeiger zu verschieben.
- In free_bm() wurde nur bei der Prüfung, ob das aufrufende Task das Betriebsmittel freigeben darf, nur für k-Tasks getestet.
- In free_bm() wurde bei der Ermittlung welches Task das Betriebsmittel als nächstes belegen darf nicht das Task mit der maximalen Priorität, sondern der laufende Zähler zugewiesen.

Lösung:
Siehe Quellcode.

Fehler im Dualprozessormessagesystem

Fehler:
Wenn sich ein nk-Task im Zustand „warten auf Nachricht“ befindet und diese Nachricht von einem Task auf dem anderen Prozessor kommen muss, ist es vom zufälligen Aufruf von askmessage() durch einen anderen Task abhängig, ob das Task die Nachricht erhält.

Beschreibung:
Testmsg() und testallmsg() überprüfen nicht die Nachrichten, die auf dem Dualportram vorhanden sind weil sie nicht die Funktion leermpram() aufrufen, die die Nachrichten auf dem Dualportram in den internen Messagepuffer integrieren.

Lösung:
Ein Aufruf der Funktion leermpram() in testmsg() und testallmsg().

```
#ifdef MPRAM
    leermpram();
#endif
```

Fehler in der Funktion end_int()

Fehler:
Nk-Tasks mit derselben Priorität wie der Interrupt Task werden nach einiger Zeit nicht mehr ausgeführt, wenn gleichzeitig k-Tasks im System vorhanden sind.

Beschreibung:
Mit jedem Aufruf von end_int() wird auch die Variable anz_wl[] die die Anzahl der ausführbaren Tasks jeder Priorität enthält um eins verringert. Dies scheint im Zusammenspiel mit der Verdrängung durch k-Tasks zu oft zu geschehen, solange bis diese Prioritätsstufe im Scheduling nicht mehr berücksichtigt wird.

Lösung:
keine

Fehler in der Funktion chng_prio() und delete_nktask()

Fehler:
Beim Aufruf der Funktion chng_prio() oder delete_nktask() hängt das System.

Beschreibung:
Um die Wartelisten für die jeweilige Priorität zu ändern wird die Zeitverwaltung des Betriebsystems mit DISABLE_TIME unterbrochen. In der while Schleife

```
while(ende)
        {

                /*umsortieren der Warteliste*/

        }
```

wird das umsortieren der Warteliste vorgenommen. Es wurde aber vergessen, dass die Zählvariable der Schleife hilf2 bei jedem Durchgang erhöht werden muss. Somit bleibt das System in der Schleife hängen und da das ENABLE_TIME für die das Reaktivieren der Zeitverwaltung nie erreicht wird, hängt das ganze eRTOS.

Lösung:
Erhöhung der Zählvariable hilf2.

```
while(ende)
        {

                /*umsortieren der Warteliste*/

        hilf2++;
        }
```

Fehler beim setzen des Bits für den DMA Interrupt im Interrupt Flag Register, Quelle: [Kra02] S. 15

Fehler:

Dies ist vermutlich kein Fehler des eRTOS sondern ein Fehler der Hardware. Obwohl der DMA-Transfer fertig war und der Interrupt ausgelöst wurde, wird das zugehörige Bit im Interrupt Flag Register nicht gesetzt.

Lösung: keine

Folgende Funktionen sind nicht implementiert:

Die System Speicherallocatoren **Sysalloc(), Sysfree().** Sie sollten in einem definierten Speicherbereich dessen Größe und Aufteilung vorher bekannt war (Anzahl der möglichen Tasks) Speicher für neue Task Controll Blöcke allocieren. (Siehe [Sch01] S. 65 Stattdessen wird nktfunc als Array statisch im Speicher definiert, dessen einzelne Elemente die Task Controll Blöcke sind.

Die Funktion **ms_first(int *zeiger)** für den Modus für mehrere Schreib- und Leseprozesse. Das erste Element im Messwertspeicher sollte an die Adresse, die Zeiger angibt, kopiert werden. In diesem Modus wird generell -1 zurückgeliefert.

Aus meiner Beobachtung heraus, ist es generell nicht zu empfehlen, Befehle, die den Status von Tasks verändern, in schneller Abfolge bzw. häufig zu verwenden, da es dadurch oft zu unvorhersehbaren Fehlfunktionen im eRTOS kommen kann.

3.4 Implementierung

3.4.1 Änderungen zur Version eRTOS 1.4.

In Version 1.4. werden die nicht kooperierenden Tasks einem speziellen Array Task zugewiesen. Die Interruptroutinen wurden durch direkten Zugriff auf die Variable nktfunc des eRTOS Betreibssystems erstellt. Die Einstellung der Perioden und Startzeitpunkte der Tasks erfolgte in der CONFIG.H über die Feldvariable ptick.
In der Version 2.0. werden diese Informationen alle direkt mit der Funktion create_nktask übergeben. (Siehe Taskverwaltung im Abschnitt Implementierung)
In Version 1.4. muss die genaue Anzahl der benutzten k und nk-Tasks angegeben werden. In Version 2.0 wird eine Obergrenze für Anzahl gleichzeitig genutzter Tasks in der CONFIG.H angegeben. Die Parameter der Wartenfunktion haben sich geändert.

3.4.2 Allgemeiner Aufbau eines Projektes

Ein eigenes Projekt muss eine Datei enthalten die die Mainroutine enthält. Die Mainroutine muss folgendem Aufbau entsprechen.

```
#include rtos.h
#include config.h

Main()
{
//Betriebssysteminitialisierung
rtos_init();

/* Initialisierungsteil*/
/*
        Hier können Nutzerinitialisierungen, Taskzuweisungen und
        Interuptaktivierungen stehen.
*/

//Start des Schedulers
OSystem();
}
```

In dieser Datei müssen mindestens die beiden Header Dateien rtos.h (enthält Grundlegende Definitionen) und config.h eingebunden werden. Über die Datei config.h kann das Betriebssystem an unterschiedliche Erfordernisse angepasst werden.

Grundsätzlich muss dabei über **#define ticklänge L** die Zeitbasis festgelegt werden. Mit der Geschwindigkeit von CPU Takt dividiert durch vier wird intern von 0 bis L gezählt, danach wird ein Timer 0 Interrupt ausgelöst und der interne Tickzähler wird um eins erhöht. Für die Ticklänge sollten Werte zwischen 100 und 600 gewählt werden. Je kleiner die Ticklänge gewählt wird umso häufiger wird die Zeitverwaltung gestartet, die ihrerseits Rechenzeit benötigt. Die nk-Tasks werden auch häufiger unterbrochen. Eine zu große Ticklänge verschlechtert das Echtzeitverhalten des Systems da die periodischen k-Tasks seltener ausgeführt werden. Für eine genauere Untersuchung des Einflusses der Ticks auf das Systemverhalten siehe [Lev00].
Über **#define ktasks n** bzw. **#define nktasks m** muss die Anzahl der Tasks angegeben werden. Aus Optimierungsgründen sollte die Anzahl der angegebenen Tasks mit der tatsächlich verwendeten Anzahl übereinstimmen. Es ist aber ohne weiteres möglich hier größere Werte einzutragen. Es sei denn, es wird das Dual-Prozessor Nachrichtensystem verwendet. Hier muss die genaue Anzahl der Tasks angegeben werden, weil andernfalls die sendmessage() Funktion die TaskIDs nicht korrekt auswerten kann. Falls das D-Module Board genutzt wird, kann dies über **#define D_MODULE** angegeben werden.
Zusätzlich kann zwischen den Prozessortypen c67 und c62 gewählt werden. Wenn **#define c67** definiert wurde nutzt das Betriebssystem die erweiterten Möglichkeiten des TMS320C67. Falls define c67 gesetzt wurde muss zusätzlich in ASSEM.ASM im Bereich Nutzerkonfiguration „**c67 .set 1“** stehen, sonst „**c67 .set 0“**.

Daraus ergibt sich folgender grundlegender Aufbau der config.h

```
#define c67                 // optimierte routinen fuer
                            // c67 verwenden, sonst fuer c62

#define D_MODULE            // Basisboard mit Bios wird verwendet

#define ktasks  3           // maximale Anzahl der ktasks
#define nktasks 4           // maximale Anzahl der nktasks

#ifdef IS_RTOS
#define ticklange 600       // Anzahl Takte pro tick
#endif

*/
Hier folgen weitere Definitionen falls weitere Komponenten des eRTOS genutzt
werden sollen
*/
```

3.4.3 Die Linker.cmd Datei

Das Linker.cmd File gibt vor wie der Compiler den zur Verfügung stehenden Speicher zuordnen soll bzw. wo sich die verschiedenen Speicherarten im Adressraum befinden.

Der Bereich MEMORY {} gibt die physische Aufteilung des Speichers an mit den Startadressen der einzelnen Bereiche und deren Länge.

Die Adressen von BIOS, IST, PRAM, DRAM0 befinden sich im Speicherblock 0 des Prozessors. DRAM1 befindet sich im Speicherblock 1 des Prozessors während SBSRAM den Adressbereich des externen Speicherbereichs angibt. MPRAM gibt den Adressbereich des Dual-Port Rams des Daytona Boards an.

Der Bereich SECTIONS{}gibt an, wo einzelne Programmteile oder Daten abgelegt werden.

- **.text:** ausführbarer Code, Konstanten
    - der gesamte ausführbare Code des Betriebssystems und des Nutzerprogramms

- **.cinit:** Tabellen für initialisierte Variablen und Konstanten
    - Variablen des Betriebssystems und des Nutzerprogramms

- **.const:** Zeichenketten, Gleitkommakonstanten, explizit definierte Konstanten
    - Konstanten aus dem Nutzerprogramm (das eRTOS enthält keine mit *const* definierten Konstanten)

- **.stack:** Systemstack
    - vom C standardmäßig verwendet, Länge 400hex

- **.switch:** Sprungtabellen für große *switch* Anweisungen
    - nicht verwendet, deshalb auch nicht initialisiert (Länge 0)

- **.tables:** keine Informationen dazu, Inhalt unbekannt
    - nicht verwendet, deshalb auch nicht initialisiert (Länge 0)

- **.data:** wird vom C-Compiler nicht benutzt, nur vom Assembler. Es besteht aber die Möglichkeit mittels Pragma DATA_SECTION Objekte darin zu plazieren.
    - nicht verwendet, deshalb auch nicht initialisiert (Länge 0)

- **.bss:** Globale und statische Variablen. Je nach Initialisierungsmodell werden Daten aus .cinit (Möglicherweise ROM) hierher kopiert.
    - statische Variablen des eRTOS und des Nutzerprogramms

- **.sysmem:** Heap für *malloc* Funktionen.
    - es werden keine derartigen Funktionen verwendet, deshalb auch keine Initialisierung (Länge 0)

- **.cio:** keine Informationen dazu, Inhalt unbekannt
    - nicht verwendet, deshalb auch nicht initialisiert (Länge 0)

- **.far:** Globale und statische Variablen, die *far* deklariert wurden
    - Nutzung durch die rts6701 Library

- **.test:** IST und eRTOS Interrupt Routinen
    - Nutzung durch ISR des eRTOS

Quelle: [Sch01] S.63

Hinweis:
Der Stack und das Array nktfunc müssen im internen Speicher liegen, da sonst die geschwindigkeitsoptimierten Betriebssystemfunktionen nicht funktionieren.
Die verschiedenen Versionen der Linker.cmd Datei

```
/* Linker Command File for Memory MAP1 für D_Modul*/

MEMORY
{
   BIOS  : org = 0x00000000, len = 0x00000c00  /*   3K int Prog-RAM */
   IST   : org = 0x00001000, len = 0x00000300  /*   8.11. ss       */
   PRAM  : org = 0x00001300, len = 0x0000ed00  /*   8.11. ss       */
   DRAM0 : org = 0x80000000, len = 0x00008000  /*  32K int Data Blk0*/
   DRAM1 : org = 0x80008000, len = 0x00008000  /*  32K int Data Blk1*/
   SBSRAM: org = 0x00400000, len = 0x0007FF80  /* 512K ext SBSRAM   */
   SDRAM : org = 0x03000000, len = 0x01000000
}

SECTIONS
{
   .text  > PRAM
   .cinit  > SBSRAM
   .nutzer > SBSRAM
   .puffer > SBSRAM
   .const  > DRAM1
   .stack  > DRAM1
   .switch > DRAM1
   .tables > DRAM1
   .data   > DRAM1
   .bss    > DRAM0
   .sysmem > DRAM1
   .cio    > DRAM1
   .far    > DRAM1
   .test   > IST
   .nprog         > PRAM
   .nvar> DRAM1
   .slowprg       > SBSRAM       /* Programmteile für langsamen Speicher (Init)*/
   .startup       > PRAM
}
```

```
/* Linker Command File for Memory MAP1  für Daytona*/

MEMORY
{

   PRAM  : org = 0x00000000, len = 0x001000
   IST   : org = 0x00001000, len = 0x0003ff      /* 8.11. ss */
   PRAM  : org = 0x00001400, len = 0x00ed00      /* 8.11. ss */
   DRAM0 : org = 0x80000000, len = 0x010000  /*  32K int Data Blk0 */
   SBSRAM: org = 0x00400000, len = 0x160000  /* 512K ext SBSRAM   080000 */
   MPRAM:   org = 01740000h   l = 00008000h  /*    32 Kbytes on Daytona    */
   SDRAM:   org = 02000000h   l = 01000000h  /* 16 Mbytes - aus ftlink.cmd*/
   PEM:  org = 03000000h   l = 01000000h  /* 16 Mbytes - aus */
}

SECTIONS
{
   .text  > PRAM
   .cinit  > DRAM0
   .nutzer > DRAM0
   .puffer > DRAM0
   .const  > DRAM0
   .stack  > DRAM0
   .switch > DRAM0
   .tables > DRAM0
   .data   > DRAM0
   .bss    > DRAM0
   .sysmem > DRAM0
   .cio    > DRAM0
   .far    > DRAM0
   .test   > IST
   .nprog > PRAM
   .nvar> DRAM0
}
```

3.4.4 Die Datei Assem.asm

In der Datei Assem.asm befinden sich die geschwindigkeitsoptimierten Assemblerroutinen der Zeitverwaltungsfunktion und die Behandlung der Interrupts wird hier gesteuert. Wenn spezielle Interrupt Service Routinen (SISR) erstellt werden sollen, müssen diese in die Interrupt Service Tabelle (IST) eingefügt werden. Für SISR stehen pro Interrupt 8 Befehle zur Verfügung. Durch einen Sprung an das Ende der Tabelle ist aber das Ausführen weiterer Befehle möglich. Es ist aber darauf zu achten, dass das Zeitverhalten des k-Tasks nicht beeinträchtigt wird und sie trotz Interrupts in ihrer Periodenzeit ausgeführt werden können. Um eine Abarbeitung in optimaler Geschwindigkeit zu gewährleisten, muss nach dem Sprung aus der IST ein NOP 5 stehen [TIB]. Nach dem Abarbeiten der Befehle muss ein Rücksprung zum Interrupt Return Point (IRP) erfolgen. Dies wird durch einen Aufruf von _endint erreicht. Die nutzint Funktionen setzen die Flags für die dynamische Interruptverwaltung im eRTOS. Damit die Interrupts für die dynamische Verwaltung genutzt werden können, muss ihre Überwachung aktiviert werden.

```
…
;Nutzerinterrupts ( 1->aktiv, 0->kein Interrupt )
nint5 .set 1
nint6 .set 0
nint7 .set 1
nint8 .set 0
nint9 .set 0
nint10 .set 0
nint11 .set 0
nint12 .set 0
nint13 .set 0
nint14 .set 0
nint15 .set 0
…
```

Die Interrupt Service Tabelle, hier können die SISR eingetragen werden.

```
;----------------------IST-----------------------------------
        .sect  ".test"
        b               _c_int00        ; reset
        nop             5
        nop             1
        nop             1
        nop             1
        nop             1
        nop             1
        nop             1
;-------------------------------------------------------------
        nop             1                       ; nmi
        nop             1
        nop             1
        nop             1
        nop             1
        nop             1
        nop             1
        nop             1
;-------------------------------------------------------------
        nop             1                       ; reserved
        nop             1
        nop             1
        nop             1
        nop             1
        nop             1
        nop             1
        nop             1
;-------------------------------------------------------------
        nop             1                       ; reserved
        nop             1
        nop             1
        nop             1
        nop             1
        nop             1
        nop             1
        nop             1
```

```
;-------------------------------------------------------------
        b               _inter4                 ; interrupt 4
        nop             1
        nop             1
        nop             1
        nop             1
        nop             1
        nop             1
        nop             1
;-------------------------------------------------------------
        nutzint nint5,_inter5,0                 ; interrupt 5
;-------------------------------------------------------------
        nutzint nint6,_inter6,4                 ; interrupt 6
;-------------------------------------------------------------
        nutzint nint7,_inter7,8                 ; interrupt 7
;-------------------------------------------------------------
        b           _dma0_isr                   ; interrupt 8
        nop             1
        nop             1
        nop             1
        nop             1
        nop             1
        nop             1
        nop             1

;-------------------------------------------------------------
        b           _dma1_isr                   ; interrupt 9
        nop             1
        nop             1
        nop             1
        nop             1
        nop             1
        nop             1
        nop             1

;-------------------------------------------------------------
        nutzint nint10,_inter10,20              ; interrupt 10
;-------------------------------------------------------------
        b           _dma2_isr                   ; interrupt 11
        nop             1
        nop             1
        nop             1
        nop             1
        nop             1
        nop             1
        nop             1

;-------------------------------------------------------------
        b           _dma3_isr                   ; interrupt 12
        nop             1
        nop             1
        nop             1
        nop             1
        nop             1
        nop             1
        nop             1

;-------------------------------------------------------------
        nutzint nint13,_inter13,32              ; interrupt 13
;-------------------------------------------------------------
        nutzint nint14,_inter14,36              ; interrupt 14
;-------------------------------------------------------------
        nutzint nint15,_inter15,40              ; interrupt 15

;-------------------------------------------------------------
;-------hier können erweiterte isr drangehängt werden-----
```

3.5 Befehle

3.5.1 DMA Funktionalität

Um die DMA Funktionalitäten nutzen zu können, sind folgende Schritte nötig:

- Die Dateien DMA.C und DMA.H müssen in das Code-Composerprojekt eingebunden werden.
- In der Datei CONFIG.H muss **#define DMA** definiert werden und die Anzahl der **nktasks** muss um **2** erhöht werden. (Für Server und Resuspend Task)
- Die Datei ASSEM.ASM muss folgendermaßen abgeändert werden:

  Im oberen Bereich müssen die Routinen für die 4 DMA Kanäle als globale Funktionen eingetragen werden.

```
…
.global _restoregc62
.global _restoregc67
.global _selfreg
.global _dma0_isr
.global _dma1_isr
.global _dma2_isr
.global _dma3_isr
…
```

  Danach muss der „Branch“ auf den Code der jeweiligen Funktion eingetragen werden. Die einzelnen DMA Kanäle haben folgende Interrupt Kennung:

  DMA0: 01000 (8)
  DMA1: 01001 (9)
  DMA2: 01010 (10)
  DMA3: 01011 (11)

  Laut INTERUPT.C werden sie deshalb im eRTOS folgenden Interrupts zugeordnet:

  DMA0: 8
  DMA1: 9
  DMA2: 11
  DMA3: 12

Daraus ergibt sich folgende Änderungen der Sprungtabelle in der Datei ASSEM.ASM:

```
...
;-------------------------------------------------------
        nutzint nint7,_inter7,8 ; interrupt 7
;-------------------------------------------------------
        b       _dma0_isr       ; interrupt 8
        nop         1
        nop         1
        nop         1
        nop         1
        nop         1
        nop         1
        nop         1

;-------------------------------------------------------
        b       _dma1_isr       interrupt 9
        nop         1
        nop         1
        nop         1
        nop         1
        nop         1
        nop         1
        nop         1

;-------------------------------------------------------
        nutzint nint10,_inter10,20      ; interrupt 10
;-------------------------------------------------------
        b       _dma2_isr       ; interrupt 11
        nop         1
        nop         1
        nop         1
        nop         1
        nop         1
        nop         1
        nop         1

;-------------------------------------------------------
        b       _dma3_isr       ; interrupt 12
        nop         1
        nop         1
        nop         1
        nop         1
        nop         1
        nop         1
        nop         1

;-------------------------------------------------------
        nutzint nint13,_inter13,32      ; interrupt 13
;-------------------------------------------------------
...
```

- In der Hauptdatei in der sich die Main() Routine befindet muss die Headerdatei DMA.H über #include eingebunden werden.
- **init_dma()** muss ausgeführt werden.

void init_dma (int dma_Server_id, int dma_Resuspendtask_id, int Channelflags)

Beschreibung:

Initialisiert die DMA Funktionen und bereitet deren Verwendung vor indem der DMA_Server Task und der DMA_Resuspend Task gestartet wird. (Siehe Kapitel DMA)

Parameter:

int dma_Server_id : Hier wird die TaskID des Servertasks angegeben (>=100)
int dma_Resuspend_id : Hier wird die TaskID des Resuspendtasks angegeben (>=100)
int Channelflags: Hier wird festgelegt welche DMA Kanäle benutzt werden können Es stehen die definierten Konstanten USE_DMA0 bis USE_DMA3 zur Verfügung. Um mehrere Kanäle gleichzeitig benutzen zu können, sind die Konstanten ODER zu verknüpfen. (USE_DMA0|USE_DMA2)

Rückgabewert: void

Hinweis:

Dieser Befehl muss im Initialisierungsbereich des eRTOS (Main() Routine) der Hauptdatei verwendet werden.

Beispiel:
init_dma (101,102, USE_DMA0|USE_DMA3)
(Siehe Beispiel zur DMA Nachrichtenübertragung)

void sendDMAMessage (int SourceAddr, int DestinationAddr, int Bytes, int Flags, int Priorität)

Beschreibung:

Sendet einen DMA Auftrag an den DMA Server.

Parameter:

SourceAddr: Hier wird der Speicheradresse angegeben, deren Inhalt transferiert werden soll. Die Variable muss global definiert werden.

DestinationAddr: Hier wird die Speicheradresse angegeben zu der transferiert werden soll. Die Variable muss global definiert werden.

Bytes: Hier wird festgelegt, wie viel Bytes übertragen werden sollen.

Flags: Hier wird festgelegt, wie Quell- und Zieladresse während der Übertragung zu behandeln sind.
Möglichkeiten:

- dma_source_const : Quelladresse nicht erhöhen
- dma_source_inc: Quelladresse erhöhen
- dma_source_dec: Quelladresse verkleinern
- dma_dest_const : Zieladresse nicht erhöhen
- dma_dest_inc: Zieladresse erhöhen
- dma_dest_dec: Zieladresse verkleinern
- dma_want_reply: der DMA Server bestätigt die erlogreiche Übertragung
- der Wert 0 steht für dma_source_inc und dma_dest_inc ohne reply

Die einzelnen Flags können ODER verknüpft werden.

Rückgabewert: void

Hinweis:
Wenn ein DMA Reply verlangt wird muss die entsprechende Nachricht auch abgefragt werden da sonst der Messagebuffer blockiert wird. Der Reply wird immer über Kanal 0 versendet.

Beispiel:

```
int  dma_dest=0;
sendDMAMessage((int)port_adr,(int)&dma_dest,4,dma_source_const|dma_want_reply,7);
```

Überträgt einen int-Wert von einer konstanten Quelladresse (z.B. serieller Port) in den Speicher und bestätigt die Übertragung.
(Siehe Beispiel zur DMA Nachrichtenübertragung)

Weitere interne DMA Befehle Siehe [Kra02]

3.5.2 Nachrichtensystem

Um das Nachrichtensystem verwenden zu können, muss es in der CONFIG.H mit **#define MSG** aktiviert werden.

Die Größe des Messagebuffers wird in CONFIG.H durch **#define MsgBufLength** angegeben. Empfohlene Werte liegen zwischen 20 und 40.

Für das Senden von Nachrichten im Mehrprozessorbetrieb muss **#define MPRAM** in CONFIG.H definiert werden. Die Länge des Puffers **#define MPRAMBufLength** muss mit **#define MsgBufLength** übereinstimmen. Über den Wert **#define Prozessor** wird festgelegt auf welchem Prozessor das System laufen soll. Im Linker.cmd File muss mindestens (**MPRAMBufLength*2+4)*32 Bit** Speicher aus dem MPRAM reserviert werden. Um die gesamten 32 Kbyte des MPRAM zu reservieren, ist folgende Befehlszeile in der Linker.cmd nötig.

MPRAM: org=1740000h l=00008000h
(Adresse abhängig von den Eigenschaften des Boards, bei Daytona 1740000h)

Das Senden von Nachrichten im Mehrprozessorbetrieb setzt das einfache Messagesystem voraus. Im Mehrprozessorbetrieb ist zu beachten, dass für eine korrekte Initialisierung des Dual Port Rams ein Breakpoint bei OSystem() in der Main() Routine gesetzt werden muss und die Projekte der beiden CPUs müssen jeweils gestartet und bis zu diesem Punkt ausgeführt werden, bevor sie fortgesetzt werden können.

Beispiel:

```
#define MSG
#define MPRAM

#ifdef MSG
#define MsgBufLength 20
#endif
#ifdef MPRAM
        #define MPRAMBufLength 20
        #define Prozessor 2
#endif
```

int askmessage(int ChannelNr, int Message[3])

Beschreibung:

Prüft, ob auf dem angegebnen Kanal eine Nachricht für das Task vorhanden ist. Falls dies so ist, wird die Nachricht in Message[3] abgelegt und aus dem Messagebuffer gelöscht.

Parameter:

ChannelNr: Empfangskanalnummer. (Wert von 0 bis 128)
Message[3]: Array in das die Nachricht abgelegt wird, bestehend aus drei 32 Bit Integerwerten

Rückgabewert: TaskID des Tasks der die Nachricht gesendet hat oder -1 wenn keine Nachricht vorhanden

Hinweis:
Es besteht kein Unterschied in der Verwendung des Befehls für interne und externe Nachrichten.

Beispiel:
SenderID=askmessage(0,Nachricht);
(Siehe Nachrichtensystem Beispiel)

int sendmessage(int EmpfID, int ChannelNr, int Message[3])

Beschreibung:

Sendet die angegebene Nachricht an den definierten Empfänger über den definierten Kanal.

Parameter:

EmpfID: TaskID des Tasks der die Nachricht erhalten soll.
ChannelNr: Nummer des Kanals auf dem gesendet werden soll (Wert von 0 bis 128)
Message[3]: Array in das die Nachricht abgelegt wird, bestehend aus drei 32 Bit Integerwerten

Rückgabewert:

- „0“ Message wurde erfolgreich gesendet. Empfang wird nicht garantiert.
- „-1“ Fehler beim Senden einer internen Message.
- „-2“ Fehler beim Senden einer externen Message.

Hinweis:
Es besteht kein Unterschied in der Verwendung des Befehls für interne und externe Nachrichten.

Beispiel:
sendmessage(102,0,Nachricht); (Siehe Nachrichtensystem Beispiel)

int testmsg(int EmpfID, int SendID, int ChannelNr)

Beschreibung:

Testet, ob für den definierten Empfänger eine Nachricht vom definierten Sender auf dem angegebenen Kanal vorliegt. Die Nachricht selbst wird nicht ausgelesen oder verändert.

Parameter:

EmpfID: TaskID des Tasks der die Nachricht erhalten soll.
SendID: TaskID des Tasks der die Nachricht gesendet hat.
ChannelNr: Nummer des Kanals der getestet werden soll (Wert von 0 bis 128)

Rückgabewert:

- „1“ es ist eine Nachricht vorhanden
- „0“ sonst

Hinweis:
Es besteht kein Unterschied in der Verwendung des Befehls für interne und externe Nachrichten.

Beispiel:
if (testmsg(102,5,1)) {};

int testallmsg(int EmpfID)

Beschreibung:

Testet ob für den definierten Empfänger eine Nachricht vorliegt, unabhängig von der SenderID oder der Kanalnummer. Die Nachricht selbst wird nicht ausgelesen oder verändert.

Parameter:

EmpfID: TaskID des Tasks der die Nachricht erhalten soll.

Rückgabewert:

- „1“ es ist eine Nachricht vorhanden
- „0“ sonst

Hinweis:
Es besteht kein Unterschied in der Verwendung des Befehls für interne und externe Nachrichten.

Beispiel:
if (testallmsg(102)) {};

int get_sendmp()

Beschreibung:

Bestimmt die Anzahl der Messages auf dem Dual-Port RAM, die von Betriebssystem an das Betriebssystem des anderen Prozessors gerichtet sind und noch nicht abgeholt wurden.

Parameter:

Rückgabewert: Anzahl der Messages

Hinweis:

Beispiel:
int count=get_sendmp();

int get_askmp()

Beschreibung:

Bestimmt die Anzahl der Messages auf dem Dual-Port RAM, die vom Betriebssystem des anderen Prozessors an das Betriebssystem gerichtet sind und noch nicht abgeholt wurden.

Parameter:

Rückgabewert: Anzahl der Messages

Hinweis:

Beispiel:
int count=get_askmp();

void initMSG()

Beschreibung:

Initialisiert den internen Messagebuffer.

Parameter:

Rückgabewert:

Hinweis:
Alle Nachrichten, die sich im internen Nachrichtenpuffer befinden, werden gelöscht.

Beispiel:

void initMPRAM()

Beschreibung:

Initialisiert den externen Messagebufferbereich, der zum Betribssystem gehört.

Parameter:

Rückgabewert:

Hinweis:
Wenn sich die Adresse des Dual-Port Ram ändern sollte, muss diese Funktion angepasst werden.

Beispiel:

3.5.3 Betriebsmittelverwaltung

Um die Betriebsmittelverwaltung nutzen zu können muss #define BMV und die Anzahl der Betriebsmittel #define BMANZAHL in der CONFIG.H definiert werden.

Beispiel:
#define BMV
#define BMANZAHL 10

int create_bm(int ID, int Instanzen)

Beschreibung:
Ein zu verwaltendes Betriebsmittel wird definiert und seine Eigenschaften festgelegt

Parameter:

ID:	Die ID des zu verwaltenden Betriebsmittels (keine negativen Werte)
Instanzen:	Die Anzahl der Instanzen die für dieses Betriebsmittel zur Verfügung stehen (keine negativen Werte)

Rückgabewert:

- „0“ erfolgreiches Anlegen des Betriebsmittels
- „-1“ ungenügend freier Speicher
- „-2“ ID schon benutzt

Hinweis:
Da der Zugriff auf die Betriebsmittel nur über ihre ID verwaltet wird und sie nicht direkt angesprochen werden, können beliebige Ressourcen als Betriebsmittel angemeldet werden. Es kann aber auch nicht kontrolliert werden, ob die Ressource an der Betriebsmittelverwaltung vorbei belegt wird.

Beispiel:
int ID_USB=4;
create_bm(ID_USB,1);
(Siehe Beispiel für BM Verwaltung)

int delete_bm(int ID)

Beschreibung:

Entfernt ein Betriebsmittel aus der Betriebsmittelverwaltung

Parameter:

ID: Die ID des zu entfernenden Betriebsmittels (keine negativen Werte)

Rückgabewert:

- „0“ Erfolgreiches entfernen des Betriebsmittels
- „-1“ ID existiert nicht
-

Hinweis:
Es wird vor dem Löschen nicht überprüft, ob das Betriebsmittel noch von einem anderen Task belegt ist.

Beispiel:
int ID_USB=4;
delete_bm(ID_USB);
(Siehe Beispiel für BM Verwaltung)

int occupy_bm(int ID)

Beschreibung:

Belegt ein Betriebsmittel aus der Betriebsmittelverwaltung

Parameter:

ID: Die ID des zu belegenden Betriebsmittels (keine negativen Werte)

Rückgabewert:

- „0“ erfolgreiches Belegen des Betriebsmittels
- „-1“ ID existiert nicht
-

Hinweis:

Beispiel:
int ID_USB=4;
occupy_bm(ID_USB);
(Siehe Beispiel für BM Verwaltung)

int free_bm(int ID)

Beschreibung:

Gibt ein vom Task belegtes Betriebsmittel aus der Betriebsmittelverwaltung frei. Der Task muss Eigentümer des Betriebsmittels sein.

Parameter:

ID: Die ID des freizugebenden Betriebsmittels (keine negativen Werte)

Rückgabewert:

- „0“ erfolgreiches Entfernen des Betriebsmittels
- „-1“ ID existiert nicht
-

Hinweis:

Beispiel:
int ID_USB=4;
free_bm(ID_USB);
(Siehe Beispiel für BM Verwaltung)

int owner_bm(int ID, int Instanz)

Beschreibung:

Stellt fest welcher Task gerade ein bestimmtes Betriebsmittel aus der Betriebsmittelverwaltung benutzt.

Parameter:

ID: Die ID des Betriebsmittels (keine negativen Werte)
Instanz: Die Instanz des Betriebsmittels (keine negativen Werte)

Rückgabewert:

- Die TaskID des Tasks der das Betriebsmittel belegt
- „-1“ das Betriebsmittels ist nicht belegt
- „-2“ ID und/oder Instanz existiert nicht

Hinweis:

Beispiel:
int ID_USB=4;
owner(ID_USB, 1);
(Siehe Beispiel für BM Verwaltung)

3.5.4 Messwertspeicher

Um auf den Messwertspeicher zugreifen zu können, muss **#define MESSW** in der CONFIG.H definiert sein und die Datei MESSV.C muss in das Projekt eingebunden werden. Es existieren zwei unterschiedliche Modi des Messwertspeichers, die in der Datei MESSV.C zu definieren sind. Es ist zu beachten, dass sich die Befehlsnamesstruktur zur eRTOS Version 2.0 geändert hat. Aus ms_x() wurde x_ms().

#define MW_ERZ
Wenn nur jeweils ein Task auf den Messwertspeicher lesend oder schreibend zugreift. (Reines Erzeuger/Verbraucher Problem)
Ist dieses Define nicht gesetzt, wird eine langsamere speicherintensivere Implementierung genutzt, bei der auch mehr als ein Task auf den Messwertspeicher zugreifen kann. Der korrekte Zugriff wird intern geregelt.

int create_ms(int Size, int Block_Size)

Beschreibung:

Diese Funktion erzeugt einen Ringspeicher mit Hilfe der dynamischen Speicherverwaltung.

Parameter:

Size: Die Größe des Messwertspeichers in 32Bit-Blöcken
Block_Size: Die Größe der einzelnen Messwerteblöcke in 32 Bit Schritten

Rückgabewert:

- „0“ erfolgreiche Erstellung des Messwertspeichers
- „-1“ falsche Parameter
- „-2“ nicht genug Speicher frei

Hinweis:
Der Speicherverbrauch ist abhängig von der Implementierung. Für die einfache Variante entspricht der Speicherverbrauch für den Ringpuffer der angegebenen Größe in Size. Für die Variante mit parallelen Zugriff müssen pro Block noch 4 Byte Headerinformation dazugerechnet werden.
Durch die zeitaufwändige Headerinitialisierung sollte diese Variante des Messwertspeichers auch immer während der Betriebssysteminitialisierung erzeugt werden.

Beispiel:
create_ms (10, 1);
erzeugt einen Ringpuffer für zehn 32Bit Messwerte
create_ms (10, 2);
erzeugt einen Ringpuffer für fünf 64Bit Messwerte
(Siehe Beispiel für Messwertspeicher)

int delete_ms()

Beschreibung:

Der Messwertspeicher wird gelöscht und der belegte Speicher freigegeben.

Parameter:

Rückgabewert:

- „0“ Messwertspeichers erfolgreich entfernt
- „-1“ sonst

Hinweis:

Beispiel:

int empty_ms()

Beschreibung:

Es wird getestet, ob der Messwertspeicher leer ist.

Parameter:

Rückgabewert:

- „0“ der Messwertspeicher ist Leer
- „-1“ sonst

Hinweis:

Beispiel:

int enqueue_ms(int *Zeiger)

Beschreibung:

Es wird ein neues Element mit dem an der Adresse von Zeiger vorhandenen Wert erstellt. Dem Messwertspeicher wird dieses Element zugefügt. Die Blockgrößen des Messwertspeichers und der Quelle an der Adresse von Zeiger müssen übereinstimmen.

Parameter:

Zeiger: Zeiger auf den Wert der gespeichert werden soll.

Rückgabewert:

- „0“ Erfolg
- „-1“ voller oder nicht initialisierter Messwertspeicher

Beispiel:

```
int Source=5 ;
enqueue_ms(&Source);
```

Der Wert 5 wird dem Messwertspeicher zugefügt.
(Siehe Beispiel für Messwertspeicher)

int dequeue_ms(int *Zeiger)

Beschreibung:

Das zuletzt gespeicherte Element im Messwertspeicher wird an die Adresse, die Zeiger angibt, kopiert und anschließend aus dem Messwertspeicher entfernt. Die Blockgrößen des Messwertspeichers und des Ziels an der Adresse von Zeiger müssen übereinstimmen.

Parameter:

Zeiger: Zeiger auf den Bereich in dem der Wert gespeichert werden soll.

Rückgabewert:

- „0“ Erfolg
- „-1“ leerer oder nicht initialisierter Messwertspeicher

Hinweis:

Beispiel:
int Source;
dequeue_ms(&Source);
Die Variable Source erhält den Wert der zuletzt im Messwertspeicher gespeichert wurde.
(Siehe Beispiel für Messwertspeicher)

int first_ms(int *Zeiger)

Beschreibung:

Das erste Element im Messwertspeicher wird an die Adresse die Zeiger angibt kopiert. Die Blockgrößen des Messwertspeichers und des Ziels an der Adresse von Zeiger müssen übereinstimmen.

Parameter:

Zeiger: Zeiger auf den Bereich in dem der Wert gespeichert werden soll.

Rückgabewert:

- „0“ Erfolg
- „-1“ leerer oder nicht initialisierter Messwertspeicher

Hinweis:
Diese Funktion ist nur für den MW_ERZ Modus definiert.

Beispiel:
int Source;
first_ms(&Source);
Die Variable Source erhält den Wert der als erster im Ringpuffer steht.

3.5.5 Taskverwaltung

Als Task können im System nur Funktionen ausgeführt werden die keine Parameter benötigen und auch keine Rückgabewerte liefern (void function(void)). Für k-Tasks muss zusätzlich gelten, dass sie in einem festen maximalen Zeitraum (innerhalb ihrer Periode) terminieren. Interruptabhängige nk-Tasks müssen sich am Ende (letzter Befehl) über end_int(IntNr) selbst suspendieren, damit sie erst wieder beim nächsten Auftreten des Interrupts ausgeführt werden.

int create_kTask(void (*fkt)(void), int ID, int Periode, int Starttick)

Beschreibung:

Erzeugt einen kooperierenden Task

Parameter:

void (*fkt)(void):	Zeiger auf eine void-Funktion die als kTask laufen soll.
ID:	Die ID des Tasks (Werte von 0 bis 99 zulässig)
Periode:	Periodendauer des Tasks in Ticks
Starttick:	Zeitpunkt an dem der Task das erste mal ausgeführt werden soll. Angabe in Ticks.

Rückgabewert:

- „0“ Erfolg
- „-1“ sonst

Hinweis:

Dieser Befehl kann im Initialisierungsbereich des eRTOS (Main() Routine) der Hauptdatei verwendet werden oder aber auch aus einem Task heraus aufgerufen werden. Die k-Tasks müssen zu unterschiedlichen Startticks beginnen und die Ausführungszeit darf ihre Periodendauer nicht überschreiten.

Beispiel:
void Task();
create_kTask(&Task,53,4,0);
Die Funktion Task erhält die TaskID 53, hat eine Periodendauer von 4 Ticks und wird im Tick 0 gestartet.
(Siehe Beispiel zur Taskverwaltung)

int del_ktask (int ID)

Beschreibung:

Löscht einen kooperierenden Task aus dem Scheduler.

Parameter:

ID:	Die ID des Tasks (Werte von 0 bis 99 zulässig)

Rückgabewert:

- „0“ Erfolg
- „-1“ Task dieser ID existiert nicht

Hinweis:
In das Betriebssystem ist keine Garbage Collection integriert! Wenn ein Task gelöscht wird, der dynamischen Speicher benutzt hat, wird dieser nicht wieder automatisch freigegeben. Darum muss sich der Nutzer kümmern. Siehe [Sch01] S. 96.

Beispiel:
delete_kTask(53);
Die Funktion mit der TaskID 53 wird aus dem Scheduler entfernt.

int change_kTask(int ID, int Periode, int Starttick)

Beschreibung:

Ändert die Eigenschaften eines kooperierenden Tasks

Parameter:

ID:	Die ID des Tasks (Werte von 0 bis 99 zulässig)
Periode:	Neue Periodendauer des Tasks in Ticks
Starttick:	Neuer Zeitpunkt an dem der Task das erste mal ausgeführt werden soll. Angabe in Ticks.

Rückgabewert:

- „0“ Erfolg
- „-1“ sonst

Hinweis:

Beispiel:
change_kTask(53,50,tick()+100);
Ändert die Periode des Tasks auf 50 Ticks. Er wird in 100 Ticks das erste mal ausgeführt.

int create_nkTask(int ID, void (*fkt)(void), int Prioritaet, int intNr)

Beschreibung:

Erzeugt einen nichtkooperierenden (unterbrechbaren) Task

Parameter:

void (*fkt)(void):	Zeiger auf eine void-Funktion die als nk-Task laufen soll.
ID:	Die ID des Tasks (Werte ab 100 bis maxInt)
Prioritaet:	Priorität des Tasks(0-höchste bis 2-niedrigste)
intNr:	Interupt das die Ausführung des Tasks auslöst. (Werte von 5 bis 15 zulässig)

Rückgabewert:

- „0“ Erfolg
- „-1“ falsche Parameter
- „-2“ nicht genügend Speicher frei

Hinweis:

Dieser Befehl kann im Initialisierungsbereich des eRTOS (Main() Routine) der Hauptdatei verwendet werden oder aber auch aus einem Task heraus aufgerufen werden. Tasks die in Zusammenhang mit Interrupts ausgeführt werden, werden vom Scheduler gegenüber normalen nkTasks bevorzugt behandelt. Damit der nk-Task nur beim Auftreten des entsprechenden Interupts ausgeführt wird, muss am Ende des Tasks ein end_int() mit der Interruptnummer aufgerufen werden.

Beispiel:

```
void Task();
create_nkTask(153,&Task,0,7);
```

Die Funktion Task erhält die TaskID 153 und wird beim Auftreten des Interrupts 7 ausgeführt.
(Siehe Beispiel zur Taskverwaltung)

int nkzustand (int ID)

Beschreibung:

Abfrage des Zustandes eines nichtkooperierenden Tasks

Parameter:

ID: Die ID des Tasks (Werte ab 100 bis maxInt)

Rückgabewert:

- Statuswert (NKS_AKTIV, NKS, BEREIT NKS, NKS_SUSPENDIERT, NKS_WARTEN)
- „-1“ nicht existierende TaskID

Hinweis:

Beispiel:
int status=nkzustand(153);
Der Variable status wird der Wert für den Status des Tasks zugewiesen
(Siehe Beispiel zur Taskverwaltung)

void selfsuspend ()

Beschreibung:

Der aktuell laufende nk-Task wird suspendiert.

Parameter:

Rückgabewert:

Hinweis:

Funktioniert nicht Richtig.

Beispiel:

int suspend (int ID)

Beschreibung:

Suspendieren des angegebenen nk-Tasks

Parameter:

ID:	Die ID des Tasks der suspendiert werden soll (Werte ab 100 bis maxInt)

Rückgabewert:

- „0“ bei Erfolg
- „-1“ nicht existierende TaskID

Hinweis:

Beispiel:
suspend(153);
Der nk-Task mit der ID 153 wird suspendiert
(Siehe Beispiel zur Taskverwaltung)

void end_int (int intNr)

Beschreibung:

Interrupttask suspendiert sich unter Beachtung des Interruptpuffers nach Ausführung selbst.

Parameter:

intNr:	Interrupt von dem das Task abhängig ist (Werte von 5 bis 15 zulässig)

Rückgabewert:

Hinweis:
Der Task wird erst beim nächsten auftreten des Interupts wieder ausgeführt. Im Zusammenspiel mit k-Tasks können Fehler auftreten.

Beispiel:
(Siehe Beispiel zur Taskverwaltung)

int resuspend (int ID)

Beschreibung:

Ein suspendierter nk-Task wird in den Zustand, den er vor seiner Suspendierung hatte zurückversetzt.

Parameter:

ID:	Die ID des Tasks der resuspendiert werden soll (Werte ab 100 bis maxInt)

Rückgabewert:

- „0“ bei Erfolg
- „-1“ nicht existierende TaskID
- „-2“ Task war nicht suspendiert

Hinweis:

Beispiel:
resuspend(153);
Der nk-Task mit der ID 153 wird in den Zustand versetzt, den er vor seiner Suspendierung hatte.
(Siehe Beispiel zur Taskverwaltung)

int warten (int ID, int Auf, int Option1, int Option2)

Beschreibung:

Versetzten eines nk-Tasks in den Zustand warten auf. Es kann eine bestimmte Zeit, auf das Eintreffen einer Nachricht oder auf das Freiwerden eines Betriebsmittels gewartet werden.

Parameter:

ID:	Die ID des Tasks das warten soll (Werte ab 100 bis maxInt)
Auf:	spezifiziert auf was gewartet werden soll Wählbar: NKS_ZEIT, NKS_MSG, NKS_BM
Option1:	NKS_ZEIT: Anzahl Ticks NKS_MSG: SenderID NKS_BM: BetriebsmittelID
Option2:	NKS_ZEIT: keine Funktion NKS_MSG: Kanalnummer NKS_BM: keine Funktion

Rückgabewert:

- „0“ bei Erfolg
- „-1“ nicht existierende TaskID
- „-2“ falsche Parameter

Hinweis:
Bei mehrfachem Aufruf speziell aus nk-Tasks kann es zu Fehlfunktionen im eRTOS kommen.

Beispiel:
warten(153,NKS_MSG,101,0);
Der nk-Task mit der ID 153 wartet auf eine Nachricht vom Task mit der ID 101 auf dem Kanal 0.
(Siehe Beispiel zur Taskverwaltung)

int delete_nkTask (int ID)

Beschreibung:

Löscht einen nicht kooperierenden Task aus dem Scheduler und gibt den von dem Task belegten Speicher frei.

Parameter:

ID:	Die ID des Tasks das gelöscht werden soll (Werte ab 100 bis maxInt)

Rückgabewert:

- „0“ bei Erfolg
- „-1“ nicht existierende TaskID

Hinweis:
Diese Funktion funktioniert nicht richtig. nk-Tasks werden nicht aus dem Scheduler entfernt. In das Betriebssystem ist keine Garbage Collection integriert! Wenn ein Task gelöscht wird, der dynamischen Speicher benutzt hat, wird dieser nicht wieder automatisch freigegeben. Darum muss sich der Nutzer kümmern. Siehe [Sch01] S. 96.

Beispiel:

int chng_int (int ID, int intNr)

Beschreibung:

Dem durch die ID gekennzeichneter Task wird ein neuer / ein Interrupt zugewiesen.

Parameter:

ID:	Die ID des Tasks (Werte ab 100 bis maxInt)
intNr:	Interupt das die Ausführung des Tasks auslösen soll. (Werte von 5 bis 15 zulässig)

Rückgabewert:

- „0“ bei Erfolg
- „-1“ sonst

Hinweis:

Beispiel:

int chng_prio (int ID, int Prioritaet)

Beschreibung:

Dem durch die ID gekennzeichneter Task wird eine neue Prioritaet zugewiesen.

Parameter:

ID:	Die ID des Tasks (Werte ab 100 bis maxInt)
Prioritaet:	neue Priorität des Tasks(0-höchste bis 2-niedrigste)

Rückgabewert:

- „0“ bei Erfolg
- „-1“ sonst

Hinweis:

Beispiel:
(Siehe Beispiel zur Taskverwaltung)

int tick()

Beschreibung:

Liefert den aktuellen Tick zurück.

Parameter:

Rückgabewert:

- positiver Integer Wert der den aktuellen Tick wiedergibt

Hinweis:

Beispiel:
(Siehe Beispiel zur Taskverwaltung)

float tps()

Beschreibung:

Liefert den aktuellen Wert für die Ticks pro Sekunde zurück.

Parameter:

Rückgabewert:

- positiver Gleitkomma Wert der den aktuellen Tick wiedergibt

Hinweis:

Beispiel:

3.5.6 Dynamische Speicherverwaltung

int *nAlloc(int Anfrage, int SArt)

Beschreibung:

Allociert Speicher in dem gewünschten Speicherbereich

Parameter:

Anfrage:	Gewünschte Speichergröße in Byte
SArt:	gewünschte Speicherart. Ist abhängig von der Speicherbelegung durch den Linker. (Standard: 0->interner Speicher; 1->SBS Speicher)

Rückgabewert:

- Zeiger auf Speicherbereich bei Erfolg
- „-1“ sonst

Hinweis:
Ab Anfragen größer als 4 Byte werden immer vielfache von 32 Bit reserviert. Anfragen die kleiner als 4 Byte sind werden in Speicherzellen der Größe 16 Bit abgelegt.

Beispiel:

int *nFree(int *Zeiger)

Beschreibung:

Freigabe des durch nAlloc belegten Speicherbereichs

Parameter:

Zeiger: Zeiger auf den freizugebenden Speicher

Rückgabewert:

- „0“ bei Erfolg
- „-1“ wenn der Zeiger auf kein dynamisches Element verweist.

Hinweis:

Beispiel:

3.5.7 Interrupt und Registerbefehle

void istinit()

Beschreibung:

Setzt den IST Pointer auf die Adresse 0x1000.

Parameter:

Rückgabewert:

Hinweis:

Die Adresse für die IST wird in der Datei LINKER.CMD festgelegt. Falls die Adresse dort verändert wird, muss die Funktion entsprechend angepasst werden.

Beispiel:

void int_multiplex()

Beschreibung:

Setzt die Interrupt Select Register unter Verwendung der Variablen intsel1 und intsel2.

Parameter:

Rückgabewert:

Hinweis:

Die Variablen intsel1 und intsel2 werden in der Datei INTERRUPT.C initialisiert.

Beispiel:

void wrwordh()

Beschreibung:

Schreibt das Highword des Interruptmultiplexers (IMX). Als Quelle dient die Variable intsel2.

Parameter:

Rückgabewert:

Hinweis:

Die Variablen intsel1 und intsel2 werden in der Datei INTERRUPT.C initialisiert.

Beispiel:

void wrwordl()

Beschreibung:

Schreibt das Lowword des Interruptmultiplexers (IMX). Als Quelle dient die Variable intsel1.

Parameter:

Rückgabewert:

Hinweis:

Die Variablen intsel1 und intsel2 werden in der Datei INTERRUPT.C initialisiert.

Beispiel:

void disablemi()

Beschreibung:

Alle Interrupts werden durch das General Interrupt Enable (GIE) Bit disabled.

Parameter:

Rückgabewert:

Hinweis:

Beispiel:

void enablemi()

Beschreibung:

Alle Interrupts werden durch das General Interrupt Enable (GIE) Bit enabled.

Parameter:

Rückgabewert:

Hinweis:

Beispiel:

void disablenmi()

Beschreibung:

Der Non-Maskable-Interrupt (NMI) wird disabled.

Parameter:

Rückgabewert:

Hinweis:

Beispiel:

void enablenmi()

Beschreibung:

Der Non-Maskable-Interrupt (NMI) wird enabled.

Parameter:

Rückgabewert:

void disableint(int INT)

Beschreibung:

Disabled den angegebenen Interrupt

Parameter:

INT: Interruptnummer (gültige Werte: 1, 4 – 15)

Rückgabewert:

Hinweis:

Beispiel:

void enableint(int INT)

Beschreibung:

Enabled den angegebenen Interrupt
Parameter:

INT: Interruptnummer (gültige Werte: 1, 4 – 15)

Rückgabewert:

Hinweis:

Beispiel:
(Siehe Beispiel zur Taskverwaltung)

void setint(int INT)

Beschreibung:

Setzt das Interruptflag des angegebenen Interrupts

Parameter:

INT: Interruptnummer (gültige Werte: 1, 4 – 15)

Rückgabewert:

Hinweis:

Beispiel:

void clrint(int INT)

Beschreibung:

Löscht das Interruptflag des angegebenen Interrupts

Parameter:

INT: Interruptnummer (gültige Werte: 1, 4 – 15)

Rückgabewert:

Hinweis:

Beispiel:

void mistart(int INT)

Beschreibung:

Startet die dem angegebenen Interrupt zugehörige Service Routine (ISR)

Parameter:

INT: Interruptnummer (gültige Werte: 1, 4 – 15)

Rückgabewert:

Hinweis:

Achtung! Wird ein Interrupt, der keine ISR besitzt, angegeben wird die ISR des nächsten Interrupts mit ISR ausgeführt. Falls kein höherer Interrupt eine ISR besitzt, wird der Code ausgeführt, der an die IST anschließt.

Beispiel:

int getint()

Beschreibung:

Liefert den höchstpriorisierten Interrupt zurück.

Parameter:

Rückgabewert:

- Interruptnummer
- „0“ kein Interruptflag ist gesetzt

Hinweis:

Beispiel:

int getsp()

Beschreibung:

Liefert den aktuellen Stackpointerinhalt zurück.

Parameter:

Rückgabewert:

- Stackpointerinhalt

Hinweis:

Beispiel:

setsp(int Stack)

Beschreibung:

Setzt den Stackpointer auf die angegebene Adresse.

Parameter:

Stack: neue Adresse des Stack (32 Bit)

Rückgabewert:

Hinweis:

Beispiel:

3.5.8 Timerbefehle

void inittimer()

Beschreibung:

Initialisiert den Timer 0 mit dem Inhalt der Variable tcrbeleg

Parameter:

Rückgabewert:

Hinweis:

Die Variable tcrbeleg wird in der Datei INTERRUPT.C initialisiert.

Beispiel:

void inittimer1()

Beschreibung:

Initialisiert den Timer 1 mit dem Inhalt der Variable tcr2beleg

Parameter:

Rückgabewert:

Hinweis:

Die Variable tcr2beleg wird in der Datei INTERRUPT.C initialisiert.

Beispiel:
(Siehe Beispiel zur Taskverwaltung)

setperiod(int Periode)

Beschreibung:

Setzen des Timer Period Registers des Timer0 mit dem angegebenen Wert.

Parameter:

Periode: Wert für das TPR (Gültige Werte: 0-65000)

Rückgabewert:

Hinweis:

Der Timer zählt mit folgender Geschwindigkeit: $\frac{\text{Taktfrequenz der CPU}}{4}$

Beispiel:

setperiod1(int Periode)

Beschreibung:

Setzen des Timer Period Registers des Timer1 mit dem angegebenen Wert.

Parameter:

Periode: Wert für das TPR (Gültige Werte: 0-65000)

Rückgabewert:

Hinweis:

Der Timer zählt mit folgender Geschwindigkeit: $\frac{\text{Taktfrequenz der CPU}}{4}$

Beispiel:
(Siehe Beispiel zur Taskverwaltung)

int getcount()

Beschreibung:

Liefert den aktuellen Inhalt des Timer Counter Registers für den Timer0 zurück.

Parameter:

Rückgabewert:

- Wert des Timer Counter Registers für den Timer0

Hinweis:

Beispiel:

inittim(int tcr)

Beschreibung:

Setzt das Timer Control Register (TCR) des Timer 0 mit dem angegebenen Wert

Parameter:

tcr: Wert für das TCR

Rückgabewert:

Hinweis:

Gültiger Aufbau eines Wertes für das TCR.
Die ersten 20 Bit sind reserviert. Danach jeweils 1 Bit für Folgende Werte in der angegebenen Reihenfolge.
TSTAT / INVINP / CLKSRC / C/P / HLD / GO / reserved / PWID / DATIN / DATOUT / INVOUT / FUNC

Beispiel:

inittim1(int tcr)

Beschreibung:

Setzt das Timer Control Register (TCR) des Timer 1 mit dem angegebenen Wert

Parameter:

tcr: Wert für das TCR

Rückgabewert:

Hinweis:

Gültiger Aufbau eines Wertes für das TCR.
Die ersten 20 Bit sind reserviert. Danach jeweils 1 Bit für Folgende Werte in der angegebenen Reihenfolge.
TSTAT / INVINP / CLKSRC / C/P / HLD / GO / reserved / PWID / DATIN / DATOUT / INVOUT / FUNC

Beispiel:

timergo()

Beschreibung:

Startet den Timer0

Parameter:

Rückgabewert:

Hinweis:

timergo1()

Beschreibung:

Startet den Timer1

Parameter:

Rückgabewert:

Hinweis:

Beispiel:
(Siehe Beispiel zur Taskverwaltung)

3.5.9 Zusatzfunktionen

Um diese Befehle verwenden zu können, muss das D-Module Board vorhanden sein und in der CONFIG.H muss #define D_MODULE gesetzt sein.

int send_char(char Wert)

Beschreibung:

Sendet einen Character an die serielle Schnittstelle des D-Modules

Parameter:

Wert: Der zu übertragende char Wert

Rückgabewert:

- „1“ bei Erfolg
- „0“ wenn Puffer voll

Hinweis:

Beispiel:

void send_string(char *Wert)

Beschreibung:

Sendet einen String an die serielle Schnittstelle des D-Modules

Parameter:

Wert: Der zu übertragende int Wert

Rückgabewert:

Hinweis:

Die einzelnen Character Werte des Strings werden nacheinander an die Schnittstelle übertragen.

Beispiel:

void sendint(int Wert)

Beschreibung:

Sendet einen Integer Wert an die serielle Schnittstelle des D-Modules

Parameter:

Wert: Ein Zeiger auf den zu übertragenden String

Rückgabewert:

Hinweis:

Der 32 Bit Integer Wert wird in 4 Character Werte (8Bit) aufgespalten und nacheinander an die Schnittstelle übertragen.

Beispiel

int receive_char();

Beschreibung:

Empfängt einen Character über die serielle Schnittstelle des D-Modules

Parameter:

Rückgabewert:

- Inhalt der seriellen Schnittstelle

Hinweis:

Über Interrupt 7 wird angezeigt, ob ein neuer Wert bereit zum Lesen ist.

Beispiel:

4 Ausblick auf künftige Entwicklungsmöglichkeiten

4.1 Mehrprozessorscheduling

Perspektivisch sollen im „Mehrkoordinaten Mess- und Positioniersystem“ mehrere DSP parallel die anfallenden Daten verarbeiten.

Mit der jetzigen Version des Einzelprozessor eRTOS ist es möglich pro DSP eine Version des eRTOS zu installieren, die dann mit der Multiprozessornachrichtenerweiterung in der Lage ist, mit anderen (momentan mit einem anderen) eRTOS zu kommunizieren. Jedes dieser eRTOS verwaltet seine Tasks separat und es ist nicht möglich beim Freiwerden von Kapazitäten eines DSPs Tasks auf diesen zur Abarbeitung zu transferieren.

Ein Beispiel aus der Bedientheorie zeigt, das es sinnvoll ist, die Verwaltung der anfallenden Aufgaben zu zentralisieren.

Um das Beispiel zu vereinfachen, wird angenommen, dass der momentane Zustand des Systems von seiner Vergangenheit unabhängig ist. Als Beispiel dient ein M/M/2-∞ Wartesystem, also ein System dessen Ankunftsrate (8) und Bedienzeit (µ) der Aufträge Negativ Expotenziell verteilt ist und somit die Markoveigenschaft besitzt. Das System besitzt 2 Bedieneinheiten und die Warteschlange ist unendlich. Eine Erweiterung des Beispiels durch die Verwendung von Warteschlangennetzen wäre auch möglich.

Es gibt nun 2 Varianten die Bedieneinheiten (oder Prozessoren) mit Aufträgen (oder Tasks) zur Abarbeitung zu versorgen. Jedes Bedienelement hat seine eigene Warteschlange die mit Aufträgen versorgt wird oder beide Bedienelemente werden aus einer Warteschlange mit Aufträgen versorgt, die alle anfallenden Aufträge aufnimmt.

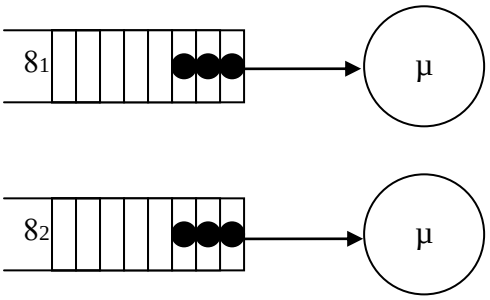


Abb. 31 Bedienelemente mit separaten Warteschlangen

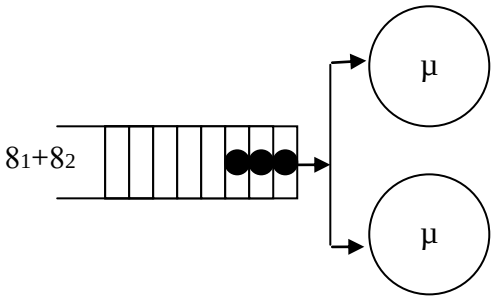


Abb. 32 Bedienelemente mit gemeinsamer Warteschlange

Die Auftragsquellen 8_1 und 8_2 sollen jeweils 20 Aufträge pro Sekunde liefern. Jedes Bedienelement ist in der Lage 50 Aufträge pro Sekunde abzuarbeiten.

Im ersten Fall handelt es sich um den Spezialfall zweier kombinierter M/M/1 Systeme, die durchschnittliche Verweildauer der Aufträge im System errechnet sich deshalb wie folgt:

Verweildauer pro M/M/1 System:

$$\mathrm{E}\{T_V\} = \frac{1}{\mu - \lambda} = 0{,}033s$$

Gesamtverweildauer:

$$\mathrm{E}\{T_V\} = \frac{\lambda_1 * \mathrm{E}\{T_{V1}\} + \lambda_2 * \mathrm{E}\{T_{V2}\}}{\lambda_1 + \lambda_2} = 0{,}033s$$

Im 2. Fall müssen die Formeln für das M/M/m-∞ Wartesystem genutzt werden. Die mittlere Verweildauer im System berechnet sich deswegen wie folgt:

$$\mathrm{E}\{T_V\} = \frac{p_W}{\lambda} * \frac{\rho}{1-\rho} + \frac{1}{\mu} = 0{,}024s$$

Die Auslastung errechnet sich dabei durch $\rho = \frac{\lambda}{m * \mu}$

Die Wahrscheinlichkeit für ein Warten auf die Bedienung durch $p_W = p_0 * \frac{(m*\rho)^m}{m!*(1-\rho)}$

Die Wahrscheinlichkeit für ein leeres System mit m=2 durch $p_0 = \left[1 + (2*\rho)^2 + \frac{(2*\rho)^2}{2!*(1-\rho)}\right]^{-1}$

Die Verweildauer der Aufträge im System mit nur einer Warteschlange für beide Bedienelemente ist also um 27 % kürzer.

Dieses Beispiel zeigt also, dass eine zentrale Verwaltung der anfallenden Aufträge günstiger ist als eine getrennte pro Prozessor. In diesem Beispiel benötigt die Verwaltung und das Holen der Aufträge keine Zeit, in der Praxis müssen aber noch mehrere Variable berücksichtigt werden, wie zum Beispiel Speicherzugriffs- und Transferzeiten oder der Aufwand für den Taskwechsel im Prozessor bzw. der generelle Rechenaufwand für die Verwaltung.
Für das Daytona kommt erschwerend hinzu, dass es keinen großen von beiden Prozessoren gemeinsam adressierbaren Speicher hat, in dem ein Taskwechsel zwischen den Prozessoren leichter durchzuführen wäre. Kommunikation zwischen den Prozessoren ist für größere Datenmengen nur über den Hurricane Controllerchip möglich.
Es ist also zu untersuchen, wie viel des theoretischen Vorteils einer zentralen Verwaltung der Aufträge in der Praxis zu realisieren ist.

Generell würde ein mehr auf objektorientierten Prinzipien beruhender Aufbau des eRTOS den Ausbau auf ein mehrprozessorfähiges Betriebssystem erleichtern. Da hier ein leichteres Verständnis und eine einfachere Steuerung der Interaktion der einzelnen Elemente des Betriebssystems gegeben wäre.

5 Beispiele

In den Dateien zu dieser Studienjahresarbeit sind mehrere Beispiele enthalten die die Funktionen des eRTOS erläutern sollen.

5.1 Messwertspeicher und DMA Funktion

Dieses Beispiel beschäftigt sich mit dem Verwenden des DMA zur Datenübertragung und der Verwendung des Messwertspeichers zum Erfassen mehrerer Eingabedaten.

Benutzt werden die Funktionen des Messwertspeichers, der Taskverwaltung, der DMA Erweiterung, der LED und Keyboard Ansteuerung von Falk Berger sowie die Funktionen des Nachrichtensystems.

Das Beispiel liest die Tastatureingaben auf dem Beispielboard in eine Variable und überträgt den Inhalt dieser Variable über DMA in einen anderen Speicherbereich. Ist die Übertragung abgeschlossen, wird der Wert aus diesem Speicherbereich in den Messwertspeicher übertragen. Sind die zehn Elemente des Messwertspeichers gefüllt, beginnt ein zweiter nk-Task den Messwertspeicher in ein Array auszulesen. Der Inhalt dieses Arrays wird dann per DMA in ein anderes Array übertragen.

Vorbereitung:
Wie im Kapitel Implementierung beschrieben muss in der Config.h das Nachrichtensystem, der Messwertspeicher, die DMA Erweiterung und die D-Modul Option aktiviert werden.

```
#define DMA
#define MSG
#define D_MODULE
#define MESSW
#ifdef MSG
#define MsgBufLength 20
#endif
```

Die Ticklänge beträgt 600
```
#define ticklange 600
```

Es werden zwei k-Tasks und zwei nk-Tasks genutzt. Plus 2 nktasks für Server und Resuspendtask.

```
#define ktasks  2
#define nktasks 4
```

Zusätzlich sind die weiteren Vorbereitungen zur Verwendung der DMA Funktionen zu erledigen.

In das Projekt müssen auch die Dateien DMessBsp.c, keyboard.c und leddriver.c eingebunden werden. Die Dateien keyboard.c und leddriver.c stellen Funktionen bereit die den Zugriff auf das LED Display und die Tastatur erleichtern. Die Datei DMessBsp.c enthält das Beispiel.

Beschreibung:

Die Main() Funktion enthält die Zuweisung der Tasks und die Initialisierung des Messwerspeichers sowie des DMA Servers.

```
void main()
{
        rtos_init();

/*      Zuweisung der kTasks    */
        create_ktask(&LED_Driver, TID_Led_Driver, 4, 0);
        create_ktask(&Tastatur_Driver, TID_Tastatur_Driver, 32, 2);

/*      Zuweisung der nkTasks  */
        create_nktask(TID_TestTaster, &TestTaster, 2, -1);
        create_nktask(TID_LiesDaten, &LiesDaten, 2, -1);

/*      DMAinitialisierungen    */
        init_dma(TID_DMA_Server,TID_DMA_Resuspend, USE_DMA3|USE_DMA0);
        //Zur DMA Übertragung werden die DMA Kanaele 0 und 3 benutzt

/*      Messwertspeicher initialisierungen */
        ms_create(PUFFER,1);
         //Es werden zehn 32 Bit (int) Elemente angelegt

/*      Aktivieren des Interrupts   */

/*      Aufruf des Scedulers    */
   OSystem();
}
```

Die Funktion Int2Disp() von Falk Berger stellt Integerwerte auf dem LED Display dar.

Die Funktion TestTaster liest die Tastatureingaben auf dem Beispielboard in eine Variable und überträgt den Inhalt dieser Variable über DMA in einen anderen Speicherbereich. Ist die Übertragung abgeschlossen, wird der Wert aus diesem Speicherbereich in den Messwertspeicher übertragen. Sind die zehn Elemente des Messwertspeichers gefüllt, wird eine Nachricht an LiesDaten geschickt, damit der Task aktiv wird.

```
void TestTaster(void)
{
        int Nachricht[3]={0,0,0};
        int DMAReplyTest;

        int taste=ReadKey();
        //Taste wird vom Board eingelesen.
        if (taste!=-1)
        {
           dma_source=taste;
           //muss kopiert werden da nicht fest steht wann der Inhalt über den DMA-Kanal übertragen wird
           //und sich der Inhalt von Tasten schon wieder geändert haben kann.
           sendDMAMessage((int)&dma_source,(int)&dma_dest,4,dma_want_reply,7);
           //Der Inhalt der Adresse dma_source wird an die Adresse dma_dest mit Priorität 7 kopiert.
           //Dabei wird Quell und Zieladresse pro übertragenen Byte erhöht
           //und es wird eine Bestätigung der Übertragung verlangt.
        }
        DMAReplyTest=askmessage(0,Nachricht);
        //Falls das Flag dma_want_reply gesetzt ist muessen die Nachrichten auch abgefragt werden,
        //andernfalls läuft der Messagebuffer voll.
        if (DMAReplyTest!=-1)
        //Wurde die Übertragung bestätigt?
        {
                if (ms_enqueue(&dma_dest)!=0)
                //Füge dem Messwertspeicher ein Element zu.
                //Falls er voll sein sollte.
                {
                        SetText("-SPVOLL-");
                        //Gib Nachicht auf Display aus
                        Nachricht[1]=111;
                        //Das setzen des Inhaltes kann auch weggelassen werden
                        //da LiesDaten den nicht auswertet.
                        sendmessage(TID_LiesDaten,0,Nachricht);
                        //an Liesdaten wird eine Nachricht über Kanal 0 verschickt damit
                        //Liesdaten aus dem Wartezustand erwacht.
                }
                //Falls nicht stelle Wert auf Display dar.
                else Int2Disp(dma_dest);
        }
}
```

Die Funktion LiesDaten() entnimmt dem Messwertspeicher Elemente und legt deren Inhalt in einem Array ab. Der Inhalt dieses Arrays wird dann per DMA in ein anderes Array übertragen.

```
void LiesDaten(void)
{
        int Nachricht[3];
        int i=0;

        if (askmessage(0,Nachricht)==-1) warten(TID_LiesDaten,NKS_MSG,TID_TestTaster,0);
        //Prüfe ob eine Nachricht auf Kanal 0 vorhanden ist.
        //Wenn nicht gehe in Zustand "warten auf Nachricht" von TestTaster auf Kanal 0
        while (ms_isempty()==-1)
        {
                ms_dequeue(&Eingabe[i]);
                i++;
        }
        //solange der Messwertspeicher nicht leer ist soll ein Element entnommen
        //und im Eingabe Array abgelegt werden.
  sendDMAMessage((int)&Eingabe,(int)&Speicher,10*4,0,0);
  //Übertragen des Inhalts vom Array Eingabe in das Array Speicher.
}
```

5.2 Betriebsmittelverwaltung

Dieses Beispiel beschäftigt sich mit der Betriebsmittelverwaltung des eRTOS.

Das Beispiel erzeugt mehrere Tasks die überschneidend auf die verwalteten Betriebsmittel zugreifen. Zur Veranschaulichung wird der Fortschritt auf dem Beispielboard angezeigt. Wenn die Belegung erfolgreich war wird die entsprechende LED aktiviert und die dem Task zugeordnete Segmentanzeige verändert.

Vorbereitung:
Wie im Kapitel Implementierung beschrieben muss in der Config.h die Betriebsmittelverwaltung und die D-Modul Option aktiviert werden.

#define D_MODULE
#define BMV
#define BMANZAHL 10

Die Ticklänge beträgt 600
#define ticklange 600

Es werden vier k-Tasks und drei nk-Tasks genutzt

#define ktasks 4
#define nktasks 3

Zusätzlich sind die weiteren Vorbereitungen zur Verwendung der DMA Funktionen zu erledigen.

In das Projekt muss auch die Datei BMVBsp.c eingebunden werden.

Beschreibung:
Die k-Tasks BelegeKT1 und BelegeKT2 belegen abwechselnd das Betriebsmittel R1.
BelegeKT3 und BelegeNKT1 sowie BelegeNKT2 belegen das Betriebsmittel R3.
BelegeNKT2 zusätzlich noch das Betriebsmittel R2.
BelegeR2KT erzeugt sich ein neues Betriebsmittel R2 das es zusammen mit BelegeR2NKT und BelegeNKT2 belegt.

Dies ist der Prototyp der Funktionen die die Betriebsmittel belegen.

```
void BelegeKT(void)
{
 if(occupy_bm(R1)==0)
 //Versucht das Betriebsmittel R1 zu belegen. Wenn es nicht frei ist muss das Task trotzdem
 //terminieren.
        {
                AnimateSEG(&SEGS[0],0);
                //Animiere Displaysegment 0 (ganz links)
                WRITE_EB_PORT(EB_LED, LEDS|=0x0080);
                //Aktiviere linke LED (Betriebmittel belegt)
                if (free_bm(R1)==0) WRITE_EB_PORT(EB_LED, LEDS&=0x007F);
                //gib Betriebsmittel wieder frei und loesche linke LED.
        }
}
```

Der Status des Betriebsmittels wird durch eine zugeordnete LED angezeigt. Wenn das Betriebsmittel belegt ist leuchtet die LED. Die dem Task zugeordnete Segmentanzeige wird pro erfolgreicher Belegung animiert, so dass die Aktivität des Tasks sichtbar wird.

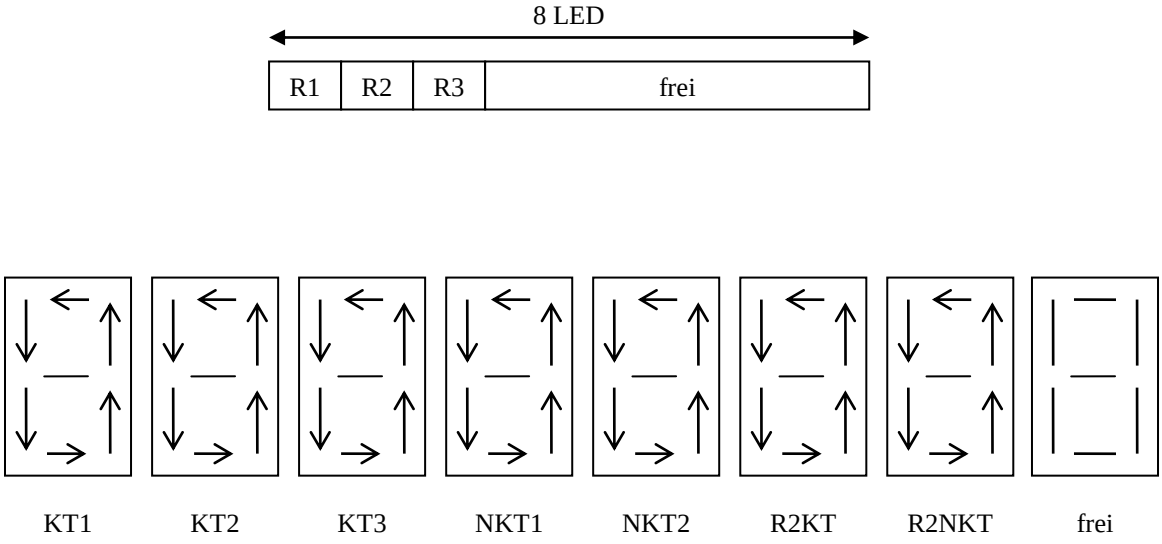


Abb. 33 Segmentzuordnung der Tasks und Belegung der LEDs

5.3 Nachrichtensystem

Dieses Beispiel beschäftigt sich mit dem Nachrichtensystem des eRTOS und dessen Dualprozessorerweiterung.

Das Beispiel läuft auf zwei unterschiedlichen CPU die Tasks auf diesen CPUs senden sich gegenseitig Nachrichten. Die Zählvariable der CPU 1 hat dabei immer den halben Wert der Zählvariable auf CPU 2.

Vorbereitung:
Vor dem Starten des Codecomposers ist das Programm **jtagboot.exe** auszuführen und die Option **b** zu wählen. Danach kann der Parallel Debug Manager des Codecomposers gestartet werden und für jede CPU muss ein eRTOS Projekt geladen werden.

Die Vorbereitung muss für jede CPU erledigt werden.
Wie im Kapitel Implementierung beschrieben, muss in der Config.h das Nachrichtensystem für interne und externe Nachrichten aktiviert werden.

```
#define MSG
#define MPRAM

#ifdef MSG
#define MsgBufLength 20
#endif
#ifdef MPRAM
        #define MPRAMBufLength 20
        #define Prozessor x
        //je nach CPU 1 oder 2
#endif
```

Die Ticklänge beträgt 600
```
#define ticklange 600
```

Es werden eine k-Tasks und zwei nk-Tasks auf CPU 1 genutzt und eine k-Task sowie eine nk-Task auf CPU 2. Es ist darauf zu achten, die genaue Anzahl der Tasks anzugeben, andernfalls treten Fehler in der sendmessage() Funktion auf.

```
#define ktasks entsprechend
#define nktasks entsprechend
```

Die Linker.cmd Datei für das Daytona Board muss in den Ordner \lnk kopiert werden. Und die Datei NSBspCPU1.c und NSBspCPU2.c muss dem Projekt der entsprechenden CPU zugefügt werden.

Es ist zu beachten das für eine korrekte Initialisierung des Dual Port Rams ein Breakpoint bei OSystem() in der Main() Routine gesetzt werden muss und die Projekte der beiden CPUs müssen jeweils bis zu diesem Punkt ausgeführt werden, bevor sie fortgesetzt werden können.

Beschreibung:
Der Abfolge der Nachrichten in diesem Beispiel kann grob so beschreiben werden.

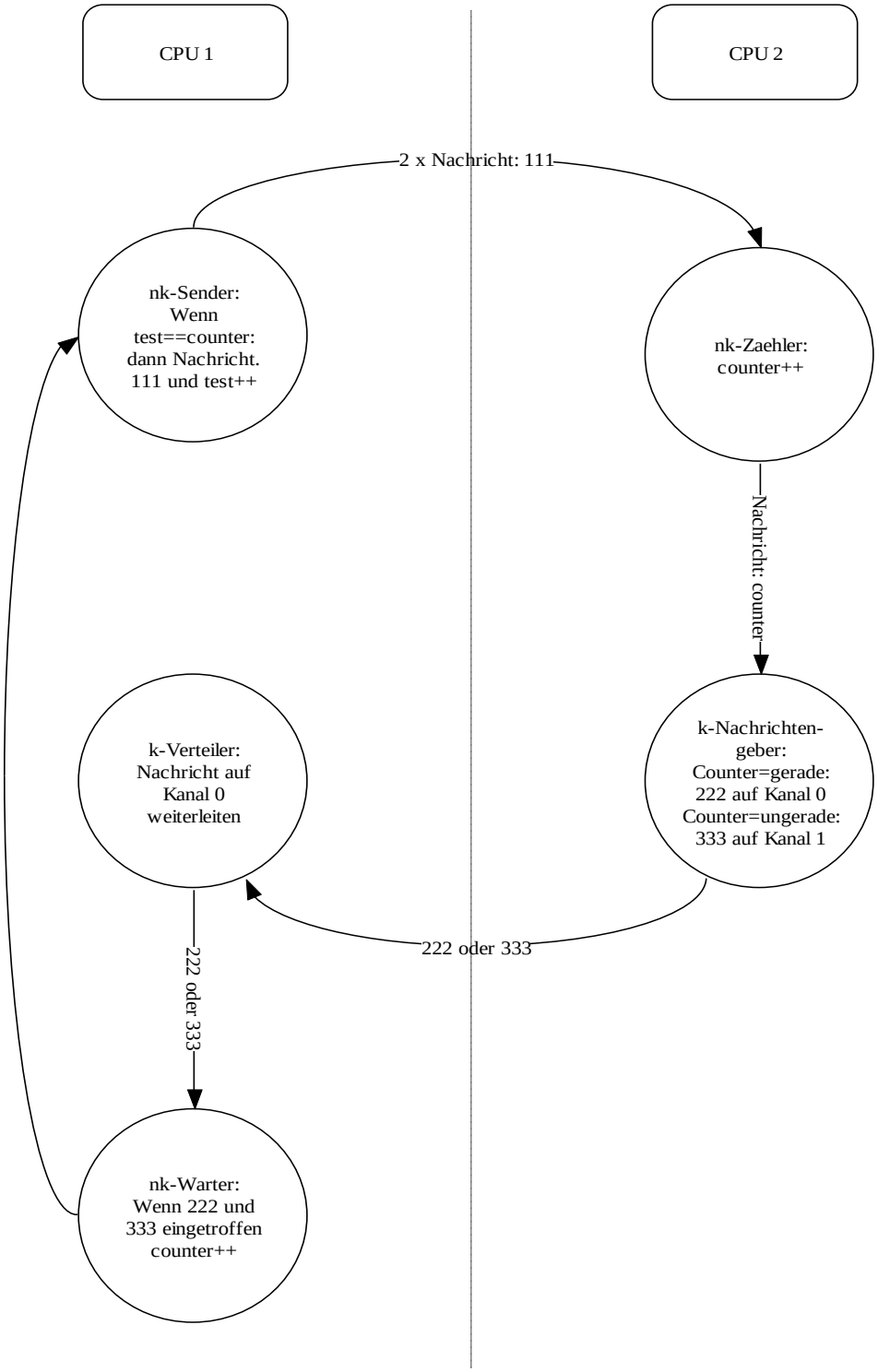


Abb. 34 Prinzip des Nachrichtensystembeispiels

Sender() auf CPU 1 schickt 2 externe Nachrichten an Zaehler auf CPU 2 der pro Nachricht die counter Variable erhöht. Der Wert der Countervariable wird per interner Nachricht an Nachrichtengeber() geschickt. In Abhängigkeit des Wertes von counter verschickt dieser

Nachrichten an Verteiler() auf CPU 1 über Kanal 1 oder 0. Der Verteiler() Task leitet diese Nachrichten auf Kanal 0 weiter an den Warter() Task. Ist dort Nachricht 222 und 333 eingetroffen erhöht dieser den Counter und der Kreislauf beginnt von neuem.

5.4 Taskverwaltung

Dieses Beispiel beschäftigt sich mit den Funktionen der Taskverwaltung die das eRTOS zur Verfügung stellt.

Benutzt werden die Funktionen der Interruptverwaltung und der Taskverwaltung. Zur Veranschaulichung wird der Fortschritt der Tasks auf dem Beispielboard angezeigt.

Vorbereitung:

#define D_MODULE

Die Ticklänge beträgt 600
#define ticklange 600

Es werden zwei k-Tasks und zwei nk-Tasks genutzt

#define ktasks 1
#define nktasks 3

In der Datei Assem.asm muss der Interrupt 5 aktiviert werden

nint5 .set 1

Die Datei TaskBsp.c muss in das Projekt eingebunden werden.

Beschreibung:
Timer 1 wird initialisiert und gestartet. Pro Timer 1 Interrupt 5 wird ein Zähltask gestartet der eine Zählvariable erhöht. Der KontrollKT Task startet suspendiert und resuspendiert den LEDNKT Task der für den Inhalt der Countervariable auf dem LED Feld ausgibt und den WechslerNKT Task. Der WechslerNKT Task ändert seine Periode in Abhängigkeit von der counter Variable . Der Fortschritt der Tasks wird auf der Segmentanzeige wiedergegeben.

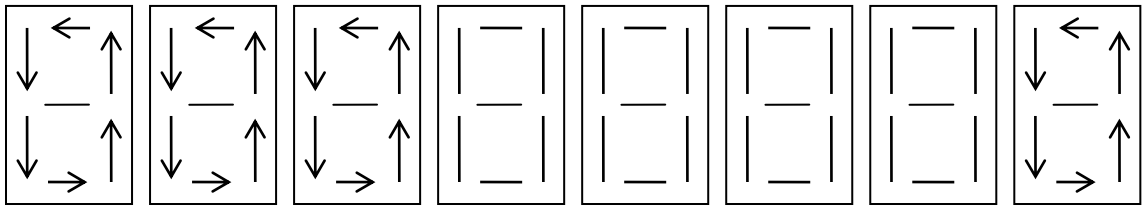


Abb. 35 Taskzuordnung auf der Segmentanzeige

Die Funktionen des Taskverwaltungsbeispiels.

```
void KontrollKT(void)
{
        if (counter%2)
        if (nkzustand(TID_LEDNKT)==NKS_SUSPENDIERT)
        resuspend(TID_LEDNKT); else suspend(TID_LEDNKT);
        //Suspendieren und Resuspendieren des LEDNKT Tasks zur LED Ausgabe
        if (nkzustand(TID_WechslerNKT)==NKS_SUSPENDIERT)
        resuspend(TID_WechslerNKT);  else suspend(TID_WechslerNKT);
        //Suspendieren und Resuspendieren des Wechsler Tasks

        if ((nkzustand(TID_LEDNKT)!=NKS_WARTEN)
          &&(nkzustand(TID_LEDNKT)!=NKS_SUSPENDIERT))
              warten(TID_LEDNKT,NKS_ZEIT,200,0);
        //Falls der LEDNKT Task ausführbar ist, soll er 200 Ticks warten.

        AnimateSEG(&SEGS[0],0);
        //Animiere Displaysegment 0 (ganz links)

}
void ZaehlerNKT(void)
{
        counter++;
        AnimateSEG(&SEGS[7],7);
        //Animiere Displaysegment 7 (ganz rechts)
        //end_int(5);
        //Task wird suspendiert und wird erst bei erneitem Auftreten
        //des Interrupts 5 wieder ausgeführt
}

void WechslerNKT(void)
{
        chng_prio(TID_WechslerNKT,counter%2);
        //ändert seine Priorität in Abhängigkeit von Counter
        AnimateSEG(&SEGS[1],1);
        //warten(TID_WechslerNKT,NKS_ZEIT,20,0);}

void LEDNKT(void)
{
        WRITE_EB_PORT(EB_LED, counter);
        //Ausgabe des Wertes von Counter Binär auf den LED Feld
        AnimateSEG(&SEGS[2],2);
}
```

```
void main()
{
rtos_init();

/*          Zuweisung der kTasks     */

create_ktask(&KontrollKT, TID_KontrollKT, 4, 0);
//Periode 4 und Startick 0


/*  Zuweisung der nkTasks  */

create_nktask(TID_ZaehlerNKT, &ZaehlerNKT, 1, 5);
//Wird Interrupt 5 zugeordnet und
create_nktask(TID_WechslerNKT, &WechslerNKT, 1, -1);
//Priorität 1
create_nktask(TID_LEDNKT, &LEDNKT, 1, -1);
//Priorität 1

/*          LED Initialisierung */

WRITE_EB_PORT(EB_LED, LEDS);
//Löscht die LEDs
WRITE_EB_PORT(EB_SEG, 0x0000);
//Löscht das Display

/*          Aktivieren des Interrupts   */

enableint(5);

/*  Initialisieren des Timer1   */

setperiod1(1500);
inittimer1();
timergo1();


/*          Aufruf des Scedulers        */
OSystem();
}
```

6 Befehlsverzeichnis

7 Abbildungsverzeichnis

8 Quellen

[Sch01] Sebastian Schmidt, Konzeption und Realisierung eines dynamisch konfigurierbaren Echtzeitbetriebsystems für den Einsatz in einem Messwerterfassungssystem, Diplomarbeit TU Ilmenau, 2001, 2001-11-01/0011/IN95/2231

[Kra02] Andreas Kraus, DMA Unterstützung im eRTOS, , Studienjahresarbeit, TU Ilmenau, 2002

[Lev00] Marc Leverenz, eRTOS Analyse, Studienjahresarbeit, TU Ilmenau, 2000

[Lev02] Marc Leverenz, Konzeption und Realisierung von Betriebssystemsoftware für ein Dual-Port-DSP, Diplomarbeit, TU Ilmenau, 2002, 2002-06-03/024/IN95/2231

[SchLev01] Sebastian Schmidt, Marc Leverenz, eRTOS 1.4 für TMS320C6x, Institut Theoretische und Technische Informatik, TU Ilmenau, 2001

[Rim02] Kirke Rimbach, Modellbasierte Analyse und Bewertung von Echtzeitbetriebssystemen, Diplomarbeit TU-Ilmenau 2002, 2002-12-02/054/IN97/2231

[Specweb] Daytona Specification, Spectrum, http://www.spectrumsignal.com/Products/_Product_PDFs/Daytona.pdf, 2002

[TIweb] TMS320c6x Specification, Texas Instruments, http://dspvillage.ti.com/docs/catalog/dspplatform/overview.jhtml?templateId=5154&path=templatedata/cm/dspovw/data/c6000_ovw, 2002

[TIB] Handbücher zum TMS320c6x, Texas Instruments, 2000

[RAweb] TU Ilmenau, Fakultät für Informatik und Automatisierung, Institut für Theoretische und Technische Informatik, Fachgebiet Rechnerarchitekturen, http://www.theoinf.tu-ilmenau.de/nanomess/, 2001

[DModweb] D.SignT Digital Signalprocessing Technology, http://www.dsignt.de/products/dmodule/d6701.htm, 2002

[Resch00] Reschke, D.; Krüger, G.; Lehr und Übungsbuch Telematik, 1. Aufl., Fachbuchverlag Leipzig, Leipzig, 2000