ISSN 0280–5316
ISRN LUTFD2/TFRT--5558--SE

Konzeption und Realisierung eines Echtzeitbetriebssystems für strukturadaptive Regelsysteme

Johan Hellqvist

Department of Automatic Control
Lund Institute of Technology
March 1996

Konzeption und Realisierung eines Echtzeitbetriebssystems für strukturadaptive Regelsysteme

Diplomarbeit

von

Johan Hellqvist

11. September 1995 - 11. März 1996

Institut für Prozessrechentechnik und Robotik
Prof. Dr.-Ing. U. Rembold
Prof. Dr.-Ing. R. Dillmann
Universität Karlsruhe, Deutschland

Department of Automatic Control
Prof. K.-J. Åström
Prof. B. Wittenmark
Lund Institute of Technology (LTH), Schweden

Referent: Prof. Rolf Johansson, LTH
Koreferent: Prof. Dr.-Ing U. Rembold
Betreuer: Dr. Tim Lüth, Forschungszentrum Informatik (FZI), Karlsruhe
und Dr. Klas Nilsson, LTH

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name
MASTER THESIS

Date of issue
March 1996

Document Number
ISRN LUTFD2/TFRT--5558--SE

Author(s)
Johan Hellqvist

Supervisors
T. Lüth (FZI), K. Nilsson, U. Rembold, R. Johansson

Sponsoring organisation

Title and subtitle
Konzeption und Realisierung eines Echtzeitbetriebssystems für strukturadaptive Regelsysteme (Design and Implementation of a Real-Time-Operating-System for Structure-Adaptive Control Systems).

Abstract

In the control of mobile autonomous robots a number of static control-architectures with their own characteristics, for example classical hierarchic and parallel overloading (behaviorbased) control-architectures, can be found. Experiences have shown that the control-architecture is as important as the control-algorithms and that there is a need of combining and changing control-architectures in run-time.

A structure-adaptive control-architecture alters the information- and control-flow in run-time in order to obtain a control-structure which changes depending on the actual mission and states.

The above concept is supported by the real-time-robot-operating-system CAIC (Cooperative Architecture for Intelligent Control). CAIC was implemented and used to control two mobile miniature manipulators (Khepera-robots). The two robots were able to cooperatively grip, raise and transport a rectangular object (the spacing-piece in the Cranfield Assembly Benchmark). They communicated with local IR-communication to send commands and acknowledgements.

Key words

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280–5316

ISBN

Language
German

Number of pages
81

Recipient's notes

Security classification

Ich versichere hiermit, die vorliegende Diplomarbeit selbständig und ohne fremde Hilfe angefertigt zu haben.

Die verwendeten Hilfsmittel und Quellen sind im Literaturverzeichnis vollständig aufgeführt.

Karlsruhe, den 11. März 1996

Johan Hellqvist

Inhaltsverzeichnis

Kapitel 1

Einleitung

Regelungs- und Steuerungssysteme werden komplexer und erreichen höhere Abstraktionsgrade. Die Entwicklung verlief von ersten industriellen Regelungen der Dampfmaschine im 19.Jahrhundert bis hin zu den heutigen Steuerungs- und Regelungssystemen, die ganze Fabriken steuern sollen. Sie sollen eigene Entscheidungen treffen und intelligent auf Funktionsfehler reagieren,
Moderne Systeme sollen auch Auf- und Umbau mit möglichst wenig Aufwand unterstützen. In der modernen Fertigungsindustrie gibt es einen Bedarf für flexible Fertigungssysteme und intelligente Roboterkooperation, z.B. beim Produktionslinienwechsel. Eine gute Entwicklungsplattform bieten die autonomen mobilen Roboter, die auch in der Fertigung eingesetzt werden können.
Autonome mobile Roboter sind in einer komplexen Umwelt tätig, sollen mehrere Aufgaben lösen, unerwartete Systemzustände umgehen, redundant sein und auf Funktionsfehler intelligent reagieren.
Heute gibt es einen Bedarf für Echtzeitbetriebssysteme, die die erwähnten Anforderungen beachten und unterstützen können.
Der Entwurf intelligenter Steuerungen wird auch in der Zukunft immer mit experimenteller Entwicklung und iterativen Veränderungen verbunden sein. Eine universelle Steuerungsarchitektur wird es nie geben, da mit jedem zusätzlichen Sensor und Aktor neue Steuer- und Regelalgorithmen implementiert und getestet werden müssen. Typischerweise verlagern sich bei der Verwendung komplexer Sensoren und Aktoren immer mehr Bestandteile der Architekturen in die Peripherie der Steuerung. Damit verbunden ist eine immer stärkere Nebenläufigkeit von Informationsverarbeitungsprozessen.
Steuerungen autonomer Roboter sind komplexe Systeme, in denen viele unabhängig voneinander entworfene zeit- und ereignisbasierte Regelkreise zusammenwirken. Beim gegenwärtigen Stand der Technik entscheidet bereits nicht mehr die Qualität des einzelnen Algorithmus über die Leistungsfähigkeit, sondern vielmehr die Steuerungsarchitektur. Daher muß sie systematisch betrachtet und speziellen Aufgabenstellungen angepaßt werden. Dies kann auch zur Laufzeit erforderlich werden.
In dieser Arbeit wird das Konzept der strukturadaptiven Steuerungsarchitekturen vorgestellt, mit dem dynamische Veränderungen des Informations- und Steuerflusses zur Laufzeit vorgenommen werden können, um situationsabhängige hierarchische oder verhaltensorientierte Steuerstrukturen zu erzeugen. Unterstützt wird das Konzept

durch ein spezielles Echtzeitbetriebssystem.

1.1 Problemstellung

Autonome Roboter sind komplexe Systeme, die ihre Umwelt modellieren, beobachten und erfassen, um einen Zielzustand zu erreichen oder zu stablisieren. Erfahrungen zeigen, daß solche Systeme *iterativ* entwickelt werden. Regelsysteme tendieren dazu, mit der Zeit nach erweiterten Spezifikationen *komplexer* zu werden. Der iterative Entwurf und die Steigerung der Komplexität führen sehr häufig zu unüberschaubaren Systemen, in denen kleine Veränderungen einen großen Integrationsaufwand fordern. Solche Systeme sind darüberhinaus schwer *änder-* und *erweiterbar*. Ein anderer Folgeeffekt ist, daß alte Teile in der späteren Version des Systems nicht mehr benutzt werden können.
In allen bisherigen Ansätzen wurden Steuerungsarchitekturen als statische Strukturen betrachtet, die nur einmal als festes Netzwerk aus Informationskanälen und Verarbeitungseinheiten entworfen werden. Ein System, das autonom wirken und möglichst viele Aufgaben lösen soll, muß zur Laufzeit für die aktuelle/-en Aufgabe/-en die optimale *Steuerungsarchitektur* bereitstellen können. Bei einer freien Bewegung im Arbeitsraum muß ein autonomer Roboter trotz Funktionsstörungen und/oder -fehler funktionieren und seine Aufgaben möglichst gut erledigen. Das System muß eine eingebaute *Redundanz* besitzen, die ohne redundante Hardwareteile funktioniert.
Neue aktive Sensoren, leistungsfähige billige Prozessoren, Multi-Roboter-Kooperation, ein höherer Grad von Parallelität usw. erfordern, daß das System *verteilt* ist. Änderungen der Steuerungsarchitektur, unerwartete Zustände, Fehlerbehebung und Aufgabenwechsel müssen zur Laufzeit *dynamisch* erfolgen.

1.2 Zielsetzung der Arbeit

Zielsetzung der Arbeit ist es, ein Roboter-Betriebssystem zu entwickeln, das

- die schnelle Implementierung von komplexen Regelsystemen ermöglicht,
- hierarchische und nebenläufige Architekturen kombinieren und zur Laufzeit die Struktur ändern kann,
- Regelsystemänderungen und Architekturwechsel zur Laufzeit ermöglicht,
- das Regelsystem auf mehrere Prozessoren, Geräte oder Roboter verteilen kann,
- Steuerfluß und Informationsfluß explizit trennt und
- den Steuerfluß während der Laufzeit verändert.

1.3 Aufbau und Kapitelübersicht

Nach der Einordnung der Arbeit, der Beschreibung ihrer Zielsetzung und Gliederung folgt im zweiten Kapitel die Theorie der Echtzeitdatenverarbeitung und ein kurzer Einblick in zwei verschiedene Steuerungsarchitekturen im Bereich autonomer mobiler Robotersteuerung.
Der Stand der Technik im Bereich der Echtzeitbetriebssysteme wird im dritten Kapitel vorgestellt.
Im vierten Kapitel wird das Konzept des *CAIC*-Systemes beschrieben.
Inhalt des fünften Kapitels ist die Beschreibung der Zielplattform, der Khepera--Roboter.
Im sechsten Kapitel wird *CAIC* Presentation System, die graphische Oberfläche und Kommunikationsschnittstelle des *CAIC* auf dem Khepera-Roboter vorgestellt.
Im Kapitel sieben werden Experimentergebnisse aufgeführt und zwei Beispiele erklärt. Gezeigt wird, auch wie zwei Roboter kooperativ mittels lokaler Infrarotkommunikation ein Benchmarkteil greifen, heben und wegtransportieren.
Das achte Kapitel enthält eine kurze Zusammenfassung und bietet einen Ausblick auf mögliche Erweiterungen des vorgestellten Echtzeitbetriebssystems.
Handbuch für Programmierer, Funktionsbeschreibung und Quellcode steht in einem Ergänzungsband dieser Arbeit zu verfügung. Durch die ganze Arbeit sind Programmcode und Variablennamen in `courier` geschrieben. Instanznamen werden *kursiv* geschrieben.

Kapitel 2

Theorie der Echtzeitdatenverarbeitung

Der Begriff Echtzeit wird häufig im Zusammenhang mit schnellen Systemen verwendet. Echtzeit heißt, daß neben den Eingabedaten auch die Zeit zu berücksichtigen ist, siehe Abb. 2.1 und Abb. 2.2 [Lauber 1989].

Abbildung 2.1: Nicht-Echtzeit-Datenverarbeitung

Abbildung 2.2: Echtzeit-Datenverarbeitung

Die Norm DIN 44300 [DIN 1985] definiert den Begriff des Echtzeitbetriebs folgendermaßen: "Echtzeitbetrieb ist ein Betrieb eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind derart, daß die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zufälligen zeitlichen Verteilung oder zu vorbestimmten Zeitpunkten auftreten." [Herrtwitch 1989].
Es gibt zwei Arten von Zeitbedingungen:

- Forderungen nach Rechtzeitigkeit
- Forderungen nach Gleichzeitigkeit

Echtzeitsysteme müssen streng deterministisch innerhalb einer Zeitschranke auf ein externes oder internes Ereignis reagieren. Das Zeitkriterium, abhängig vom dem jeweiligen Prozeß, kann zwischen *ms* und *Minuten* liegen. Die Reaktionszeit t_R eines Systems setzt sich aus einer Wartezeit t_w und der Verarbeitungszeit t_v zusammen, Abb. 2.3. Um eine schritthaltende Verarbeitung zu gewährleisten, ist es bei einem Echtzeitsystem absolut notwendig, daß die Reaktionszeit t_R zu jeder Zeit kleiner als eine gewisse Grenzzeit t_{Rmax} ist. Die Grenzzeit t_{Rmax} soll auch höher als die schnellsten Prozeßereignisse t_p sein [Wollert 1996].

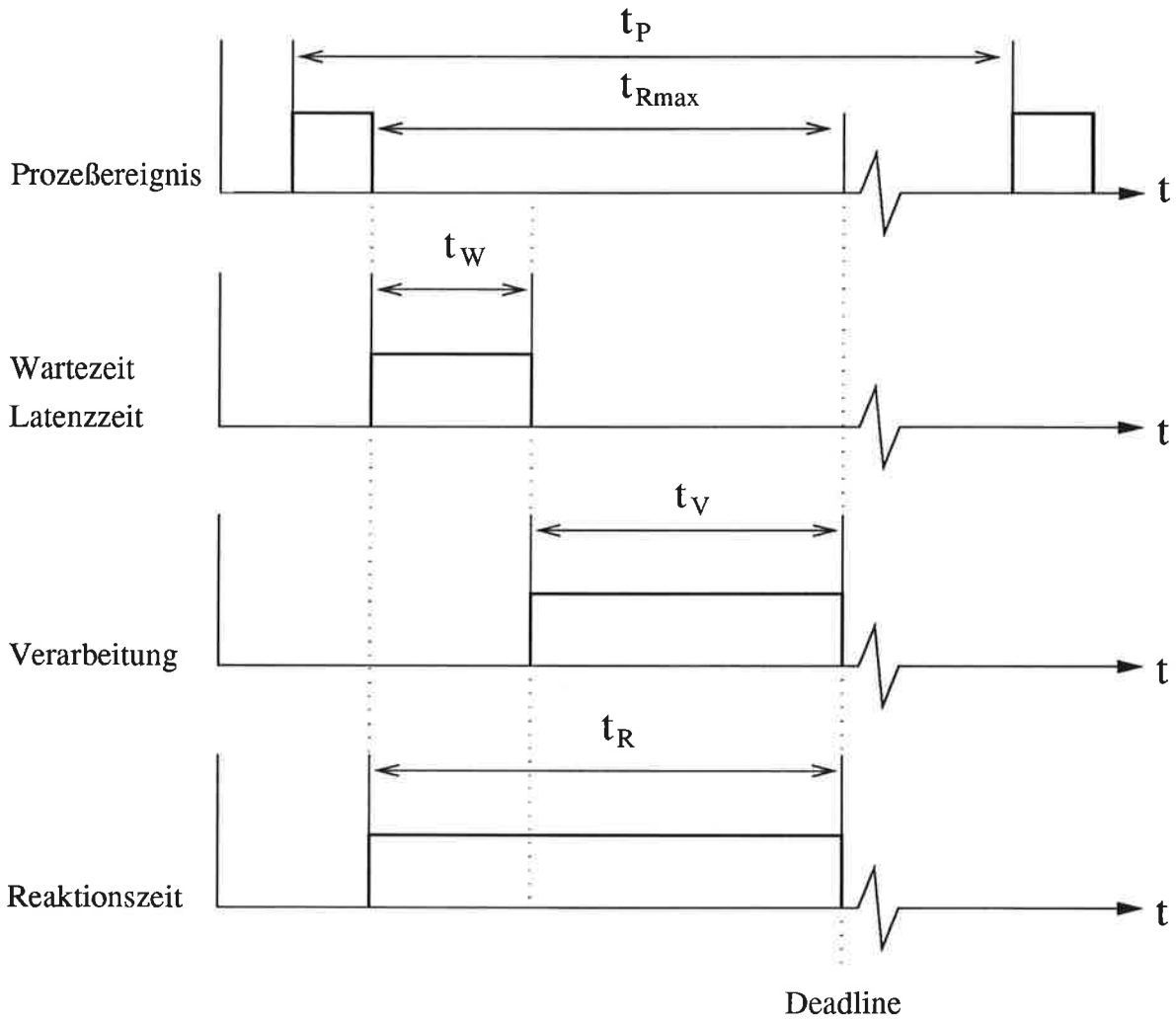

Abbildung 2.3: Definition der Zeiten in Echtzeitprozessen bzw. bei der schritthaltenden Datenverarbeitung

Ein weiterer Aspekt der Echtzeitdatenverarbeitung ist die Nebenläufigkeit von Prozessen, die zu den oben genannten Forderungen nach Gleichzeitigkeit führt. Die üblichen eingesetzten *von-Neumann*-Architekturen können Programme nur sequentiell abarbeiten. Man wünscht aber eine parallele Bearbeitung von Prozessen. Mit einem Einprozessor-Rechner spricht man von quasi-Parallelität, die durch einen Scheduler ermöglicht wird.

Der Scheduler teilt jedem Prozeß eine bestimmte Rechenzeit zu. Als Schedulingverfahren kommen verschiedene Verfahren, meistens heuristische, zum Einsatz.

2.1 Scheduling-Algorithmen

Der einfachste Scheduling-Algorithmus FCFS (First Come First Serve) arbeitet nach einem FIFO-Prinzip. Alle Ereignisse treffen in einer Warteschlange ein und werden nacheinander abgearbeitet. Eine deterministische Antwortzeit kann für solche Systeme nicht garantiert werden.
Static Cyclic Scheduling, SCS, ist vielleicht der älteste Schedulingsansatz und ist ein sogenannter *off-line*-Ansatz. Die Zeitschranken der Tasks werden vorher festgelegt und in einer Ausführungstabelle, *calendar*, geschrieben [Tindell 1996]. Die Tabelle wird im Betrieb zyklisch durchlaufen.
Im Round-Robin-Verfahren wird jeder Rechenprozeß in einer bestimmten Zeit bearbeitet, bevor er unterbrochen und der nächste Prozeß bearbeitet wird. Dieses Verfahren wird in der Regel in Time-Sharing-Systemen eingesetzt und hat in Prozeßrechensystemen weniger Bedeutung.
Beim kooperativen Multitasking [Wollert 1996] entscheidet jeder Prozeß selbst, wann er eine Weitergabe der Rechenzeit zulassen soll. Dazu wird in unregelmäßigen Abständen der Scheduler explizit aufgerufen. Ein Nachteil des Verfahrens ist mangelnde Robustheit bei fehlerhaften Prozessen und die Notwendigkeit, für einen echtzeitfähigen Einsatz die maximalen Abstände des Aufrufs festzulegen.

2.2 Prioritätsgesteuertes Multitasking

Die klassische Lösung für Echtzeitanwendungen ist das prioritätsgesteuerte Scheduling. Jeder Prozeß bekommt eine Priorität, wobei immer der Prozeß abgearbeitet wird, der die höchste Priorität hat. Sind mehrere Prozesse gleichzeitig aktiv, wird die Rechenzeit des Prozessors nach einem FCFS oder Round-Robin-Verfahren verteilt. Tritt während der Abarbeitung ein Interrupt eines Prozesses mit höherer Priorität auf, so wird der laufende Prozeß unterbrochen und der Prozeß mit der höheren Priorität abgearbeitet (siehe Abb. 2.4).

Nach Beendigung des Prozesses wird der alte Prozeß zu Ende geführt. Das zuvor beschriebene Verfahren bezeichnet man als Preemptive. Ein solches System nennt man ein preemtives Multitaskingsystem.
Neben den beschriebenen Systemen gibt es eine Menge von Systemen, die Prioritäten dynamisch verteilen, um eine bessere Verteilung der Rechenzeit zu erreichen. Rechenprozesse mit kürzerer Laufzeit, geringer Rechnerbelastung oder schneller Reaktionszeit sollten eine hohe Priorität erhalten, und Prozesse mit einer hohen Verarbeitungszeit sollten eine niedrige Priorität zugeteilt bekommen. Die Vorgabe der Prioritäten muß schon während der Codierung berücksichtigt werden.
Die Zuteilung von Rechenzeit und die Verteilung von Prioritäten spielt eine zentrale Rolle in Echtzeitsystemen. Ungünstige Wahl von Prioritäten kann zu Deadlocks führen

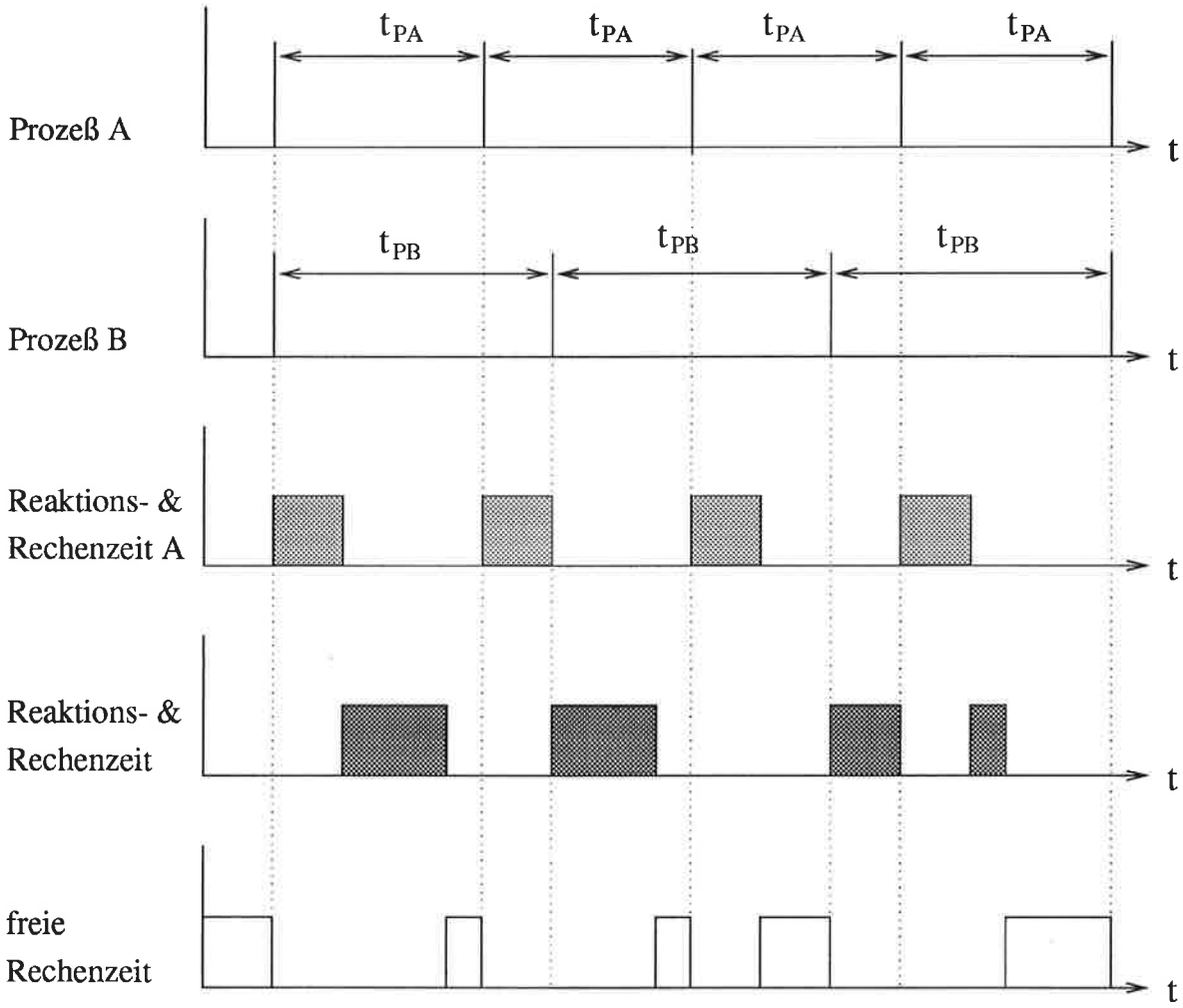

Abbildung 2.4: Preemtives Multitasking bei zwei zyklischen Prozessen

und das System zum Stillstand bringen. Ein klassisches Beispiel ist die Prioritäten-Inversion.

2.3 Prioritäten-Inversion

Bei der Verwendung von festen Prioritäten kann sowohl die Interprozeßkommunikation als auch der gemeinsame Zugriff auf Ressourcen zum Problem der Prioritäten-Inversion führen. Liegen drei Prozesse A, B und C mit den Prioritäten a, b und c vor, wobei gilt: a >b >c, so tritt im Falle des Zugriffs von A auf eine von C benutzte Ressource das Phänomen der Prioritäten-Inversion auf. Obwohl A eine höhere Priorität besitzt, muß erst die Ressource bei C freigegeben werden.
Eine allgemeine Lösung dieses Problems bietet die Vererbung von Prioritäten. Bei der Beanspruchung einer blockierten Ressource muß die aktuelle Priorität des beanspruchenden Prozesses an den Prozeß übergeben werden, der die Ressource blockiert. Das Prioritätenkonzept garantiert dabei, daß die Priorität des aktuellen Prozesses immer über der Priorität des blockierenden Prozesses liegt. Dieses Verfahren führt unter Beachtung der Prioritäten anderer Prozesse zur schnellstmöglichen Freigabe der Ressourcen. Bei der Interprozeßkommunikation kann ein analoges Vorgehen angewandt werden.

2.4 Dynamische Prioritäten

Neben der statischen Kopplung von Prioritäten an einen Prozeß ist auch die dynamische Übergabe von Prioritäten möglich. Dabei müssen sogenannte Deadlines betrachtet werden. Eine Zuteilung der Rechenzeit nach Deadlines erfordert zusätzliche Rechenschritte zur Laufzeit und Kenntnisse über die Rechenzeit, die eine Operation benötigt. Diese Rechenzeit ist normalerweise ziemlich schwierig und vor allem unsicher zu schätzen.
Letztere Problematik hat zu einer geringen Akzeptanz dynamischer Prioritäten geführt. Die üblichen Echtzeitsysteme arbeiten aus diesem Grund mit statischen Prioritäten. Dabei wird in Kauf genommen, daß eine garantierte Aussage über die Reaktionszeit nur für die Prozesse mit den höchsten Prioritäten gemacht werden kann. Ein theoretisches Rahmenwerk zur Verifikation für periodische Prozesse bietet das Verfahren Rate Monotonic Analysis (RMA). Die Voraussetzung ist, daß die Laufzeit bekannt ist. Die höchste Priorität wird hierbei dem Prozeß mit dem kürzesten Zeitrahmen verliehen. Dieses Verfahren ist optimal für Systeme mit statischer Prioritätszuordnung und führt zu einer oberen Prozessorauslastung von etwa 70 %.

2.5 Harte und weiche Echtzeit

In vielen Anwendungen ist die Einhaltung von strengen Zeitschranken, Deadlines, unabdingbar. Bei der Darstellung von Daten auf Visualisierungssystemen in Echtzeit sind die Zeitschranken nicht mehr so streng. Das führt zu der Aufweichung des strengen Begriffs Echtzeit in „harte" und „weiche" Echtzeitanforderungen. Es geht wie üblich in diesem Fall nicht nur um Zeitschranken und optimale Systeme, sondern auch um

Kosten, Abb. 2.5.

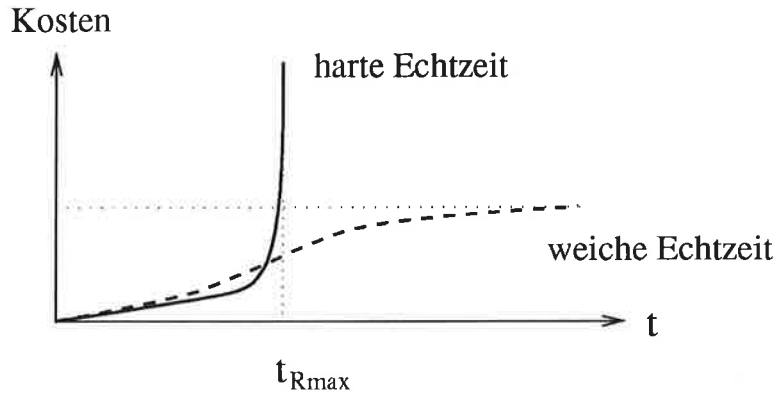

Abbildung 2.5: Harte und weiche Echtzeitbedingungen folgen einer Kostenfunktion [Wollert 1996]

Entsteht bei der Überschreitung der Deadline in der Anwendung ein erheblicher Schadensumfang, oder wird das System instabil, so sind harte Anforderungen zu stellen. Bei harten Echtzeitsystemen ist eine Verletzung der Zeitschranken mit einem Versagen des Rechensystems gleichzusetzen. Bei einem weichen Echtzeitsystem sind die Zeitschranken einzuhalten. Eine Überschreitung der Zeitschranken führt nicht zu fatalen Systemzuständen, sondern eine Verletzung der Echtzeitbedingungen kann in Ausnahmesituationen toleriert werden. Die Garantie der Einhaltung von Zeitschranken erfordert in der Regel ein für den Normalbetrieb erheblich überdimensioniertes Rechensystem.

2.6 Intelligente Steuerungen für autonome Roboter

Die Steuerung von autonomen Robotern ist ein hochkomplexes und vielseitiges System. Typischerweise soll ein autonomes Fahrzeug selbst navigieren und auf unerwarte Zustände „intelligent" reagieren. Die wissenschaftliche Fokussierung auf autonome Fahrzeuge hat in der letzten Zeit stark zugenommen, aber die Entwicklung intelligenter Systeme wurde schon 1948 begonnen. Unten folgt eine chronologisch geordnete Liste von wichtigen Ergebnissen im Bereich intelligenter Steuerungen für autonome Roboter [Lüth 1996].

- 1948, Wiener: Grundlagen der Kybernetik
- 1951, Shannon: Mikromaus im Labyrinth
- 1961, Walter: Elmar und Elsie, dezentrale analoge Regelkreise
- 1969, Nilson: Zentrale diskrete Zustands-/Aktionsmodelle (KI)
- 1971, Fikes: Zentrale Suche in diskreten Zustandsräumen
- 1979, Hayes-Roth: Dezentrale Suche in zentralen Zustandsräumen

- 1981, Albus: Hierarchische Steuerungen
- 1984, Braitenberg: Erweiterungen der Arbeiten von Walter
- 1986, Brooks: Nebenläufige, ereignisgesteuerte Regelkreise, Verhalten
- 1987, Georgeff: Echtzeit-Planungssysteme
- 1988, Thorpe: Dezentrale Echtzeit-Planungssysteme
- 1990, Arkin: Überlagerung der Stellgrößen nebenläufiger Regelkreise
- 1993, Musliner: Lastadaptive verteilte Echtzeitregelsysteme
- 1993, Bajscy: Zeit-/ereignisdiskrete Robotersteuerungen

2.6.1 Steuerungsarchitekturen für autonome Roboter

In dem Bereich für Steuerung von autonomen Fahrzeugen gibt es zwei häufig benutzte Steuerungsarchitekturen: hierarchische und verhaltensorientierte bzw. nebenläufige Steuerungsarchitekturen, Abb. 2.6.
Hierarchische Architekturen bieten durch eine zentrale Steuerung maximale Flexibilität. Die Funktionsweise des Roboters wird situationsabhängig zur Laufzeit festgelegt. Ihr Nachteil, die begrenzte Verarbeitungsgeschwindigkeit und die erzwungene Synchronisation durch ein globales Modell, fallen bei dezentralen, verhaltensorientierten Architekturen weit weniger ins Gewicht. Deren Nachteil ist die schwer vorhersagbare und erklärbare Funktionsweise, die mit dem experimentell basierten und iterativen Entwickeln von Regelkreisen zusammenhängt. Diese Regelkreise können sich zur Laufzeit unbeabsichtigt überlagern. Selbstverständlich lassen sich auch mit verhaltensorientierten Architekturen durch gezielte Überlagerung von Stellgrößen oder durch Unterbrechung einer Stellgrößenübertragung hierarchische Verhaltensschichten erzeugen. „Verhaltensbasierte Steuerung" bedeutet eigentlich nur eine dezentral und unüberwacht ablaufende Steuerung [Lüth 1996].
Hybride Ansätze, die auf hierarchischen und zeitkritischen Ebenen verhaltensorientierte Strukturen besitzen, beruhen ebenfalls auf festen Strukturen.
In der Abb. 2.6 entsprechen die gestrichelten Linien dem Steuerfluß und die durchgehenden Linien dem Informationsfluß. Im linken Teil wird eine hierarchische Steuerungsarchitektur dargestellt. Typisch für diese Architektur ist, daß das System in verschiedene Abstraktionsebenen geteilt ist. Jede Steuerungsebene kann nur die jeweils tiefere beeinflussen. Wenn ein Plan auf der höchsten Ebene durchgeführt werden soll, muß erst die Information durch die unteren Ebenen „geholt" werden, und dann werden die Steuergrößen durch die Hierarchie sukzessiv bis zum Aktor übergeben.
In der rechten nebenläufigen Architektur laufen die Regelkreise nebeneinander. Der Steuerfluß verläuft einfach in der gleichen Richtung wie der Informationsfluß. Der kritische Punkt ist die Überlagerung von Stellgrößen zum Aktor. Diese Überlagerung ist problemabhängig und läßt sich nicht trivial durchführen. Stabilitätskriterien werden bei einer Überlagerung nicht mathematisch garantiert.

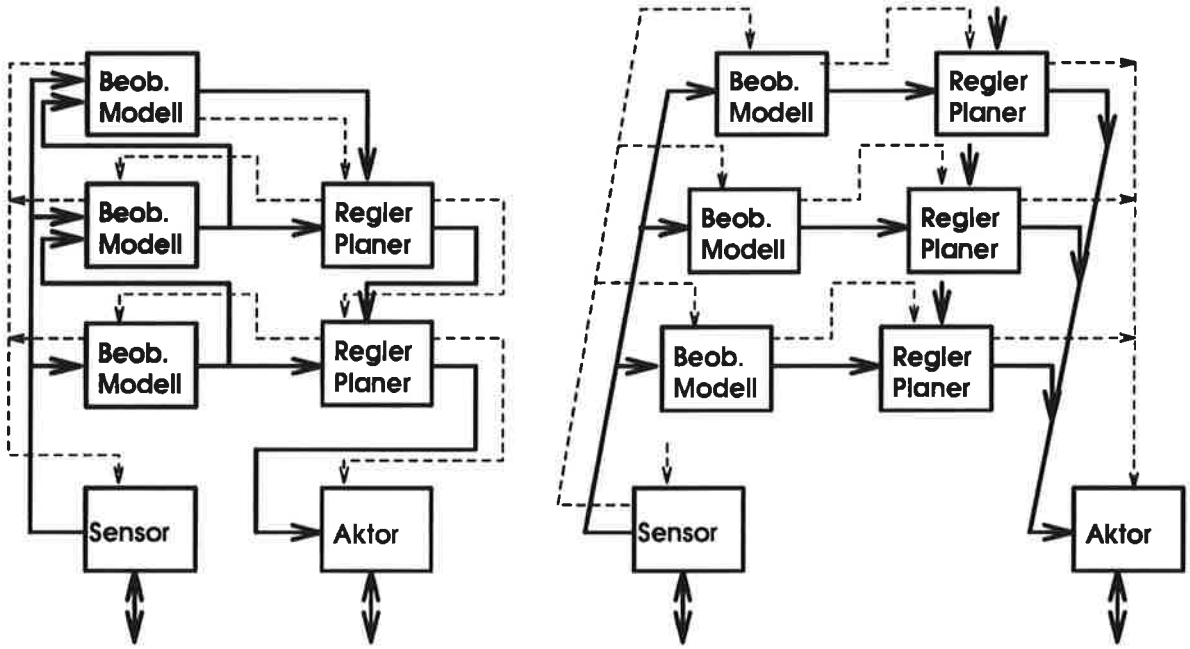

Abbildung 2.6: Hierarchische und verhaltensorientierte Steuerungsarchitektur.

Der Steuer- und Informationsfluß der beiden obigen Architekturen ist unterschiedlich. Bisher gibt es kein Echtzeitbetriebssystem, das beide obigen Arten von Steuerungsarchitekturen unterstützt und keines, das zwischen den beiden zur Laufzeit wechseln und/oder diese kombinieren kann.

Kapitel 3

Echtzeitbetriebssysteme

Echtzeitsysteme garantieren eine deterministische Bearbeitung von Prozessen, eine für Automatisierungssysteme absolute Notwendigkeit. Betriebssysteme wie UNIX, WINDOWS 95 und OS/2 erfüllen dieses Prinzip nicht. Die dramatische Steigerung der Leistungsfähigkeit von Prozessoren und Mikrocontrollern macht deren Einsatz in allen Automatisierungsbereichen attraktiv. Eine Voraussetzung für eine effiziente Programmentwicklung ist neben der eigentlichen Entwicklungsumgebung ein geeignetes Echtzeitbetriebssystem. Im Anhang A.1 findet sich tabellarisch eine Zusammenstellung von 16 Betriebssystemen [Wollert96].

3.1 Einsatz und Anwendungen von Echtzeitbetriebssystemen

Betriebssysteme umfassen sämtliche Programme, die notwendig sind, um die Betriebsmittel eines Rechners geeignet zu verwalten. Die Aufgaben eines Betriebssystems umfassen somit die Steuerung und Protokollierung des Programmablaufs, die Organisation und Koordination des Betriebsablaufs und die Schnittstelle zum Benutzer. Während normale Betriebssysteme wie z.B. UNIX die Prozessorauslastung in den Vordergrund stellen, liegt der Schwerpunkt eines Echtzeitbetriebssystems auf der schritthaltenden Bearbeitung von Prozessen. Unter Echtzeit-Betrieb werden sämtliche Prozesse innerhalb einer definierten Zeitschranke ausgeführt.
Die schritthaltende Betriebsart ist immer dann notwendig, wenn Systeme gesteuert und geregelt werden müssen oder auf Ereignisse reagiert werden muß. Die Echtzeitdatenverarbeitung umfaßt auch den Bereich der Prozeßankopplung eines Rechners über Datenschnittstellen mit der realen Prozeßwelt. Einige wichtige Stichwörter sind A/D- und D/A-Umsetzer, Digital-IO, Feldbusse oder DDC-Regelung. Früher war Echtzeitdatenverarbeitung nur Prozeßrechnern vorbehalten, die für Echtzeitdatenverarbeitung gebaut wurden, z.B. ABB Masters. Der Trend geht heute zu billigeren Rechnern. Besonders wichtige sind die VME-Bus-Rechner und der MultiBus-Rechner , aber auch der klassische PC wird mehr und mehr verbreitet. Der PC hat preiswerte Komponenten, leistungsfähige Entwicklungswerkzeuge und Standardsoftware.
Die Steuerungs- und Regelungsaufgaben werden immer größer und komplexer. Durch

die wachsende Komplexität nehmen Anforderungen und Anwendungen von Echtzeitbetriebssystemen stetig zu.

3.2 Kategorien von Echtzeitbetriebssystemen

Die Aufgaben von Echtzeitbetriebssystemen sind vielfältig, so auch deren Ansätze. Man kann die verschiedenen Echtzeitbetriebssysteme in hauptsächlich fünf Kategorien einteilen. Die Kategorien sind nicht streng aufgeteilt, sondern ein Echtzeitbetriebssystem kann mehreren Kategorien angehören oder sie unterstützen.

3.2.1 Echtzeit-UNIX

In der Gruppe Echtzeit-UNIX sind alle die Systeme zusammengefaßt, die im wesentlichen kompatibel zum UNIX-System V sind. Zur Realisierung der Echtzeitfähigkeit erscheint der Betriebssystemkern nicht als monolithischer Block, sondern als Micro-Kernel bzw. als unterbrechbarer Kern mit geeigneten Ausstiegspunkten. Echtzeitanforderungen für UNIX-Systeme werden in der Regel bei Prozeßleiteinrichtungen oder Prozeßsteuerungseinrichtungen gestellt.

3.2.2 Echtzeit-Kerne

Echtzeitkerne basieren häufig auf einem UNIX-kompatiblen Micro-Kernel, der die wesentlichen Betriebssystemdienste bereitstellt. Hierzu gehören die Speicherverwaltung, die Interruptverarbeitung, der Scheduler sowie die gesamte Taskverwaltung. Darüber hinaus bedienen Echtzeitkerne Schnittstellen auf der Basis des TCP/IP-Protokolls. Echtzeitkerne sind den speziellen Anforderungen zumeist optimal angepaßt und besitzen einen gut optimierten Code für die unterschiedlichsten Zielplattformen.

3.2.3 Echtzeit-Betriebssystem-Erweiterung

Betriebssystemerweiterungen werden zumeist für MS-DOS-Systeme eingesetzt. Eine Bibliothek, die zu den häufigen Entwicklungssystemen von Borland und Microsoft gebunden wird, ermöglicht die Einhaltung der Echtzeitbedingungen innerhalb einer ausführbaren Datei. Leider weisen sämtliche Echtzeiterweiterungen für MS-DOS wesentliche Nachteile auf: Probleme mit TSR-Programmen (Terminate Stay Ready), Reentryprobleme bei DOS-Aufrufen und Fließkommabibliotheken sowie eingeschränkter Speicherbereich.

3.2.4 Echtzeitbetriebssysteme

Reine Echtzeitbetriebssysteme sind in der Regel sehr schlanke und flexibel zu konfigurierende Systeme. Graphische Benutzeroberflächen und Netzwerkanbindungen sind auch Stand der Technik, aber häufig nicht im Entwicklungsstadium eines Projekts, wie es bei den Massen-Betriebssystemen üblich ist. Reine Echtzeitbetriebssysteme orientieren sich vom äußeren Erscheinungsbild her zumeist an UNIX.

3.2.5 Systeme für Embedded Systems

Bei Echtzeitlösungen für Embedded Controller handelt es sich in der Regel um Cross-Entwick-
lungssysteme mit Echtzeitbibliotheken für die Zielprozessoren. Entwicklungsumgebungen sind von einem 8-Bit-Prozessor 8051 bis hin zum 64-Bit-RISC-Prozessor verfügbar. Die Ausstattung und Leistungsfähigkeit dieser Systeme ist sehr unterschiedlich. Als Gemeinsamkeit der Systeme für Embedded Systems kann festgestellt werden, daß es sich bei den Entwicklungsumgebungen normalerweise um Cross-Entwicklungssysteme handelt. Es existiert ein Entwicklungsrechner (Host) und ein Zielsystem (Target). Einige Echtzeitbetriebssysteme, besonders Echtzeiterweiterungen für die gängigen Prozessorfamilien von Motorola (68x0) und Intel (80x86) lassen auch die Programmentwicklung auf dem Target zu.

3.3 Verteilung von Echtzeitsystemen

Echtzeitsysteme sind häufig Bestandteil einer vernetzten Rechnerstruktur. Normalerweise sind dabei drei Anwendungsgebiete zu beobachten.

3.3.1 Anschluß der Prozeßperipherie

Die wichtige Anbindung an den Prozeß kann dezentral über ein Netzwerk erfolgen. Ein wichtiger Punkt ist dabei die Anbindung intelligenter Geräte, die die Teile des Systems mit extremen Echtzeitbedingungen abdecken können. Üblicherweise werden sogenannte Feldbusse (Interbus-S, PROFIBUS, CAN-Bus) eingesetzt. Die Datenraten sind relativ gering, aber die Anforderungen an das Zeitverhalten sind demgegenüber hoch. Die Bedingungen der schritthaltenden Datenverarbeitung sind unbedingt einzuhalten. Diesen Anforderungen gehorchen die üblichen Feldbusse.

3.3.2 Aufteilung der Echtzeitaufgaben auf ein verteiltes Rechensystem

Im Sinne der Skalierbarkeit von Rechenanlagen werden zur Erfüllung einer Echtzeitaufgabe verschiedene Systeme gekoppelt. Eine Kopplung kann sowohl eng als auch lose realisiert werden. Bei einer engen Kopplung werden die einzelnen Rechner z.B. über einen gemeinsam genutzten Speicher verbunden. Bei lose gekoppelten Systemen werden Netzwerke eingesetzt. Beide Kopplungsarten sollten von einem Echtzeitbetriebssystem unterstützt werden.
Die Anforderung an die Datenrate und die Echtzeitfähigkeit sind gleichermaßen hoch. Die oftmals für diesen Zweck eingesetzten Verfahren (z.b. RPC, remote procedure calls) sind nur eingeschränkt echtzeitfähig. Problematisch sind in der Regel die eingesetzten Transportebenen. Ethernet ist beispielsweise nicht deterministisch. Die weite Ausbreitung von Ethernet (z.B. SINEC) führt trotzdem zu einem Einsatz im Bereich schritthaltender Datenverarbeitung. Erfolgreich ist dieses Vorgehen nur, wenn das

Netz klein ist und in der Auslegung des Gesamtsystems das Netzverhalten berücksichtigt wird.

3.3.3 Verbindung zu einem übergeordneten Bereich

Moderne Echtzeitanlagen sind oftmals Bestandteil eines unternehmensweiten Rechnerverbundes. In diesem Fall ist die Anbindung an ein Betriebsdatenerfassungsnetz oder ein Produktionsplanungssystem erforderlich. In derartigen Applikationen ist die Datenrate hoch und die Frage nach der Echtzeittauglichkeit von nachrangiger Bedeutung. In diesem Anwendungsfall liegt die eigentliche Domäne der kommerziellen Netzwerke (Netware, NFS, TCP/IP).

3.4 Stand der Technik der Echtzeitbetriebssysteme

Nachdem die Grundlagen der Problematik "Echtzeitbetriebssysteme" behandelt wurden, folgt nun eine nähere Betrachtung der heute auf dem Markt verfügbaren Echtzeitbetriebssysteme. 16 Echtzeitbetriebssysteme sind tabellarisch im Anhang beschrieben [Wollert, 1996].

3.4.1 Modularität und Kernelgröße

Aufgrund der vielfältigen Anforderungen, die an einen Rechner beim Einsatz in der Automatisierungstechnik gestellt werden, sind moderne Echtzeitbetriebssysteme in der Regel flexibel auf die jeweilige Anwendung hin zu kofigurieren. Dies hat für den Anwender den Vorteil, nur die für das System wichtigen Komponenten einzusetzen. Kritisch ist die Modularität für Rechner mit eingeschränktem Speicherangebot. Hier ist die benötigte Funktionalität mit der zur Verfügung stehenden Menge an Speicher zu realisieren. Neben dem Programmspeicher ist auch der für die Daten notwendige Speicher zu betrachten.

3.4.2 Programmierumgebung

Zunächst muß die gewünschte Programmiersprache für das jeweilige Echtzeitbetriebssystem verfügbar sein. Wichtig ist dabei, daß der gewünschte Sprachumfang (z.B. ANSI-C) unterstützt wird und die gewünschten Programmierwerkzeuge, wozu z.B. ein Quellcodedebugger gehört, bereitstehen. Aus dem komplexen Zusammenspiel zwischen Betriebssystem und Programmierumgebung ergeben sich eine Vielzahl von Fehlerquellen. Werden verbreitete Programmiersprachen eingesetzt, so kann von einer weitgehenden Fehlerfreiheit ausgegangen werden.
Bei einem Crossentwicklungssystem (Ziel- und Entwicklungsplattform sind nicht identisch) müssen beide Systeme miteinander kommunizieren. Bei umfangreichen Programmen stellen serielle Schnittstellen nicht immer die gewünschte Kommunikationsleistung zur Verfügung. Dieses führt zu hohen Turn-Around-Zeiten bei der Entwicklung der Applikationen. Beachtet werden muß auch, ob ein Quellcodedebugging im vollem Umfang, wenn überhaupt, unterstützt wird.

3.4.3 Leistungsziffern

Im Zusammenhang mit Echtzeitbetriebssystemen ist eine Reihe von Kennzahlen verbreitet. Eine davon betrifft die maximale Anzahl der Tasks (Prozesse). Dabei ist zu beachten, daß bei vielen Anwendungen die Anzahl der Tasks eher gering ist und die theoretischen Grenzen daher von wenig praktischer Relevanz sind. Die gleiche Aussage gilt auch für die Anzahl der Prioritäten.
Kennzeichen eines Echtzeitbetriebssystems sind schnelle Taskwechselzeiten und kurze Interruptlatenzzeiten. Bei den meisten Systemen sind diese Zahlen im Bereich von wenigen μs angesiedelt und weitestgehend von der Hardwareumgebung abhängig. Bei UNIX-Implementationen sollte beachtet werden, daß diese nicht in extrem zeitkritischen Anwendungen eingesetzt werden, da derartige Systeme in der Regel wesentlich höhere Reaktionszeiten aufweisen, als dies bei echten Echtzeitkernen der Fall ist.

3.4.4 Anpassung an spezielle Umgebungen

Liegt ein Rechensystem vor, welches nicht einem verbreiteten Standard entspricht, so muß das Echtzeitbetriebssystem angepaßt werden. Diese Notwendigkeit ergibt sich zumeist bei Embedded Systems. Grundvoraussetzung ist die Unterstützung der vorhandenden CPU. Für einen minimalen Kern sind zusätzlich zumindest eine Anpassung an den eingesetzten Interruptcontroller und Timerbaustein erforderlich. Werden Elemente zur Ein-/Ausgabe, wie z.B. Tastatur, Videoausgabe, Massenspeicher oder Netzwerke, eingesetzt, muß es möglich sein, geeignete Treiber zu erstellen und einzubinden. Ohne eine umfassende Unterstützung durch die Hersteller sind diese Anpassungen mit einem hohen Aufwand verbunden.
Soll das Programm zusammen mit dem Echtzeitsystem im ROM-Speicher des Rechners untergebracht werden, so ist auf die ROM-Fähigkeit zu achten. Die Anbindung an ein Dateisystem oder Netzwerk sollte vom Betriebssystem unterstützt werden. Eine eigene Erweiterung ist sehr aufwendig.
Bei Verwendung eines Netzwerkes ist zu prüfen, ob die geforderten Zeitschranken eigehalten werden können. Die erforderlichen Protokolle (z.B. PROFIBUS, TCP/IP) müssen unterstützt werden. Die üblichen Dateisysteme, wie z.B. DOS (FAT) oder UNIX, erfüllen in der Regel nicht die Anforderungen, die an ein Echtzeitdateisystem zu stellen sind. Dieser Umstand ist bei der Anwendung zu beachten oder es ist ein auch in diesem Bereich Echtzeit-geeignetes System zu benutzen.

3.4.5 Globale Eigenschaften

Bei allen betrachteten Systemen handelt es sich um prioritätsgesteuerte, preemptiv arbeitende. Es bleibt abzuwarten, ob sich Systeme mit komplexeren Zuteilungsalgorithmen wie das Deadlinescheduling in Zukunft durchsetzen werden. Die Unterstützung umfangreicher Mittel zur Taskkommunikation vereinfachen die Programmierung. Sollte ein Element fehlen, so läßt es sich zumeist mit Hilfe der vorhandenen nachbilden. Diese Aussage gilt jedoch nicht für die Prioritätenvererbung, die Bestandteil des Betriebssystem sein sollte, wenn sie benötigt wird. Wird ein Multiprozessorsystem eingesetzt,

macht es Sinn, ein Betriebssystem einzusetzen, das diese Eigenschaften unterstützt. Dabei ist zu prüfen, ob alle Anforderungen abgedeckt werden. Diese Aussage bezieht sich insbesondere auch auf das Zeitverhalten.
Ein wichtiges Element ist die Mensch-Maschine-Schnittstelle. Zum Teil werden vom Markt grafisch gestaltete, fensterorientierte Systeme verlangt. Die Auslegung und Programmierung ist mit einem erheblichen Aufwand verbunden, so daß eine umfangreiche Unterstützung sinnvoll ist. Die Unterstützung muß sich sowohl auf die verfügbaren Bibliotheken als auch auf die Entwurfs- und Gestaltungswerkzeuge beziehen.

3.5 TOROS, Beispiel eines Forschungs-EZBS

TOROS – Task Oriented Real-time Operating System ist ein Echtzeitbetriebssystem, das am Institut für Informatik in München entwickelt worden ist [Schrott 1994]. TOROS besitzt eine eigene Sprache, um die verschiedenen Teile des Systems zu beschreiben. Ein Steuerungs-/Regelungstask wird im TOROS-Steuerungssystem in verschiedene Module geteilt. Die Module sind als Zustandsmaschinen programmiert und benutzen geschützte Komandos. Die Module können auf mehrere Prozessoren verteilt werden. TOROS ist nicht für harte Echtzeitbedingungen geeignet. Die Module werden strikt zyklisch ausgeführt.

3.5.1 Die Module

Mit Hilfe der TOROS-Sprache kommunizieren Modul in einer festen Vorgehensweise durch Senden von sogenannten ,,task-calls", die Aktivitäten in anderen Modulen starten oder Nachrichten empfangen. Jedes Modul hat eine Menge von Zuständen, in dem jeder Zustand disjunkt mit den anderen Zuständen ist. Ein ganzes TOROS-L Programm beginnt mit der Deklaration der verfügbaren Rechner und den bestehenden Kommunikationsschnittstellen und wird von Modulen gefolgt. Die ganze Definition eines Moduls ist in folgende Teile getrennt.

- Deklaration des Zielrechners, auf dem das Modul ausgeführt werden wird.
- Deklaration und Initialisierung der Submodule, Funktionen, z.B. PID-Regler, Timer u.s.w..
- Deklaration der ,,task-calls", die in dem Modul erlaubt sind.
- Teil der Ausführung, ,,action part".

Beispiel:

```
hostdecl pc386 pc1;
hostdecl ti525 sps1;
conndecl pc1 com_1_9600 to sps1 com_9600

module robot host pc1;
```

```
...
endmodule

module chute host sps1;
...
endmodule;
```

3.5.2 Submodule

Auf der niedrigsten Steuerungs- und Regelungsebene befinden sich die Submodule. Die Submodule sind Funktionen, wie z.B. analoge und digitale Ein- und Ausgänge, Timer, PID-Regler. Sie werden in der `decls`-Phase initialisiert. Die Submodule werden als „Tasks" durch „task-calls" in den Modulen aufgerufen.

3.5.3 Zustände und „Actions"

Der „Action"-Teil des TOROS-Systems ist in Zustände unterteilt. Die „Actions" oder Zustände werden nach einer erfüllten Bedingung aktiviert. Nach jeder „Action" folgt eine Reihe von Submodulen und/oder „Task-calls" an andere Module. Die Zustände werden innerhalb des Moduls zyklisch abgeprüft.
Ein Modul kann wie folgt aussehen:

```
module chute host sps1,

decls
DigOut motor(Y2,on,off);
DigOut direction(Y3,up,down);
DigIn  bottom_key(X2,on,off);
Digin  top_key(X2,on,off);
Timer  delay;

tasks

empty do  newstate up;
bool ready return instate ready;

states

ready/shutdown:
once => motor.off;

up:
once => { direction.up; motor.on}
top_key.on =>newstate ready;
```

```
waiting:
once => { motor.off; delay(1000); }
delay.tout => newstate down;

down/initial:
once:  => { direction.down; motor.on }
bottom_key.on => newstate ready;

endmodule;
```

Kapitel 4

Konzept des Systems

In diesem Kapitel wird das Konzept des *CAIC*-Systemes präsentiert. Zuerst wird das Konzept allgemein eingeführt, danach werden die verschiedenen Hauptteile des Konzeptes beschrieben: Instanz, Strukturadaption, Steuerungsmodus, Kommandoformat, Puffer und Speicherverwaltung.
Außerdem werden zwei neue Begriffe eingeführt: Strukturadaption und der MULTI-Puffer.

4.1 *CAIC*. Eine Konzepteinführung

CAIC steht für *Cooperative Architecture Intelligent Control* und ist ein Regelungs- und Steuerungssystem, das besonders für mobile Roboter und andere komplexe und veränderliche Systeme, z.B. Fertigungssysteme mit kleinen Produktionsserien, geeignet ist. *CAIC* ist ein zeit-ereignis-orientiertes System. Jeder Regelkreis wird in diesem Konzept Verhalten genannt. Ein Verhalten wird in mehrere Instanzen zerlegt, die jeweils als Prozeß laufen. Die Instanzen sind allgemein kleiner als entsprechende Prozesse eines herkömmlichen Regelungs- und Steuerungssystems. In einem herkömmlichen Regelungs- und Steuerungssystem läuft der ganze Regelkreis sequentiell als Prozeß ab, während in *CAIC* der Regelkreis in mehrere Instanzen zerlegt wird und verteilt abläuft.

Vorteil: Bessere Parallelisierungs- und Distributionsmöglichkeiten.
Problem: Synchronisierung der Instanzen.

Die Aktivierung der verschiedenen Instanzen kann entweder zyklisch, ereignisbasiert oder sowohl zyklisch als auch ereignisbasiert erfolgen.
Das System ist strukturadaptiv, die Struktur paßt sich der aktuellen Aufgabe an und baut sich selbst auf, d.h. es startet nur die Teile, die benötigt werden. Die Systemstruktur verändert sich durch Strukturadaption während der Laufzeit.
Gemeinsamer Speicher, Shared Memory (SM), wird geschützt gepuffert. Normaler Puffer (FIFO) wird z.B. für Nachrichten benutzt. Der Name Puffer wird für die beiden obigen Puffertypen verwendet, die beide in *CAIC* gleich aufgebaut sind. Es gibt zwei verschiedene *CIAC*-Puffertypen, SINGLE- und MULTI-Puffer. Der SINGLE-Puffer ist ein normaler Shared Memory-Puffer, der auch als FIFO-Puffer benutzt werden kann.

Der MULTI-Puffer ist ein Puffer, der aus mehreren Shared Memory-Teilpuffern besteht. Der MULTI-Puffer kann als Redundanz-Puffer verwendet werden, wenn beispielsweise mehrere Meßwerte in einen MULTI-Puffer geschrieben werden, und der Mittelwert als Steuerungswert benutzt wird, oder als Überlagerungseingabe zu einem verhaltensorientierten Regelungskreis. Ressourcen ist der Sammelbegriff für Instanzen und Puffer.

4.2 Scheduling

Das *CAIC*-System verfügt über einen ungewöhnlichen Schedulingansatz. Die Grundidee ist, daß jede Instanz selbst entscheidet, wann sie aktiv sein soll. Um den Nachteil beim kooperativen Multitasking mit mangelnder Robustheit bei fehlerhaften Prozessen [Wollert 1996] zu vermeiden, wird das Scheduling in zwei Schritten durchgeführt. Ein Prozeß oder, wie im *CAIC*, eine Instanz besteht aus einem Kopf und einem Körper. Den Instanzköpfen eines Systems werden vom H-Scheduler (Scheduler der Köpfe) Zeitschranken zugeteilt. Die Köpfe entscheiden danach, ob ihre Körper Rechnerzeit bekommen sollen oder nicht.

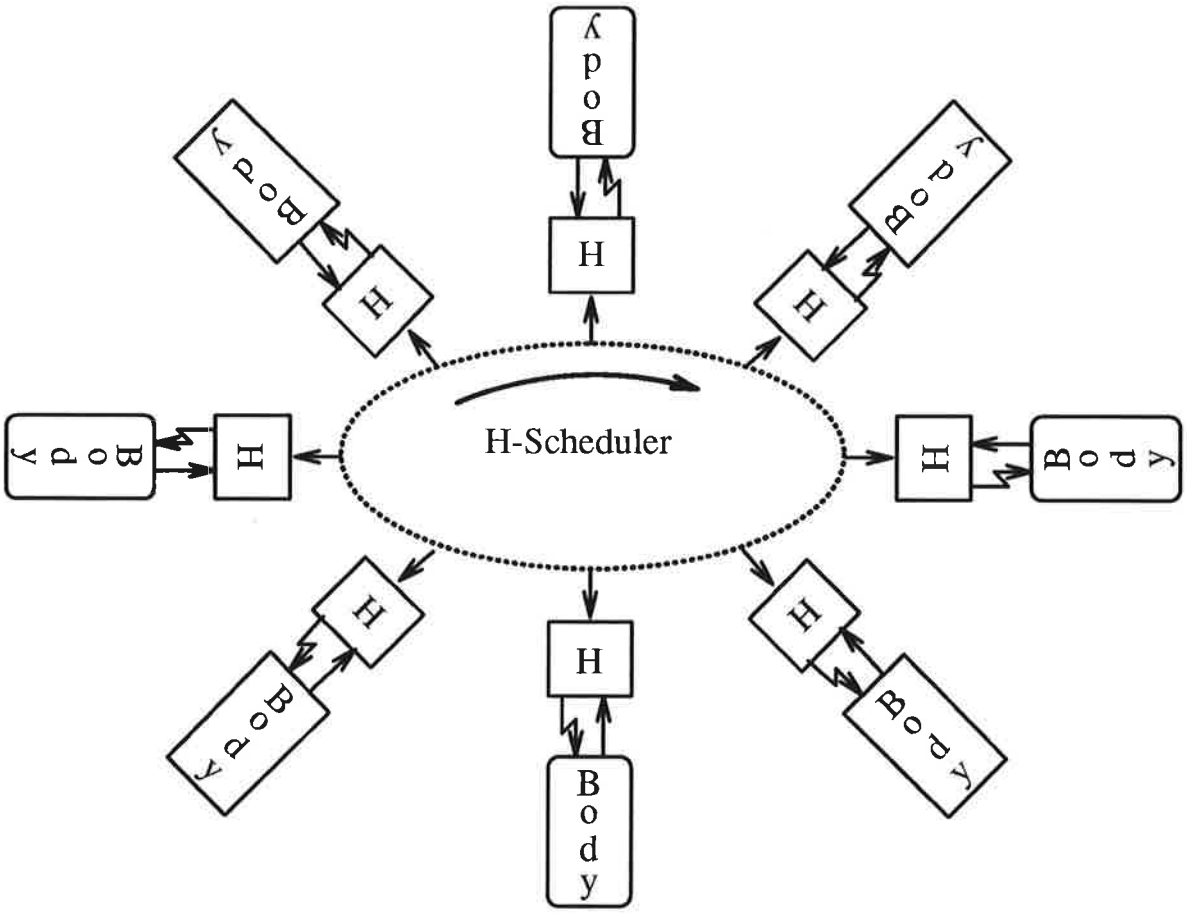

Abbildung 4.1: Das Verhältnis zwischen dem H-Scheduler und Head-Body-Aktivierung

Das Schedulingprinzip dieses semiverteilten Algorithmus ist in Abb. 4.1 dargestellt.

4.3 Unterteilung der Verhalten in Instanzen

Ein Verhalten besteht aus mehreren Teilen: aus Sensoren, Beobachtern, einem oder mehreren Regler, Planern usw.. Diese Teile können als Teilprozesse implementiert werden, die Instanzen genannt werden. Natürlich kann z.B. der Regler in manchen Fällen in weitere Teile zerlegt werden. Der Vorteil mehrerer Prozesse ist, daß man in einem größeren Bereich Standard-Instanzen benutzen kann. Standardteile gibt es in allen Regelungs- und Steuerungssystemen, aber diese sind normalerweise nicht als Prozesse implementiert, sondern als Funktionsblöcke innerhalb eines Prozesses. Ein weiterer Vorteil ist, daß der Regelkreis, je mehr er in Teile zerlegt ist, in desto höherer Ausdehnung parallelisiert werden kann. Ein Nachteil ist, daß die Synchronisierung und die richtige Reihenfolge der Instanzen im Gegensatz zu einem sequentiell bearbeiteten Vorgang nicht trivial zu erreichen ist.

4.4 Strukturadaptive Regelung und Steuerung

Strukturadaption darf nicht mit adaptiver Regelung verwechselt werden. Mit Strukturadaption wird die Eigenschaft bezeichnet, daß die Instanzen und ihre Verknüpfung zur Laufzeit variieren, während die adaptive Regelung die Reglerparameter optimiert. Der Unterschied ist also, daß sich Strukturadaption zwischen den Instanzen (oder Prozessen, Modulen, Tasks, abhängig von der Implemenation) abspielt, während die Adaption in der klassischen Regelung und Steuerung innerhalb der Instanz und oft innerhalb eines Funktionsblockes einer Instanz arbeitet. *CAIC* verändert seine Struktur während der Laufzeit.Die verschiedenen Teile, Instanzen und Puffer, werden automatisch und dynamisch im Speicher reserviert, aufgebaut und verändert.
Abb. 4.2 zeigt eine einfache Darstellung eines Systemaufbaus. Zunächst läuft nur die Root-Instanz, $t0$. In den nächsten Zeitpunkten $t1 - t4$ verändert sich die Struktur des Systems. Die Zeitpunkte sollen als Abtastzeitpunkte der Struktur des Systems betrachtet werden und nicht als Ereigniszeitpunkte.

4.4.1 Starten einer Instanz

In *CAIC* wird möglichst viel dezentral getätigt, so auch das Starten der Instanzen, das dynamisch erfolgen kann. Eine Instanz initialisiert sich selbst und reserviert Speicher für eventuelle Puffer, die die Instanz verwalten kann.
Eine Instanz kann entweder als Root-Instanz (am Anfang gestartet), von einer anderen Instanz oder interaktiv gestartet werden. In den beiden letzten Fällen wird die Instanz nicht explizit aufgerufen, sondern es wird lediglich eine Nachricht übergeben, daß die Instanz gebraucht wird. Wenn zwei Instanzen dieselbe Instanz brauchen, wird diese nur einmal beim ersten Bedarf gestartet. Es gibt keine Parent-/Childprozeßrelation zwischen Instanzen, sondern jede Instanz ist selbständig und unabhängig.

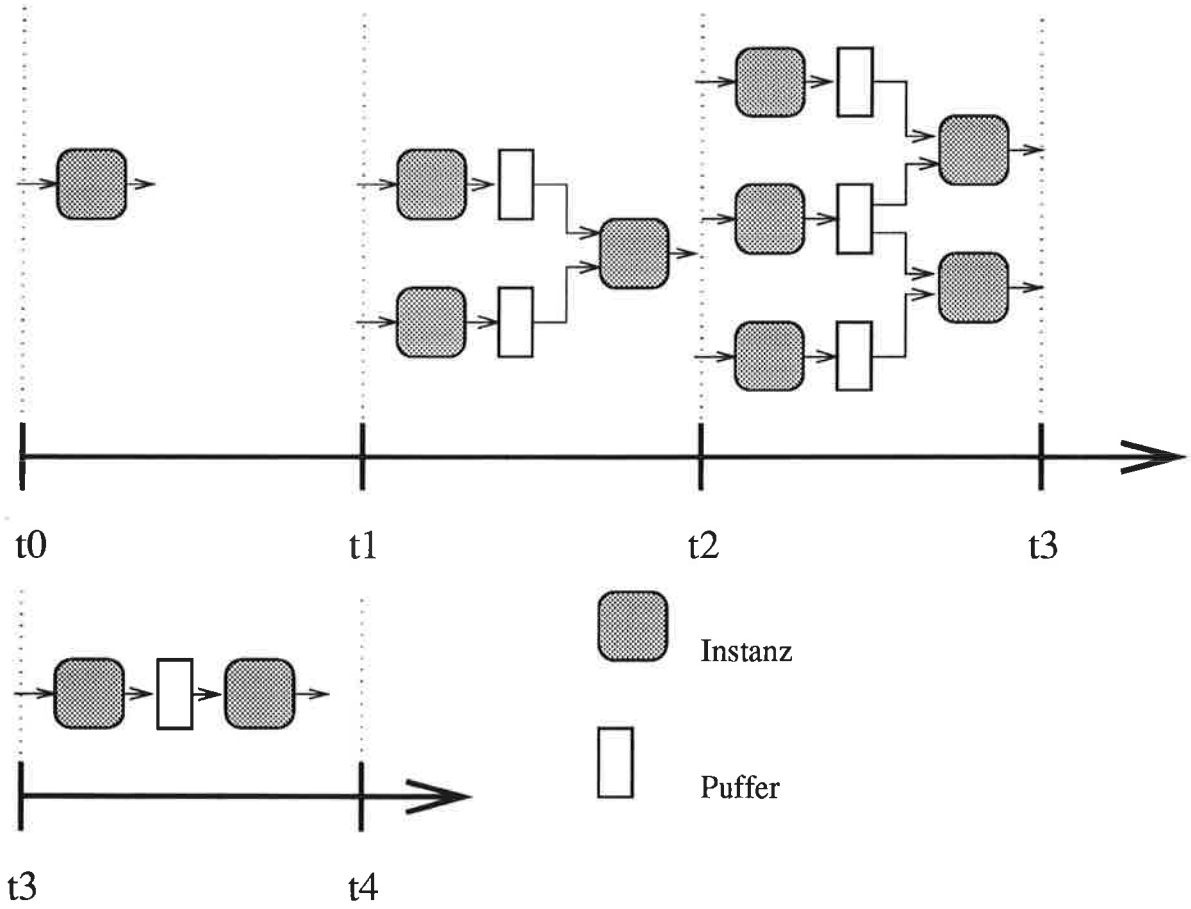

Abbildung 4.2: Veränderungen der Struktur eines Systems während der Zeit

4.4.2 Terminierung einer Instanz

Ein kritischer Punkt eines Systems ist die Terminierung von Instanzen. In *CAIC* darf eine Instanz nicht explizit terminiert werden, sondern sie entscheidet selbst, wann sie beendet werden soll. Dieses Konzept kann man folgendermaßen begründen:
Die Instanz weiß, welche Instanzen und Puffer sie braucht, um arbeiten zu können. Sie weiß aber nicht, ob andere Instanzen die gleichen Puffer und Instanzen brauchen, die sie selbst nutzt. Deswegen darf eine Instanz nicht andere Instanzen explizit terminieren.
Eine Instanz überprüft, ob ein Bedarf für sie vorliegt oder nicht. Wenn nicht, löscht sie eventuell verwaltete Puffer und terminiert sich selbst. Zuvor meldet sie den Instanzen und Puffern, die sie benutzt hat, daß sie ihre Dienste nicht mehr benötigt. Die Bedarfsüberprüfung wird nach jedem Durchlauf einer Instanz durchgeführt, und sorgt für eine schnelle Reaktion. Durch eine Kettenreaktion von „Brauche-dich-nicht-mehr"-Meldungen wird binnen kurzer Zeit die Struktur eines ganzen Systemes verändert.

4.5 Steuerungsmodus, Runmodus

In *CAIC* werden verschiedene Steuerungs- und Regelungsmöglichkeiten unterstützt. Eine Instanz kann entweder zyklisch, ereignisbasiert oder kombiniert zyklisch-ereignisbasiert aktiviert werden. Diese Möglichkeit gibt es auch in vielen anderen Echtzeitsystemen. Der Unterschied zwischen *CAIC* und anderen Systemen ist, daß man während des Zeitablaufes die Steuerungsmodi, entweder automatisch vom System oder interaktiv, variieren kann. Automatisch bedeutet, daß eine Instanz abhängig vom aktuellen Strukturzustand (siehe Abschnitt 4.7) ihren Steuerungsmodus und den anderer Instanzen zur Laufzeit ändern kann. Bei einem Wechsel der Steuerungsarchitektur, z.B hierachisch zu nebenläufigen Architekturen (Abschnitt 2.6.1), muß der Steuerfluß geändert werden. Eine Instanz kann in vier verschiedenen Steuerungsmodi oder Runmodi laufen:

1. Zyklisch: Die Instanz wird mit einer gewissen Taktzeit aktiviert, ohne ihre Umgebung zu berücksichtigen (Abb. 4.3). Dies ist das allgemeine Steuerungsverfahren in den meisten Regelungs- und Steuerungssystemen.

2. Ereignis: Die Instanz wird nach einem gewissen Ereignis aktiviert (Abb. 4.4).

3. Zyklisch, Ereignis senden: Wie im 1. Fall, aber nach der Aktivierung der Instanz wird ein Ereignis gesendet. Die Instanz triggert andere Instanzen.

4. Ereignis, Ereignis senden: Wie im 2. Fall, aber die Instanz triggert auch andere Instanzen.

4.5.1 Runmodus: Cyclus

Bei der klassischen Steuerung und Regelung hat jeder Regler eine Abtastperiode, innerhalb der alle Berechnungen abgeschlossen werden müssen. Dies wird auch in *CAIC* unterstützt.

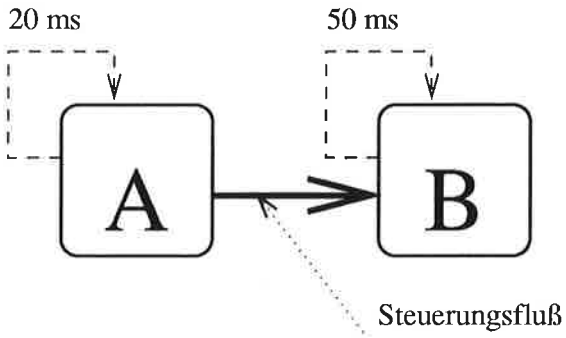

Abbildung 4.3: Steuerungsmodus: Cycle

Zyklisch gesteuerte Instanzen (Abbschnitt 4.3) sind beispielsweise sensorabtastende und sensorverarbeitende Instanzen. Jede Instanz, die zyklisch gesteuert ist, hat typischerweise eine Jitter-Erlaubnis oder Zyklustoleranz von etwa 5 bis 10 Prozent [Nielssen 1995]. Die Jitter-Erlaubnis hängt von der aktuellen Aufgabe ab. Ein Problem ist die Synchronisierung des Informationflusses, wenn mehrere Instanzen kooperieren, die mit der gleichen Taktzeit zyklisch ablaufen. Die Information muß in der richtigen Reihenfolge aktualisiert werden.

4.5.2 Runmodus: Event

Eine eventgesteuerte Instanz bleibt inaktiv, bis sie ein entsprechendes Ereignis (Event) oder eine Kombination von Ereignissen bekommt.

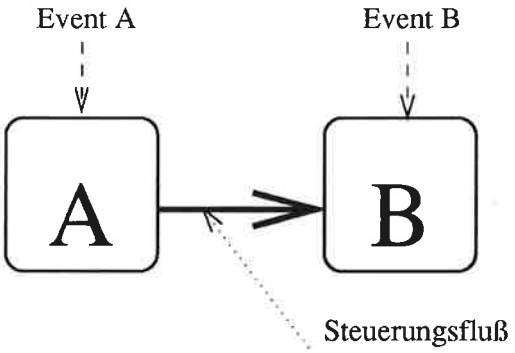

Abbildung 4.4: Runmodus: Event

In Abb. 4.4 werden die Instanzen A und B extern getriggert. Das Wort Event bedeutet in diesem System, daß die Instanz getriggert ist. Der Runmodus Event ist sehr praktisch, wenn man Instanzen synchronisieren möchte.

4.5.3 Runmodus: Cyclus-Trigg

Häufig kommt es vor, daß der Steuerungs-/Runmodus eine Mischung zwischen Cyclus und Event darstellt. Die Instanz im Cycle-Trigg-Modus läuft als eine Instanz mit zeitlicher Begrenzung, aber nach einem Durchlauf werden andere Instanzen getriggert.

Die Instanz generiert ein Ereignis und triggert eine oder mehrere Instanzen. Deswegen wird dieses Verfahren Trigg genannt.

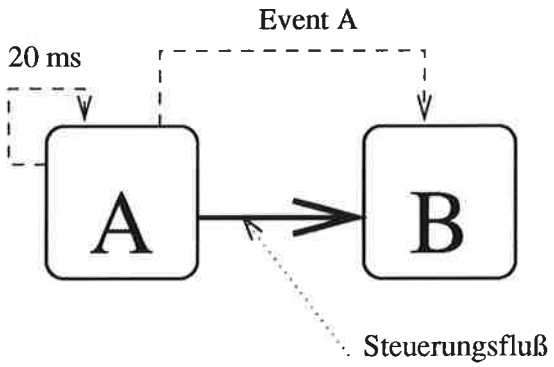

Abbildung 4.5: Runmodus: Cyclus-Trigg

In Abb. 4.5 läuft Instanz A im Cycle-Trigg-Modus und Instanz B im Event-Modus. A könnte beispielsweise eine abtastende oder meßwertverarbeitende Instanz sein, die nach einem Update andere, von den Meßwerten abhängige Instanzen, triggert.

4.5.4 Runmodus: Event-Trigg

Der Runmodus Event-Trigg erzeugt gewöhnlich eine Kette von ereignisgesteuerten Instanzen.

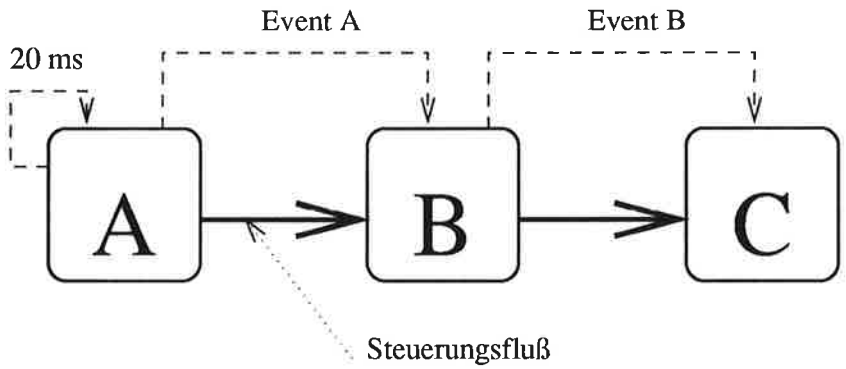

Abbildung 4.6: Runmodus: Event-Trigg

Abb. 4.6 zeigt eine Kette von Instanzen. Instanz A läuft im Cycle-Trigg-Runmodus und triggert Instanz B, die im Event-Trigg-Runmodus läuft. Diese Kette wird in einem herkömmlichen Regelungs- und Steuerungssystem typischerweise als Prozeß implementiert. Die synchronisierte Kette hat durch die erste Instanz eine "gemeinsame" Zeit. Der Runmodus Event-Trigg muß nicht in einer Regelungskette eingesetzt werden, sondern kann z.B. auch als getriggerte Planerinstanz, die nach einem Ereignis eine neue Instanz startet oder/und beobachtet, arbeiten.

4.6 Das *Control*format

Der Steuerfluß des Systems wird dadurch geändert, daß die Runmodi und die Parameter der aktuellen Instanzen geändert werden. Die Runmodi der Instanzen werden durch *Control*befehle geändert. Diese Befehle haben eine festgelegte Syntax:

1. Der Name der Instanz, deren Runmodi und/oder Parameter geändert werden sollen.
2. Der (neue) Runmodus.
3. Die Parameter des neuen Runmodus.
4. Ende
   (a) des Befehles.
   (b) des Kommandos. Das nächste Kommando des Befehls folgt entsprechend obiger Syntax.

Die Runmodi, deren Parameter und Trennzeichen sind in Abb. 4.1 aufgelistet:

`cyclus`	Parameter 1: Zykluszeit in [ms].
`event`	Parameter 1: Name der triggernden Instanz.
`cyclus_trig`	Parameter 1: Zykluszeit in [ms]. Parameter 2: Name der Instanz, die getriggert werden soll.
`event_trig`	Parameter 1: Name der triggernden Instanz. Parameter2: Name der Instanz, die getriggert werden soll.
,	Trennzeichen für Runmodus und Parameter
:	Trennzeichen zwischen zwei Kommandos.
@	Ende des Befehles.

Tabelle 4.1: Die Runmodi, Parameter und Trennzeichen des *Control*befehles

Nachfolgend sind zwei Beispiele eines *Control*befehles dargestellt.

```
instanceA,cyclus,50@
```

Die Instanz *instanceA* wird mit einer Zykluszeit von 50 ms zyklisch aktiviert.

```
instanceA,cycle_trig,75,instanceB:instanceB,event,instanceA@
```

1. Die Instanz *instanceA* wird mit einer Zykluszeit von 75 ms zyklisch aktiviert und triggert die Instanz *instanceB*.
2. Die Instanz *instanceB* wird von der Instanz *instanceA* getriggert.

Die *Control*befehle können entweder von den Instanzen oder interaktiv benutzt werden.

4.7 Strukturzustand STATE

Eine grundlegende Eigenschaft einer Instanz sind die Strukturzustände, kurz STATEs. Die Strukturzustände heißen so, um eine Verwechslung mit einem Modellzustand zu vermeiden. Strukturzustände mit Funktionsblöcken und ihren Verknüpfungen können wie in Abb. 4.7 illustriert werden.

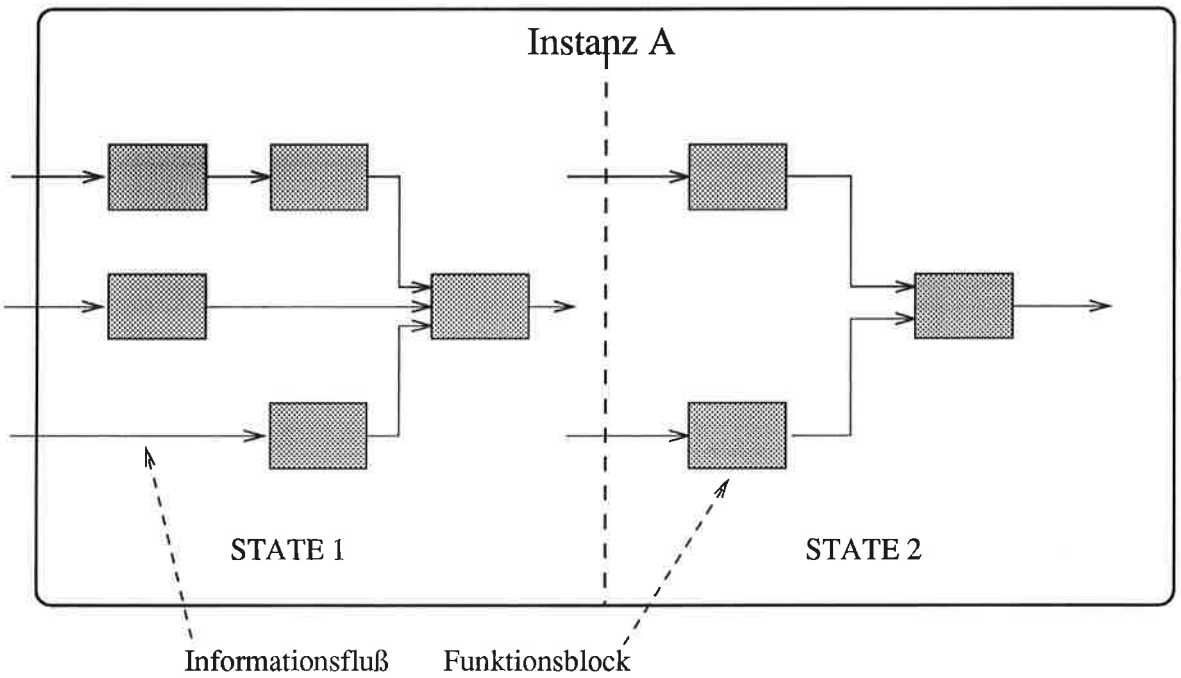

Abbildung 4.7: Strukturzustände einer Instanz

Ein Strukturzustandswechsel bedeutet, daß man beispielsweise zu einer anderen Regelungsstruktur innerhalb der Instanz wechselt oder einen alternativen Steuerungsalgorithmus benutzt. In einem herkömmlichen Steuerungs- und Regelungssystem (Abb. 4.8) sind die Strukturzustände mehrere Instanzen (Prozesse, Tasks, Module). Die Strukturzustände können dabei auch in mehrere Instanzen zerlegt werden.
Durch die Strukturzustände kann eine Instanz, abhängig von der Aktualisierungsfrequenz ihrer Eingaben, einen stabilen Regleralgorithmus wählen. Die Zustände können auch verschiedenen Aktionsplänen der Instanz entsprechen.

4.8 Die Puffer

Es gibt zwei verschiedene *CAIC*-Puffer. Der erste ist ein klassischer Puffer in der Hinsicht, daß ihm lediglich ein Speicherbereich zu Verfügung steht . Die verschiedenen Instanzen lesen bzw. schreiben aus einem bzw. in einen reservierten Speicherbereich. Diese Art von Puffer wird in *CAIC* SINGLE-Puffer genannt. Ein SINGLE-Puffer kann ein Shared-Memory oder ein FIFO-Puffer sein.
Die zweite Puffervariante ist der MULTI-Puffer. Ein MULTI-Puffer ist ein Puffer, dem mehrere reservierte Speicherbereiche zur Verfügung stehen. Die schreibenden Instanzen

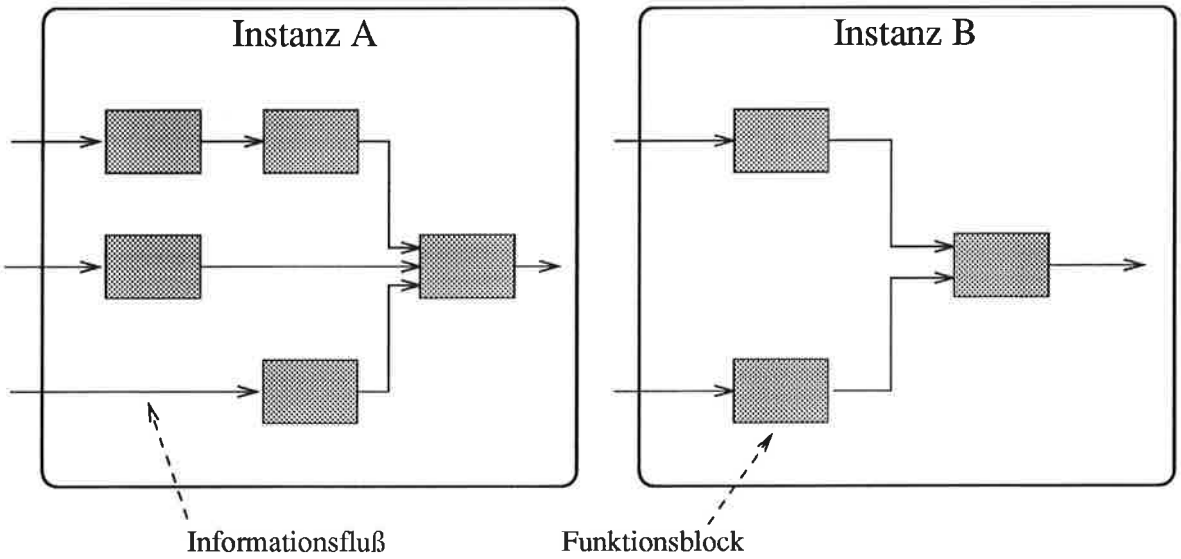

Abbildung 4.8: Strukturzustand in einem allgemeinen Regelungs- und Steuerungssystem

haben eigene Teilpuffer. Die lesenden Puffer können eine ganze Liste von Teilpuffern lesen und anschließend die Pufferwerte bearbeiten. Der MULTI-Puffer ist ein Versuch, Informationen, die z.B. überlagert werden sollen, systematisch und dynamisch zu behandeln. Die Art, wie die Informationen in einem MULTI-Puffer überlagert werden, ist problemabhängig.

4.8.1 Puffermodi

Es gibt, wie bereits erwähnt, zwei verschiedene Puffermodi, doch alle Puffer oder Teilpuffer sind einheitlich analog den Instanzen aufgebaut. Jeder Puffer besteht aus zwei Teilen: Einem Teil mit allgemeiner Information (Aktualisierungszeit, Erzeuger usw.) und dem Datenteil mit der pufferspezifischen Information.

4.8.1.1 SINGLE-Puffer

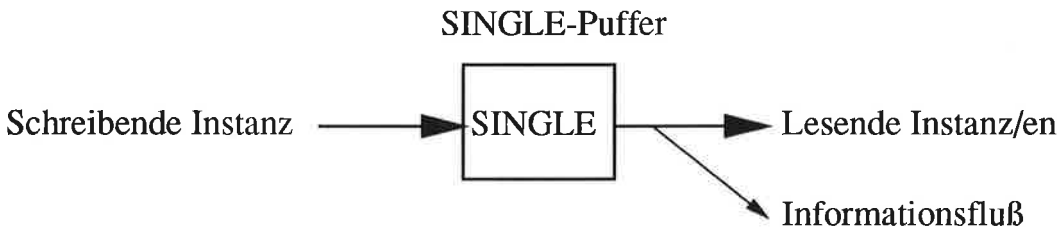

Abbildung 4.9: SINGLE-Puffer

Der erste Modus ist SINGLE und entspricht einem klassischen Puffer mit einem Speicherbereich. Die SINGLE-Puffer können als Monitore betrachtet werden, in denen nur

eine Instanz gleichzeitig Operationen auf dem Puffer ausführen darf. Der SINGLE-Puffer wird entweder als Shared-Memory zwischen Instanzen oder als FIFO-Puffer bei Schnittstellen verwendet.

4.8.1.2 MULTI-Puffer

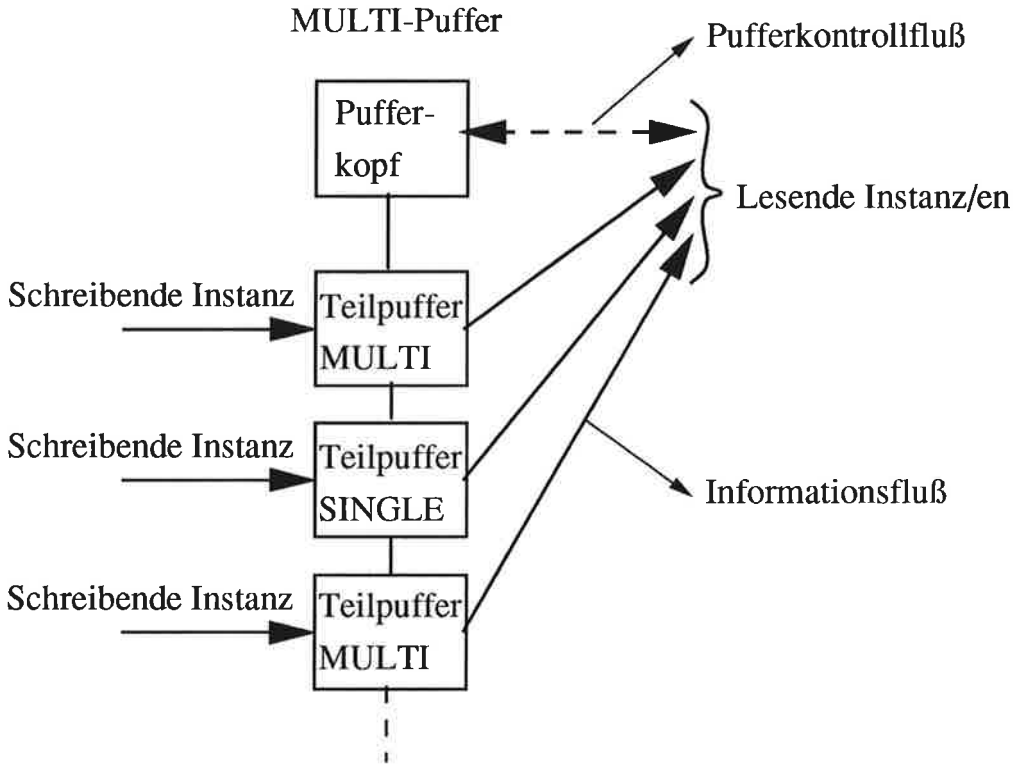

Abbildung 4.10: MULTI-Puffer

Der MULTI-Puffer besteht aus einem Kopf und einer Liste von Teilpuffern. Jede schreibende Instanz hat ihren eigenen Teilpuffer. Der Erzeuger des Puffers, MAKER, verwaltet den Pufferkopf. Andere Instanzen haben Zugriffsrecht zum Pufferkopf und den Teilpuffern, dürfen aber im Kopf nichts verändern. Der Pufferkopf hat den Puffermodus MULTI.
Die Teilpuffer wiederum sind entweder vom Modus SINGLE oder MULTI. In diesem Fall haben die Begriffe SINGLE und MULTI andere Bedeutung:

- Ein Teilpuffer mit Modus MULTI darf mit anderen MULTI-Teilpuffern interferenzieren.
- Ein Teilpuffer mit Modus SINGLE darf nicht mit anderen Teilpuffern inteferenzieren.

Welche Priorität ein MULTI- gegenüber einem SINGLE-Puffer hat bzw. wie die verschiedenen Typen von Teilpuffern behandelt werden sollen, ist im System nicht von vorneherein festgelegt, sondern hängt von der lesenden Instanz bzw. den lesenden Instanzen ab.

4.8.2 Interaktion zwischen Puffern und Instanzen

Abb. 4.11 zeigt ein Beispiel einer Interaktion zwischen verschiedenen Puffern und Instanzen.

1. Puffer A ist ein SINGLE-Puffer mit einer schreibenden und zwei lesenden Instanzen. MAKER bedeutet, daß Instanz 1 der Verwalter des Puffers ist.

2. Puffer B ist ein typischer MULTI-Puffer. Instanz 3 ist hier der Leser und Verwalter des Puffers. Instanz 3 verwaltet den Kopf des Puffers und geht vom Kopf aus, wenn sie die Teilpuffer liest (deswegen zeigt der Pfeil in die „falsche" Richtung). Instanz 2 und Instanz 5 haben ihre eigenen Teilpuffer, in die sie unabhängig voneinander schreiben. Beide Instanzen schreiben im MULTI-Modus, was bedeutet, daß die Informationen, die geschrieben werden, überlagert werden dürfen.

3. In Puffer C schreibt eine der Instanzen im SINGLE-Modus. Ein Teilpuffer, der im SINGLE-Modus geschrieben wird, darf nie mit Informationen aus anderen Teilpuffern gemischt werden. Instanz 5 entscheidet, wie der SINGLE-Teilpuffer behandelt werden soll. Beispielsweise kann ein SINGLE-Puffer automatisch höhere Priorität als ein MULTI-Teilpuffer haben.

4.9 Der Aufbau einer allgemeinen Instanz

Alle Instanzen sind, unabhängig von ihrer Aufgabe, gleich aufgebaut. Das ist einer der Vorteile des *CAIC*-Systems. Die Schnittstelle des Programmierers ist immer gleich (Abb. 4.12).

4.9.1 Der Instanzinformationsblock

Jede Instanz hat alle ihre wichtigen Informationen in einem Instanzinformationsblock. Dieser Block enthält Information über:

- Name und Identität
- Zykluszeit, Zyklustoleranz
- Runmodus
- Von wem wird die Instanz getriggert und wen triggert sie ?
- Initialisierungskontrollvariablen
- Von wie vielen Instanzen wird die Instanz benutzt ?
- Welche Instanzen und Puffer benutzt die Instanz (Ein- und Ausgabepufferlisten)?
- Strukturzustand

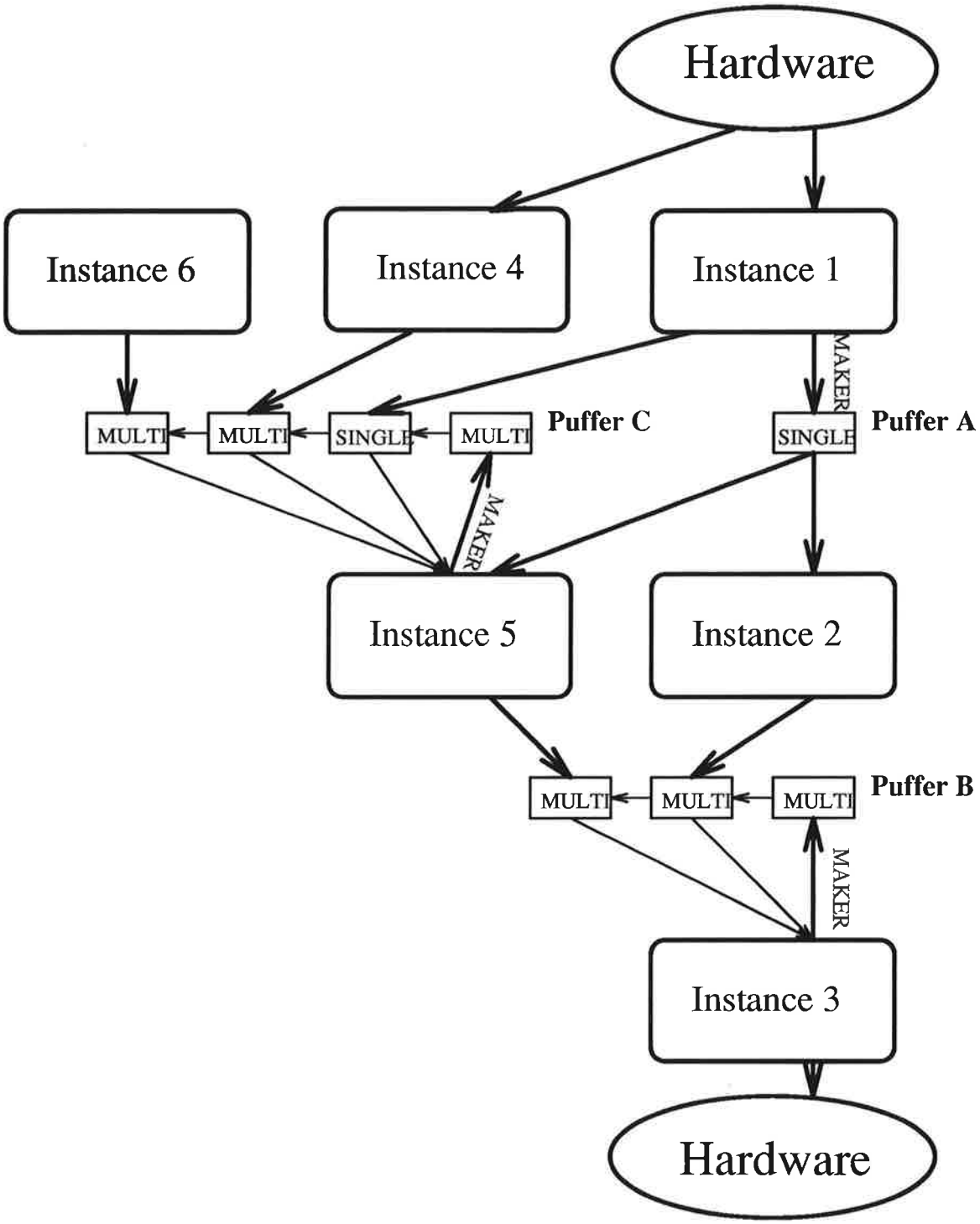

Abbildung 4.11: Interaktion zwischen Puffern und Instanzen

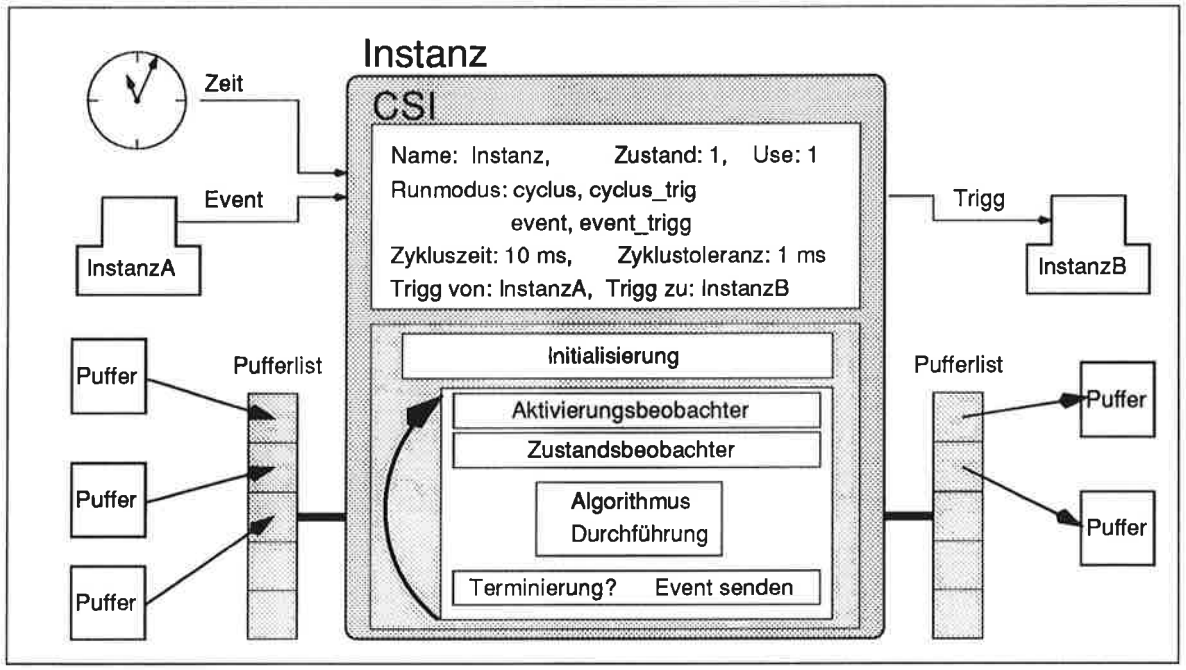

Abbildung 4.12: Die Instanz

Eine Instanz hat immer Zugriff auf ihren eigenen Instanzinformationsblock, *CSI* (*Control Structure Information*), und unter gewissen Bedingungen auch Zugriff auf den anderer Instanzen. Wenn eine Instanz z.B. den Steuerfluß anderer Instanzen durch einen *Control*-Befehl verändern will, dann hat sie Zugriff und Schreibrecht auf die betreffenden Instanzen.

4.9.2 Initialisierung der Instanz

Die Initialisierung wird von der Instanz selbst vorgenommen. In der Initialisierungsphase haben andere Instanzen keinen Zugriff auf die Instanz. Die Instanz muß nach der Initialisierung immer lauffähig sein.

1. Initialisierung.
   Hier werden alle *CSI*-Variablen gesetzt und einige grundlegende Funktionen der Instanz werden gestartet.
   - Der Name der Instanz wird global definiert.
   - Alle Default-Werte des Instanzinformationsblocks werden initialisiert.
   - Die Ein- und Ausgabelisten werden initialisiert.
2. Speicherreservierung des Puffers.
   Speicher für alle Puffer, die von der Instanz verwaltet werden, wird im Speicher reserviert und initialisiert.
3. Instanzüberprüfung.
   Die Instanz kontrolliert, welche Instanzen anfangs gebraucht werden, um lauffähig zu sein. Diejenigen Instanzen, die noch nicht gestartet sind, werden

automatisch gestartet. Um einen bestimmten Puffer benutzen zu können, muß die verwaltende Instanz aufgerufen werden.

4. Verbindung zu den Ein- und Ausgabepuffern.
   - Die Informationen der verschiedenen Puffer werden in den Ein- und Ausgabepufferlisten gespeichert.
5. Nach diesem Schritt ist die Instanz einsatzbereit.

4.9.3 Die Instanz während der Laufzeit

Nach der Initialisierung läuft die Instanz als Prozeß. Jeder Durchlauf der Instanz wird in drei Sektionen aufgeteilt: PRE STATE, STATE und POST STATE.

PRE STATE besteht aus drei Teilen:

1. Dem Aktivierungsbeobachter. Er entscheidet, ob die Instanz aktiv sein soll oder nicht.
2. Dem Lesen des Eingabepuffers zur Aktualisierung der Eingabewerte.
3. Dem Zustandsbeobachter, der aus den aktualisierten Eingabewerten den neuen Zustand berechnet.

STATE Durchführung des aktuellen Strukturzustands.

POST STATE besteht aus drei Teilen:

1. Dem Schreiben des Ausgabepuffers zur Aktualisierung der Ausgabewerte.
2. Der Terminierung, wenn die Instanz nicht mehr gebraucht wird.
3. Dem Senden eines Ereignisses, wenn die Instanz im Runmodus Cyklus-Trigg oder Event-Trigg läuft.

4.10 Dynamische Speicherreservierung

In *CAIC* werden zahlreiche Speicherreservierungen und -freigaben in Echtzeit durchgeführt. Das Problem bei Echtzeit und dynamischer Speicherreservierung ist, daß nach einer Weile der Speicher segmentiert ist. Der freie Speicher besteht nicht mehr aus einem homogenen Block, sondern liegt zersplittert im ganzen Speicherbereich vor (Abb 4.13). Dann kann der Fall auftreten, daß der größte zusammenhängende Speicherbereich kleiner ist als die Speichermenge, die man reservieren will, obwohl es insgesamt genügend freien Speicher gibt. Ist das System unfähig, angeforderten Speicherbereich in kleineren Blöcken zu verwalten, kann dies zu Problemen führen. Auch wenn das System diese Technik beherrscht, wirkt sich segmentierter Speicher ungünstig aus. Die Leistung des System wird geringer (Speicherreservierungen und -freigaben dauern länger), je mehr der Speicher segmentiert ist. In einem Echtzeitsystem ist dies besonders ungünstig, weil auch die Laufzeit der Instanzen sich abhängig vom Grad der Segmentierung ändert. In *CAIC* gibt es derzeit keinen echtzeitfähigen Desegmentierer.

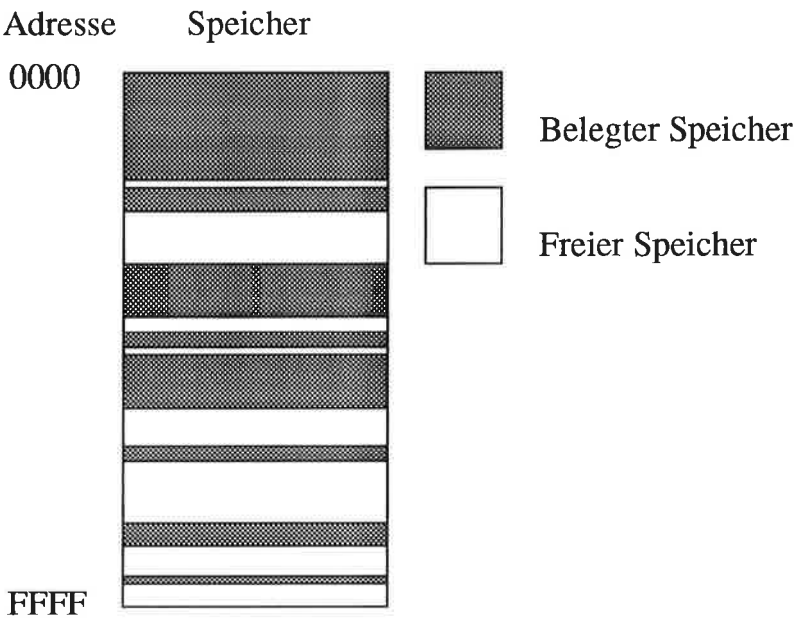

Abbildung 4.13: Segmentierter Speicher

In *CAIC* wird das Problem teilweise unterdrückt in dem Sinn, daß möglichst oft gleich große Speicherblöcke reserviert werden. Ein Ansatz für einen echtzeitfähigen "Garbage Collector" (der Desegmentierungsteil) wurde von [Magnusson 1995] vorgeschlagen.

Kapitel 5

Die Zielplattform, der Kheperaroboter

Als Zielplattform für *CAIC* wurde der Khepera-Roboter ausgewählt. Anstatt einer Simulationsumgebung wurde ein realer mobiler Roboter gewählt, weil die Interaktion eines autonomen mobilen Roboters mit seiner Umgebung von schwer erfaßbaren Umständen abhängt, die sich in einer Simulationsumgebung schwer modellieren läßt. Auf der anderen Seite sind Tests mit realen Robotersystemen aufwendig und kostenintensiv. Der Khepera-Miniaturroboter bietet eine relative billige Zielplattform an, die den letzten Einwand entkräftet. Der Khepera-Roboter wurde am SFIT (Swiss Federal Institute of Technology) in Laussane, Schweiz, entwickelt.

5.1 Die Hardwarekonfiguration

Ein Khepera, wie er in Abbildung 5.1 dargestellt ist, ist folgendermaßen ausgestattet:

- Motorola 68331 Mikrocontroller mit 16MHz, 32-Bit Datenbus
- 256KByte RAM
- 100g Gewicht mit Greifer
- Serielle Schnittstelle zum Anschluß an SUN-Workstation oder PC
  - Steuerung des Roboters durch Workstation oder PC
  - Laden eines Steuerungsprogrammes in den Speicher des Roboters
  - Übermittlung von Befehlen und Informationen

5.1.1 Das Antriebsmodul

Der Khepera ist modular aufgebaut und besteht aus aufeinandergesteckten Platinen, in der Grundversion aus dem Antriebsmodul und dem Prozessormodul.

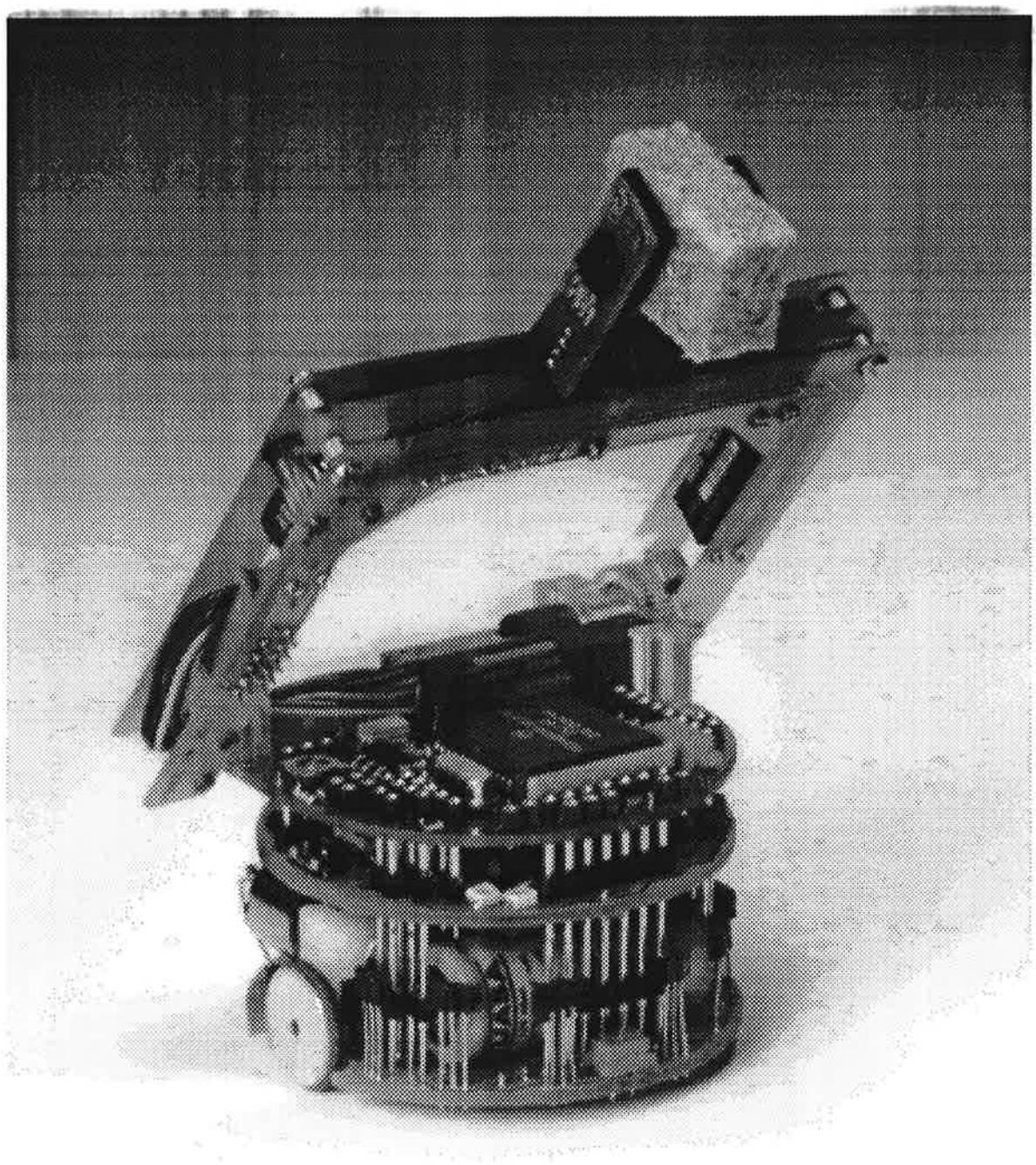

Abbildung 5.1: Der Khepera-Roboter mit Greifermodul [K-Team]

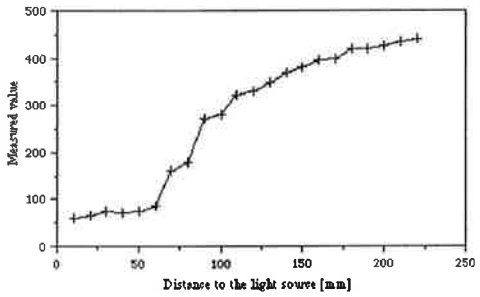

Abbildung 5.2: Der Abstand eines Objektes als Funktion des reflektierten Lichtes [K-Team 1994]

Das Antriebsmodul, das in Abbildung 5.4 dargestellt ist, enthält neben dem Antrieb auch Infrarotsensoren. Es handelt sich dabei um Reflexlichtschranken, einer Kombination aus Leucht- und Photodiode. Mit ihnen kann zum einen die Intensität des ambienten Lichts gemessen (bei abgeschalteter LED) und zum anderen die Entfernung zu einem Hindernis (bei eingeschalteter LED) durch Messung der Intensität des reflektierten Lichts von der LED ermittelt werden, wie in Abb. 5.2 dargestellt. Die Sensoren decken die ganze Umgebung des Roboters ab, Abb. 5.3. Außerdem befinden

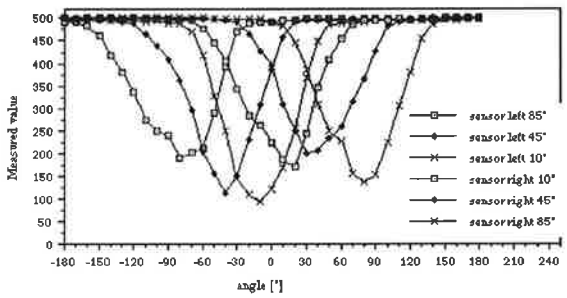

Abbildung 5.3: Anregung der IR-Sensoren unter verschiedenen Winkeln [K-Team 1994]

sich NiCd-Akkumulatoren zum autonomen Betrieb auf der Antriebseinheit.

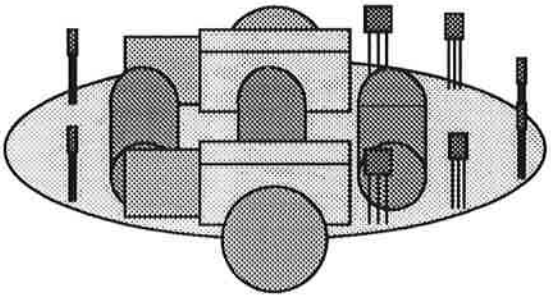

Abbildung 5.4: Antriebsmodul des Khepera [Heinzmann 1995]

5.1.2 Das Prozessormodul

Das Prozessormodul, schematisch in Abbildung 5.5 dargestellt, enthält einen Mikrocontroller vom Typ 68331 von Motorola, 256 KByte RAM und ein 256 oder 512 KByte PROM. Daneben befindet sich noch der Anschluß für die serielle Schnittstelle und ein Jumperblock. Mit den Jumpern kann der Betriebsmodus des Roboters und die Übertragungsrate der seriellen Schnittstelle eingestellt werden. Die verschiedenen Betriebsmodi sind in [K-Team, 1994] ausführlich beschrieben. Rechts vorn auf der Platine sind zwei SMD-LEDs angebracht, die vom Benutzer angesteuert werden können. Sie bieten so die Möglichkeit, unkompliziert Zustände des Steuerungssystems oder Zykluszeiten von Schleifen sichtbar zu machen.

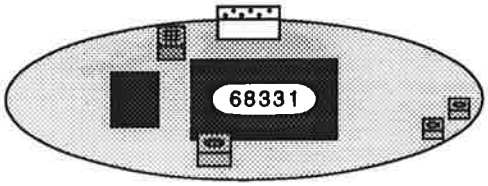

Abbildung 5.5: Prozessormodul des Khepera [Heinzmann 1995]

5.1.3 Das Greifermodul

In der hier vorgestellten Arbeit wurde noch ein Greifermodul verwendet. Das Greifermodul (siehe Abbildung 5.6) wird auf dem Prozessormodul montiert. Es enthält im Gegensatz zum Antriebsmodul einen eigenen Prozessor vom Typ 6811, der für die Motorsteuerung des Arm- und des Greiferantriebes verantwortlich ist. Das Greifermodul enthält verschiedene Sensoren, um den Zustand des Greifermoduls zu erfassen. Es können sowohl die Stellung des Armes als auch des Greifers bestimmt werden. Der Greifer selbst enthält eine Lichtschranke, um die Anwesenheit eines Objektes im Greifer zu erfassen, und Kontakte, über die der elektrische Widerstand eines gegriffenen Objektes gemessen werden kann. Alle Messwerte werden vom Prozessor ständig aktualisiert und zur Verfügung gestellt.

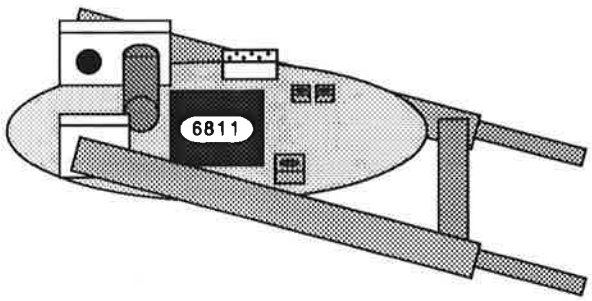

Abbildung 5.6: Greifermodul des Khepera [Heinzmann 1995]

Während einfache Module wie das Antriebsmodul direkt über den Prozessorbus des 68331 in das System integriert werden, sind intelligente Module wie das Greifermodul

mit einem eigenen Prozessor über ein Netzwerk an den Hauptprozessor angeschlossen. Das BIOS des Kheperas enthält spezielle Funktionen, um Nachrichten über dieses Netzwerk zu versenden. Beide Kommunikationsschienen laufen über Steckkontakte, mit denen die Module untereinander verbunden sind. Dadurch ist der Zugriff auf jeder Modulebene auf beide Schienen möglich, und es können immer sowohl einfache wie auch intelligente Module auf eine bestehende Modulkonfiguration aufgesteckt werden. Sowohl die Modulfunktionen als auch die Verbindungen untereinander sind in Abb. 5.7 dargestellt.

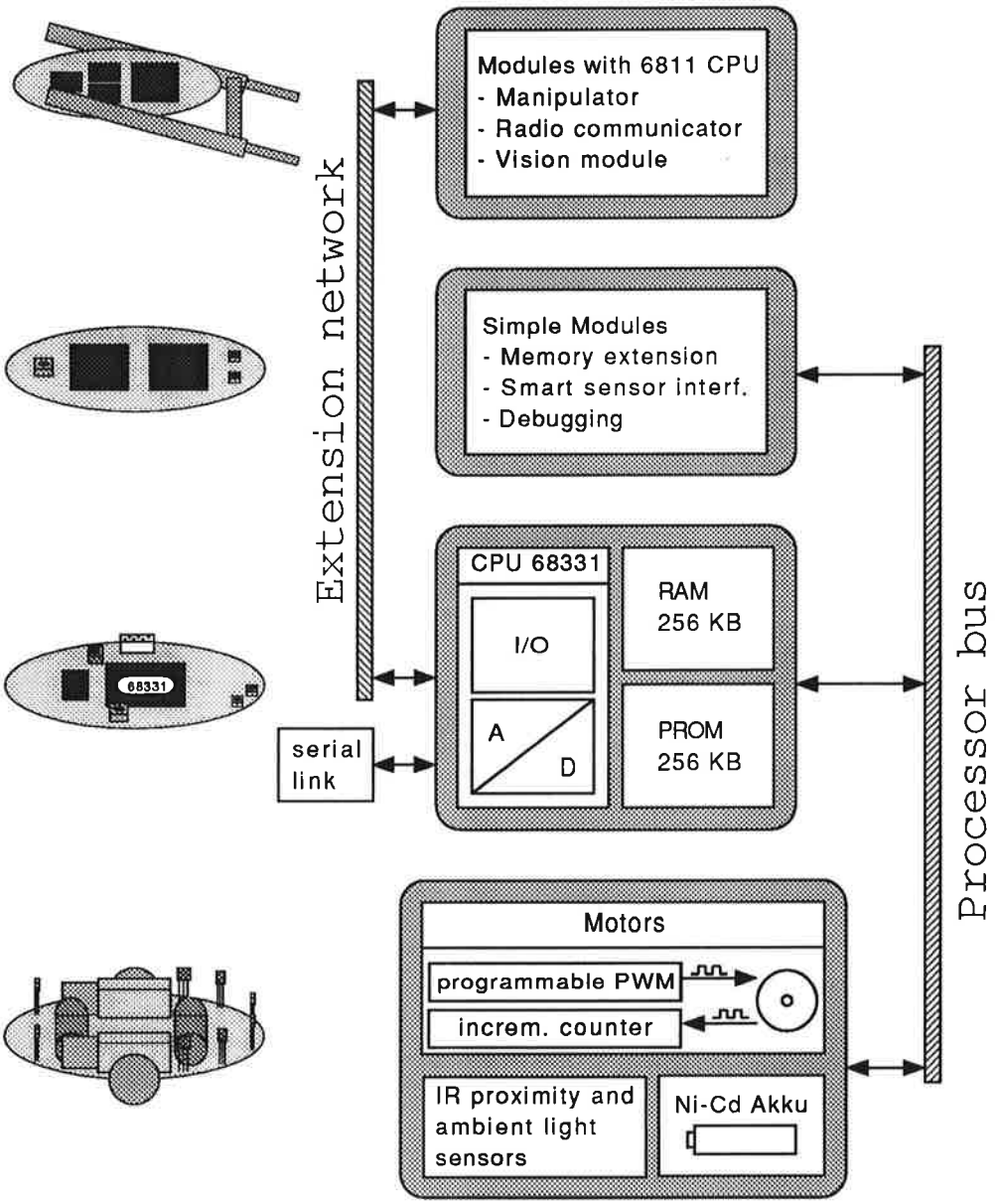

Abbildung 5.7: Hardwarearchitektur der Khepera-Roboter [Heinzmann 1995]

5.1.4 Das I/O-Modul

Das I/O Modul („General I/O-Turret") ist ein Erweiterungsmodul, das dem Anwender den Aufbau und die Integration eigener Hardwareerweiterungen erlaubt, beispielsweise das IRC-Modul unten. Das I/O-Modul steht unter der Kontrolle des Hauptprozessor. Es bietet im wesentlichen mehrere digitale Ein- und Ausgänge, analoge Eingänge und eine vollständige H-Brücke zur Motorsteuerung.
Ein Großteil der Platinenfläche nimmt eine freie Verbindungsfläche ein, auf der der Benutzer eigene Aufbauten verwirklichen kann.

5.1.5 Das IRC-Modul

Das infrarotbasierte Inter-Roboter-Kommunikationsmodul ist eine Eigenentwicklung des IPR [Grasman 1996]. Es stellt auf einer Steckplatine vier IrDA-kompatible Infrarot-Tranceiver zur Verfügung, deren Sende- und Empfangsbereich den gesamten Raum abdeckt.
Das IR-Modul wurde entwickelt, um zu zeigen, daß die drahtlose lokale Inter-Roboter-Kommunikation ein geeignetes Mittel ist, um eine wirkungsvolle Steuerungskopplung für die kooperative Lösung von Aufgaben durch mehrere Roboter zu erreichen [Grasman 1996].

5.2 Das Betriebssystem des Khepera

Das Steuerprogramm wird mit einem Crosscompiler übersetzt, damit es direkt auf dem 68331 des Prozessormoduls lauffähig ist, und über die serielle Schnittstelle in den Speicher des Khepera geladen oder aber in ein PROM gebrennt und gegen das alte Khepera-ROM auf der Unterseite des Prozessormoduls ausgetauscht.

5.2.1 Laden von Programmen in den Khepera

Das Steuerungsprogramm kann direkt auf dem Prozessor des Khepera ausgeführt werden, muß aber erst mit einem Crosscompiler übersetzt werden. Dieser übersetzt den Quellcode von der Workstation in ein Maschinenprogramm, das auf einem System mit einem anderen Prozessor, in diesem Fall dem Khepera, lauffähig ist. Bevor das übersetzte Programm über die serielle Schnittstelle in den Khepera geladen werden kann, muß es noch in ein spezielles Format, das S-Format, umgesetzt werden. Der wesentliche Unterschied zwischen der Binärdatei und der Datei in S-Format besteht darin, daß im S-Format jedes Byte durch eine Hexadezimalzahl in ASCII-Zeichen repräsentiert wird, ähnlich einem Hex-Dump. Die Umsetzung der Binärdatei in S-Format wird von einem kleinen Hilfsprogramm erledigt. Nun kann das Programm in den Speicher des Khepera geladen werden. Das Steuerungsprogramm wird automatisch gestartet. Ein großer Nachteil ist die fehlende Möglichkeit, das Steuerungsprogramm während der Laufzeit zu debuggen. Eine ausführliche Beschreibung des Ladens von Programmen in den Khepera wird in [Heinzmann,1995] und [K-Team,1994] aufgeführt.

5.3 Khepera Betriebssystem

Der Khepera verfügt über ein Multitasking-Betriebssystem, das in dieser Diplomarbeit erweitert wird. Der Khepera verfügt über keinen mathematischen Coprozessor, weshalb die Verwendung von Fließkommazahlen nur in solchen Fällen anzuraten ist, in denen ganze Zahlen nicht einsetzbar sind. Alle Betriebsystemfunktionen des Khepera können mit Hilfe einer Bibliothek, `bios.h`, genutzt werden, die in das Programm eingebunden ist. Die Bios-Funktionen sind in [Franzi 1994] beschrieben.

5.3.1 Scheduling und Prozeßverwaltung

Das Khepera-Betriebssystem kann gleichzeitig 15 Prozesse verwalten. Die Prozesse werden durch ein Round-Robin-ähnliches Verfahren aktiviert. Jeder Prozeß ist maximal 5 ms aktiv und wird danach geschedulet. Die Prozeßhantierung läßt sich deutlich in Abb. 5.8 erkennen:
Die erste Liste enthält die „Prozesse", die nicht gestartet sind.
Die „Execution"-Liste enthält die Prozesse, die bereit zum Ausführen sind.
In der dritten Liste befinden sich die Prozesse, die zeitbasiert suspendiert sind.
Die letzte Liste enthält die suspendierten Prozesse, die auf ein Ereignis warten, um wieder bereit zum Ausführen zu werden.

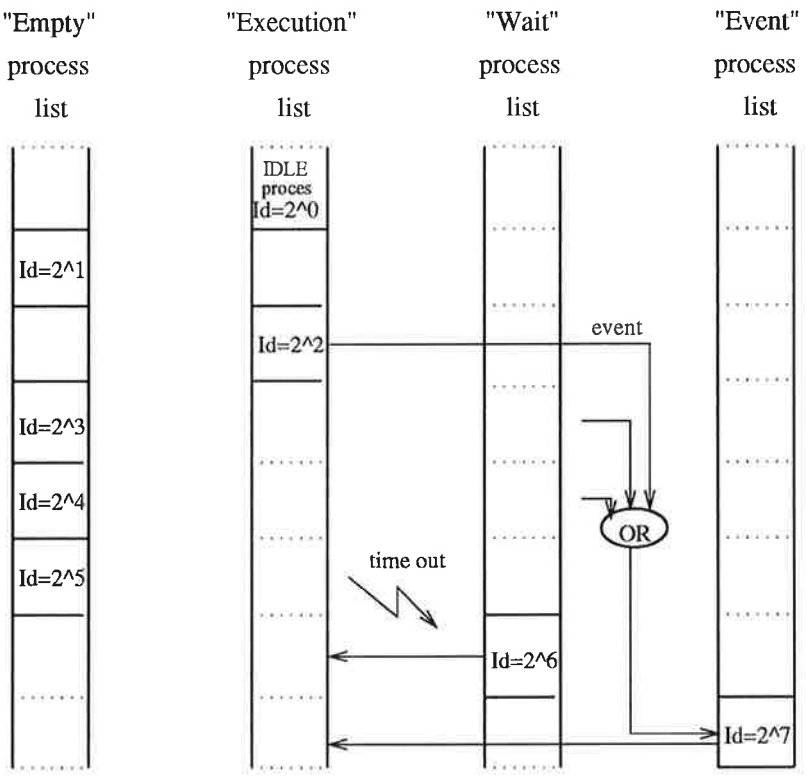

Abbildung 5.8: Prozeßzustände im Khepera-Scheduler

Kapitel 6

CAIC-Arbeitsoberfläche

6.1 *CAIC Presentation Program*

Die Arbeitsoberfläche zu *CIAC* heißt *CAIC Presentation Program* und wurde in *tcl/tk/expect* entwickelt, siehe Abb. 6.1. *Expect* ist eine Erweiterung von *tcl/tk* und unterstützt die Kommunikation mit interaktiven Programmen, in diesem Fall die Kommunikation mit dem *tip*-Program, das die Kommunikation zwischen den Robotern und der Workstation regelt.

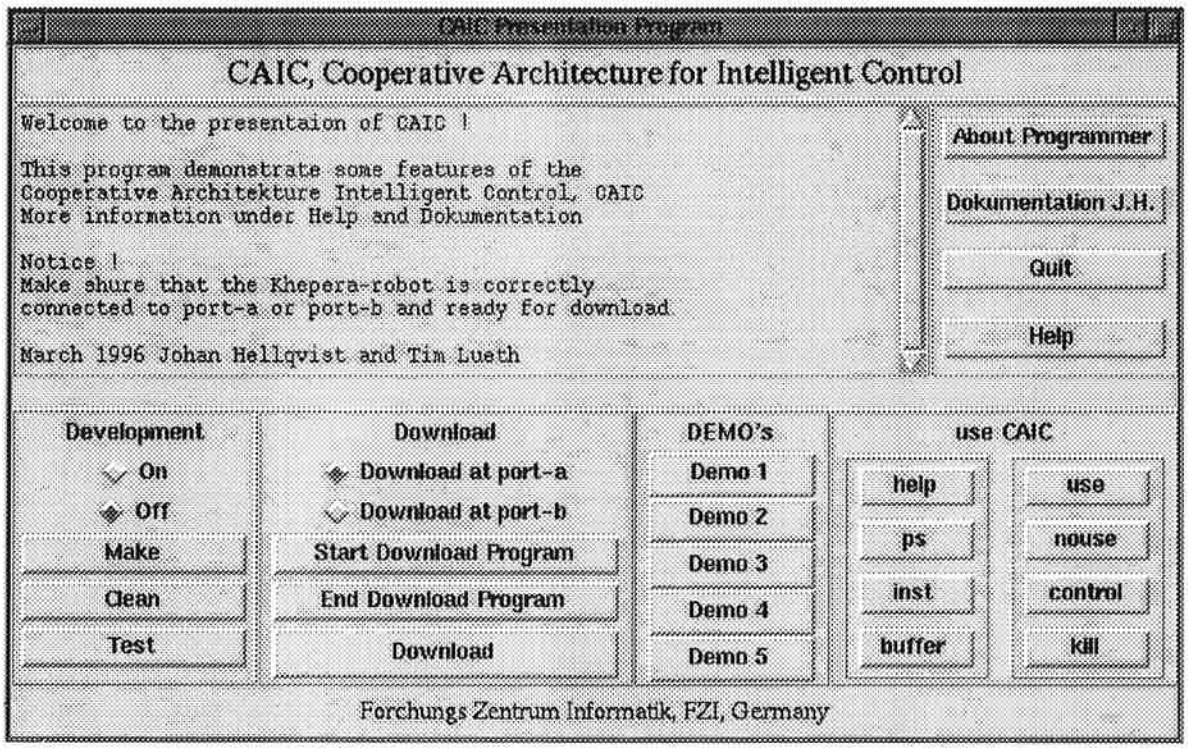

Abbildung 6.1: Darstellung der *tcl/expect*-Arbeitsoberfläche

Das *CAIC Presentation Program* besteht aus fünf Sektionen und einem Präsentations-

fenster. Die fünf Sektionen sind:

1. Die Informationstasten, *About Programmers*, *Documentation J.H.*, *Help* und *Quit*, siehe Abschnitt 6.1.1.
2. Die *Development*-Sektion, siehe Abschnitt 6.1.2.
3. Die *Download*-Sektion, siehe Abschnitt 6.1.3.
4. Die *DEMO's*-Sektion, siehe Abschnitt 6.1.4.
5. Die „*use CAIC*-Sektion", siehe Abschnitt 6.1.5.

Das *CAIC Presentation Program* wird mit der Maus bedient. Die Kommunikation mit dem Roboter wird entweder mit der *use CAIC*-Sektion gesteuert oder manuell im Fenster, in dem das *CAIC Presentation Program* gestartet wurde, eingegeben. In dem gleichen Fenster werden alle Meldungen vom Roboter geschrieben und Kommandos an den Roboter eingegeben. Das bedeutet in der Praxis, daß das *CAIC Presentation Program* im Vordergrund eines *term*-Fensters gestartet werden muß.

Allgemein funktioniert die linke Maustaste als „Aktivierungstaste" und die mittlere, in dem Fall, daß es etwas „unter" dem Button gibt, als „Info- und Hilfstaste". Die Information vom *CAIC Presentation Program* wird im Präsentationsfenster gezeigt und die Information vom und zum Roboter im „Startfenster".

6.1.1 Die Informationstasten

Die Informationstasten oben rechts enthalten Hilfe und Informationen über *CAIC* allgemein und über die Handhabung des *CAIC Presentation Program.*

About Programmers Adressen und Personeninformationen der Programmierer und Entwickler von *CAIC.*

Dokumentation J.H. Startet das *xdvi*-Programm und zeigt diese Arbeit an.

Quit *CAIC Presentation Program* beenden.

Help Hilfe zur Handhabung des *CAIC Presentation Program.*

Alle Informationen, außer *Dokumentation J.H.*, werden im Präsentationsfenster angezeigt.

6.1.2 Die *Development*-Sektion

In dieser Sektion kann man den Quell-Code von *CAIC* kompilieren und Binärcode auf den Roboter laden. Um diese Sektion zu aktivieren muß die Radiotaste in der Position *On* sein, um unerwünchte Kompilierungen usw. zu vermeiden.

Make Kompiliert den Quellcode im Verzeichnis *CAIC/ca* zu der auf einem Khepera-Roboter ladbaren Datei, *caic.s.*

Clean Löscht alle kompilierten Dateien im Verzeichnis *CAIC/ca*. Diese Funktion sollte nur benutzt werden, wenn man eine Änderung der Programmstruktur vorgenommen hat.

Test Laden der letzten Version des *CAIC* auf den Khepera-Robotern. Die aktuelle Schnittstelle (*port-a* oder *port-b*) ist mit den Radiotasten in der *Download*-Sektion einzustellen.

6.1.3 Die *Download*-Sektion

Diese Sektion kontrolliert das Starten des *tip*-Programms und das Laden von ausführbaren *CAIC*-Programmversionen. Die Wahl der Schnittstelle *port-a* oder *port-b* ist global, d.h. beeinflußt alle Programmsektionen. Man kann beide Schnittstellen gleichzeitig aktiviert haben, aber man kontrolliert nur die Schnittstelle, die gewählt ist.

Start Downloadprogram Startet das *tip*-Program an der gewählten Schnittstelle.

End Downloadprogram Beendet das *tip*-Program an der gewählten Schnittstelle.

6.1.4 Die *DEMO's*-Sektion

Laden eines Demo-*CAIC*-Programms an der gewählten Schnittstelle. Mit Hilfe der mittleren Maustaste erhält man eine kurze Beschreibung der Demo.

6.1.5 Die *use-CAIC*-Sektion

Mit dieser Programsektion kann man alle *CAIC*-Kommandos interaktiv während des Betriebs des Roboters benutzen. Die Kommandos werden vom Roboter interpretiert und bearbeitet. Kommandos können auch direkt im Startfenster eingetippt werden, z.B. wenn man ein nicht vorgespeichertes *control*-Kommando senden möchte. Diese Kommandos werden über die serielle Schnittstelle zum Roboter geschickt. Die gesendeten Kommandos kann man im Startfenster sehen. Alle Meldungen und Informationen vom Roboter werden auch im Startfenster angezeigt.

help Kurze Information über die verfügbaren *CAIC*-Kommandos.

ps Listet Informationen über die im Moment laufenden Instanzen.

inst Liefert Informationen über eine ausgewählte Instanz zurück. Die Instanz wird aus einem Auswahlfenster ausgewählt. Die Namen der implementierten Instanzen sind in der Datei *instances.txt* aufgeführt. Wenn man eine neue Instanz erstellt, sollte man den Namen der neuen Instanz in diese Datei einfügen.

buffer Zeigt dan Inhalt eines ausgewählten Puffers an. Der Puffer wird aus einem Wahlfenster ausgewählt. Die Namen der implementierten Puffer sind in der Datei *buffers.txt* gespeichert. Wenn man einen neuen Puffer erstellt hat, muß man den Puffer in diese Datei einfügen.

use Führt das Kommando *use* aus. Vorgehensweise wie oben.

nouse Führt das Kommando *nouse* aus. Vorgehensweise wie oben.

control Im *CAIC Presentation Program* sind einige *control*-Befehle gespeichert. Diese werden nach dem obigen Verfahren zum Roboter geschickt. Es ist praktisch, häufig benutzten *control*-Sequenzen zu speichern, weil sie ziemlich lang sein können. Die gespeicherten *control*-Befehle findet man in der Datei *control.txt*.

kill Beendet *CAIC* und reinitialisiert den Roboter.

Falls der Roboter einmal abstürzen sollte (was hoffentlich nie passieren), so drücken Sie die Reset-Taste am Roboter und verfahren weiter, als ob Sie das Kommando ***kill*** ausgeführt hätten. Das *CAIC*-System selbst ist nicht beendet, wenn das *CAIC Presentation Program* beendet wird, weil *CAIC* selbständig auf dem Zielsystem läuft.

Kapitel 7

Experimente und Beispiele

Ein wichtiger Maßstab eines Echtzeitbetriebssystems ist, wieviel Zeit die Systemfunktionen brauchen. Ein Nachteil des Khepera-BIOS ist, daß man nur mit einer Genauigkeit von $1ms$ messen kann. In Tabelle 7.1 sind alle Funktionen 1000mal ausgeführt, um eine Genauigkeit von μs zu erreichen.

Die Zeit für das Messen und Ausschreiben: $0,5ms$

Funktion	Kommentar	Laufzeit
`CA_INST_init`	Davon Speicherreservierung und Initialisierung der Ein- und Ausgabepufferliste	$30,0ms$ $28,3ms$
`CA_BUFF_make`	SINGLE-Puffer MULTI-Puffer	$3,1ms$ $3,1ms$
`CA_BUFF_connect`		$3,2ms$
`CA_INST_itime_set`		$51\mu s$
`CA_INST_use`	Die Instanz läuft bereits	$101\mu s$
`CA_INST_nouse`		$85\mu s$
`CA_get_time`		$33\mu s$
`CA_generate_event`		$45\mu s$
`CA_define_association`		$168\mu s$
`CA_find_association`		$70\mu s$

Tabelle 7.1: Einige Systemfunktionen und deren Laufzeit.

Die Zeit für Systemfunktionen und Initialisierung wurde mit einer leeren Instanz (nur Systemfunktionen) gemessen.

Die Zeit für die Initialisierung einer Instanz: $28 - 29ms$

Die Zeit für die Systemfunktionen in der endlosen Schleife: $155\mu s$

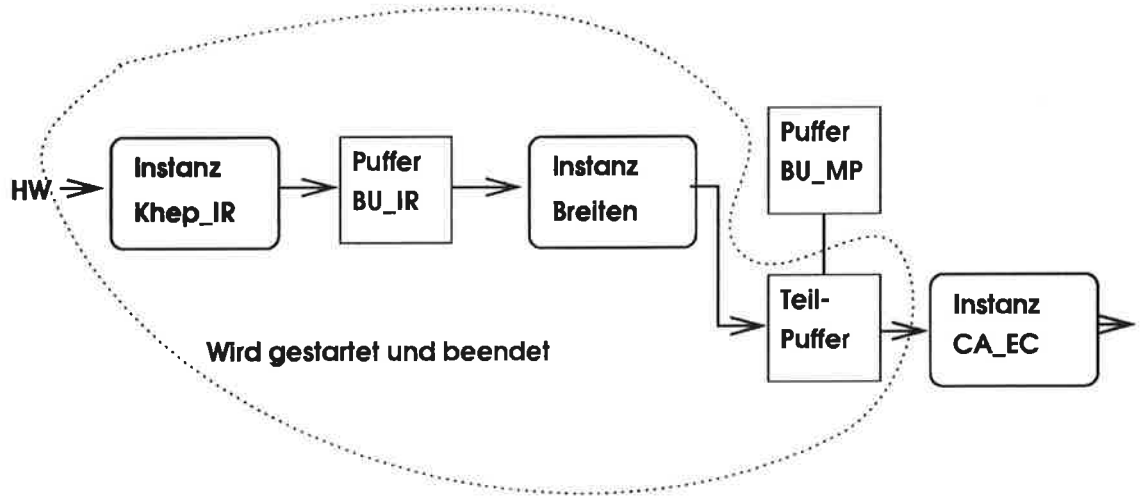

Abbildung 7.1: Hindernisvermeidung

Im Abschnitt 4.10 wurde der Bedarf eines echtzeitfähigen Defragmentierers diskutiert. Um zu testen, ob die Fragmentierung in dieser Implementierung ein reales Problem ist, wurden zwei Instanzen (*Breiten* und *Khep_IR*) 1000mal ohne Probleme gestartet und beendet, siehe Abb. 7.1. *Breiten* reserviert Speicher für einen Teilpuffer des MULTI-Puffers `BU_MP` und startet *Khep_IR*. *Khep_IR* reserviert Speicher für einen SINGLE-Puffer `BU_IR`.
Die Initialisierungszeiten für die Instanzen *Breiten* und *Khep_IR* wurden kontinuierlich gemessen. Ein Unterschied der Initialisierungszeiten aufgrund von Fragmentierung wurde nicht festgestellt.

Die Initialisierungszeit für *Khep_IR*: $32 - 33ms$

Die Initialisierungszeit für *Breiten*: $77 - 79ms$

Die Initialisierungszeit einer leeren Instanz ist $28 - 29ms$.
Die zusätzliche Zeit für *Khep_IR* ist also: $3 - 5ms$

Breiten startet erst *Khep_IR* und wartet, bis sie fertig initialisiert ist. Wenn man auch die Initialisierungszeit um die Initialisierungszeit einer leeren Instanz subtrahiert: $15 - 19ms$

7.1 *mvalong*, Beispiel einer Strukturerzeugung

In *CAIC* erzeugt ein Kommando eine ganze Regelungstruktur. Dieses Kommando kann entweder ein interaktiver Befehl, der über die serielle Schnittstelle gesendet wird, sein oder eine Systemfunktion innerhalb einer Instanz.In diesem Beispiel wird es gezeigt, wie ein interaktiver Befehl *use mvalong*, schrittweise eine Regelungsstruktur erzeugt. Anfangsweise laufen nur die Root-Instanzen. Die intressante Root-Instanz in diesem

Fall ist *CA_EC*, die auch den MULTI-Puffer BU_MP verwaltet.

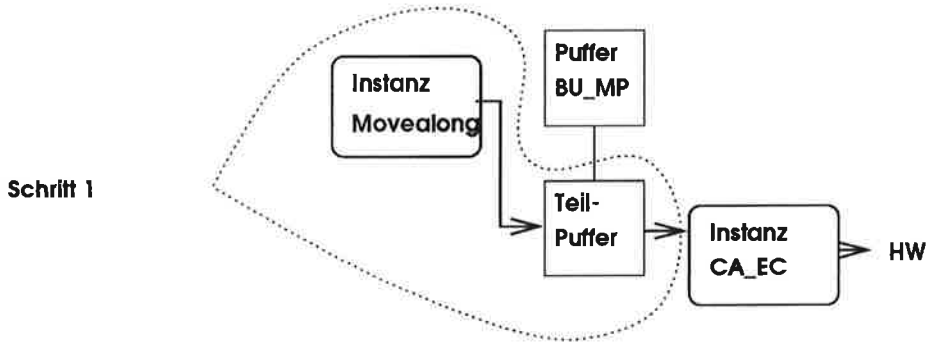

Abbildung 7.2: Strukturerzeugung, Schritt 1

Als erster Schritt wird die *mvalong* initialisiert, und der Teilpuffer des BU_MP wird im Speicher reserviert, siehe Abb. 7.2. Um seine Aufgabe lösen zu können, braucht *mvalong* andere Instanzen und deren Puffer. Es wird durch die Systemfunktion CA_INST_use durchgeführt:

```
/* INST default use begin*/
 CA_INST_use(CA_EC, "CA_EC");
 CA_INST_use(Khep_IR, "Khep_IR");          /*Schritt 2*/
 CA_INST_use(dontstay, "dontstay");        /*Schritt 3 und 4*/
 I.meminit = READY;
 /* INST default use end*/
```

Diese drei Systemfunktionen erzeugen die unten genannten Strukturänderungen, siehe Abb. 7.3, Abb. 7.4 und Abb. 7.5.

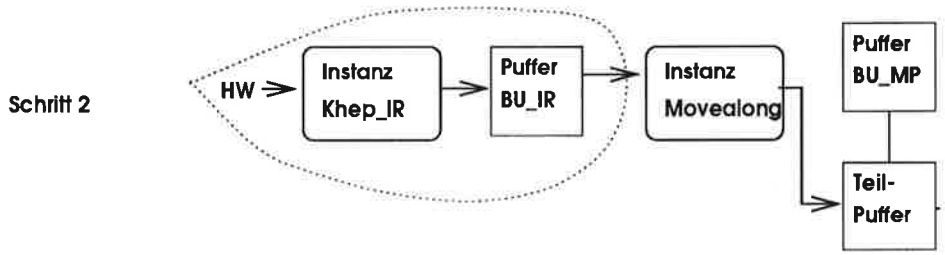

Abbildung 7.3: Strukturerzeugung, Schritt 2

1. CA_INST_use(CA_EC, ''CA_EC''). *CA_EC* wird wegen des BU_MP Puffers benutzt. *CA_EC* läuft schon, deren CSI.use wird um eins erhöht. Siehe Abb. 7.2.

2. CA_INST_use(Khep_IR, ''Khep_IR''). *Khep_IR* wird gestartet, initialisiert sich selbst und reserviert Speicher für den SINGLE-Puffer BU_MP. *Khep_IR* braucht keine zusätzliche Instanz. Siehe Abb. 7.3.

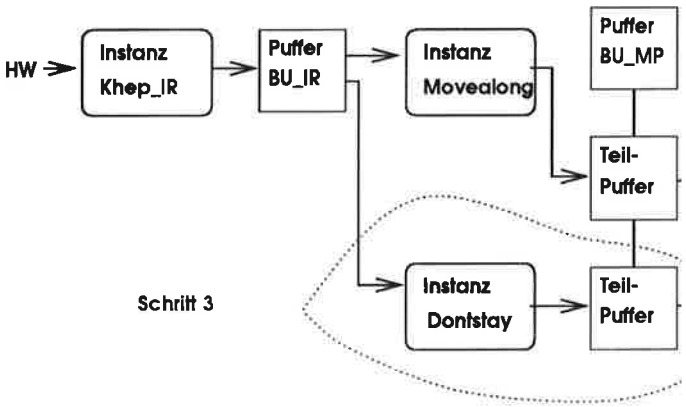

Abbildung 7.4: Strukturerzeugung, Schritt 3

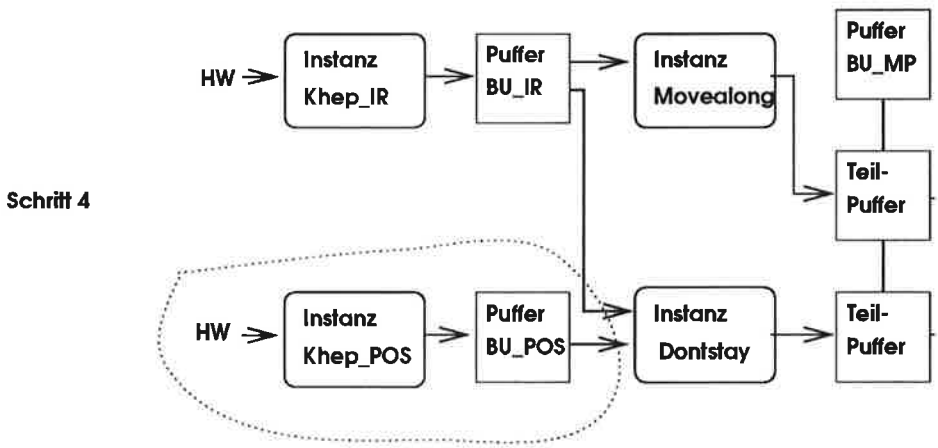

Abbildung 7.5: Strukturerzeugung, Schritt 4

3. `CA_INST_use(dontstay, ''dontstay'')`. *dontstay* braucht andere Instanzen um zu funktionieren. Es macht das gleiche wie *mvalong*; startet die Instanzen, die noch nicht laufen. Siehe Abb. 7.4

   (a) `CA_INST_use(CA_EC, ''CA_EC'' )`. *CA_EC* wird benutzt wegen des `BU_MP` Puffers. *CA_EC* läuft schon und deren `CSI.use` wird um eins erhöht.

   (b) `CA_INST_use(Khep_IR, ''Khep_IR'')`. *Khep_IR* läuft schon, deren *CSI.use* wird um eins erhöht.

   (c) `CA_INST_use(Khep_POS, ''Khep_POS'')`. *Khep_POS* wird gestartet und reserviert Speicher für den SINGLE-Puffer `BU_POS`, siehe Abb. 7.6.

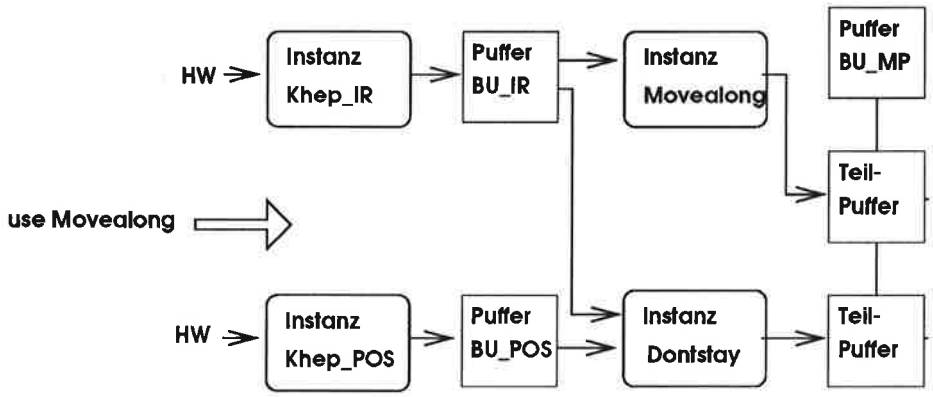

Abbildung 7.6: Die gesamte Strukturerzeugung

Abbildung 7.6 stellt die gesamte Strukturerzeugung dar. Wie gesehen, verläuft die Strukturerzeugung nach einer rekursiven Methode. Die Initialisierungszeit hängt von der Anzahl der schon gestarteten Instanzen ab und ist schwer zu schätzen. Das obige Beispiel sollte man als *Worst-Case* betrachten.

7.1.1 Die Instanz *mvalong* in Quell-Code

```
void mvalong()
{
  /*system variable */
  CSI    I;          /*pointer to Information block*/
  /* Task information*/
  /*instance variable*/
  BU_IR       *inIR;             /*pointer to sensor buffer*/
  type_IR          IR;             /*parameter to mvlong*/
  BU_MP       *outMP;            /*pointer to out move buffer*/
  int32       l=0,r=0;              /*speed_left speed_right*/
  int32       w=0;                 /*weight*/
  /***************INIT begin **********************/
  /*CA_IDENT default begin*/
  strncpy(I.name,"mvalong",16);
```

```
strncpy(I.to_trigg,"",16);
I.cycletime = 100;     I.state = 1;
I.cycletoler = 5;     I.run_mode = cyclus;
/*CA_IDENT default end*/
CA_INST_init(&I);
/*LOCAL variable init begin*/
init_IR(&IR);
/*LOCAL variable init begin*/
/* INST default use begin*/
CA_INST_use(CA_EC, "CA_EC");
CA_INST_use(Khep_IR, "Khep_IR");
CA_INST_use(dontstay, "dontstay");
I.meminit = READY;
/* INST default use end*/
/*BUFFER connect begin*/
inIR =  (BU_IR *) CA_BUFF_connect(&I, Bin, SINGLE, "BU_IR");
outMP = (BU_MP *) CA_BUFF_connect(&I, Bout , MULTI, "BU_MP");
CA_BUFF_register(&I, "dontstay");
/*BUFFER connect end*/
CA_INST_itime_set(&I);
/************** INIT end ************************/
for(;;)
  {
    /********** PRESTATE begin ******************/
    CA_PRE_STATE(&I);
    /* READ buffers begin*/
    READ_BU_IR(/*in*/inIR,/*out*/ &IR);
    /* READ buffers end*/
    /********* PRESTATE end *********************/
    /********* STATE begin **********************/
    switch((int)(I.state))
      {
      case 1:
        /*STATE 1*/
        movealong(/*par*/&I,2,/*in*/ &IR, /*out*/ &l,&r,&w);
        break;

      default:
        CA_error_abort("inst","no such state");
      }
    /******** STATE end ************************/
    /******** POSTSTATE begin ******************/
    /*write buffer begin*/
    CA_tim_lock();
    outMP->speed[0] = l;
    outMP->speed[1] = r;
    CA_tim_unlock();
    /*write buffer end*/
```

```
      CA_POST_STATE(&I); /*CA_generate_event etc..*/
      /******** POSTSTATE end *********************/
    }
}
```

7.1.2 Die Instanz *Khep_IR* in Quell-Code

```
void Khep_IR()
{
  /*system variable */
  CSI    I;          /*pointer to Information block*/
  /* Task information*/
  /*instance variable*/
  CA_BU    *BUFF=NULL;
  BU_IR    *outIR= NULL;                /*pointer to IR-Buffer*/
  int16      i=0;                   /*counter*/
  int16 *sensorfield=NULL;
  /***************INIT begin **********************/

  /*CA_IDENT default begin*/
  strncpy(I.name,"Khep_IR",16);
  strncpy(I.to_trigg,"",16);
  I.cycletime = 100;    I.state = 1;
  I.cycletoler = 20;    I.run_mode = cyclus;
  /*CA_IDENT default end*/
  CA_INST_init(&I);

  /*BUFFER make begin*/
  BUFF =  CA_BUFF_make(&I, SINGLE, "BU_IR");
  /*BUFFER make end*/

  I.meminit = READY;
  /*BUFFER connect begin*/
  sensorfield=sens_get_pointer(); /* to HARDWARE*/
  outIR = (BU_IR *) CA_BUFF_connect(&I, Bout ,MAKER, "BU_IR");
  /*BUFFER connect end*/

 CA_INST_itime_set(&I);
  /************** INIT end ************************/

  for(;;)
    {
      /********** PRESTATE begin *****************/
      CA_PRE_STATE(&I);

      /********* PRESTATE end *********************/

      /********* STATE begin **********************/
```

```
      switch((int)(I.state))
        {
        case 1:
          /*STATE 1*/
             CA_tim_lock();
             outIR->time=I.time;
             outIR->dtime=I.dtime;
             outIR->id = I.id;
             outIR->cycletime = I.cycletime;
             for (i=0;i<6;i++)
               outIR->PR[i]=sensorfield[i];
             outIR->PR[6]=(sensorfield[5]+sensorfield[6])>>1;
             outIR->PR[7]=sensorfield[6];
             outIR->PR[8]=sensorfield[7];
             outIR->PR[9]=(sensorfield[0]+sensorfield[7])>>1;
             for (i=0;i<6;i++)
               outIR->AM[i]=sensorfield[8+i];
             outIR->AM[6]=(sensorfield[13]+sensorfield[14])>>1;
             outIR->AM[7]=sensorfield[14]; outIR->AM[8]=sensorfield[15];
             outIR->AM[9]=(sensorfield[8]+sensorfield[15])>>1;
             CA_tim_unlock();
           /*  CA_printf( CA_GL_SE, "ir1: %d, ir2: %d\r\n", (int)
                          outIR->PR[1], (int)  outIR->PR[2]);*/
          break;
        default:
          CA_error_abort("STATE default","no such state");
        }
      /******** STATE end ************************/

      /******** POSTSTATE begin ******************/
      CA_POST_STATE(&I);
      /******** POSTSTATE end ********************/
    }
}
```

7.2 *Avoid*, Beispiel einer Steuerflußänderung

Avoid ist eine Instanz, die andere durch die Systemfunktion `CA_INST_control` beeinflußt.
Diese Instanz ist ein Beispiel einer Steuerflußänderung zur Laufzeit. Am Anfang laufen

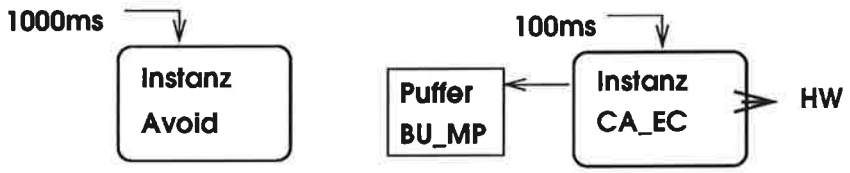

Abbildung 7.7: Darstellung der *Avoid*-Instanz

nur *Avoid* und *CA_EC*, siehe Abb. 7.7. *Avoid* benutzt keinen Puffer. *Avoid* startet die *Default*-Hindernisvermeidungsstruktur mittels der Systemfunktion
`CA_INST_use(Breiten,''Breiten'')`,
die die Struktur in der Abb. 7.8 erzeugt. Diese Struktur läuft anfangs mit unabhängi-

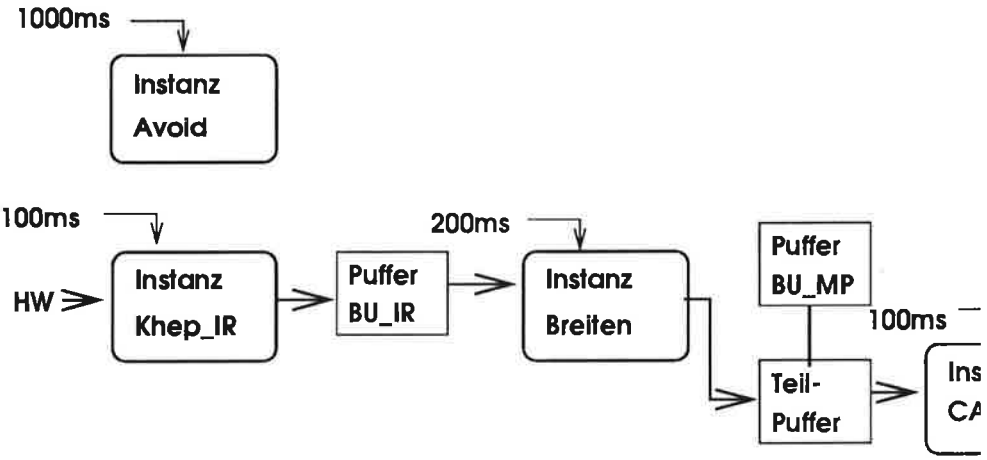

Abbildung 7.8: Darstellung der *Avoid*-Instanz

gen zyklisch aktivierten Instanzen. Um eine engere und effizientere Verknüpfung der Instanzen zu bekommen, ändert *Avoid* den Steuerfluß der Instanzen.

```
CA_INST_control("Khep_IR,cyclus_trig,100,Breiten@");
CA_INST_control("Breiten,event_trig,Khep_IR,CA_EC");
CA_INST_control("CA_EC,event,Breiten@");
```

Der Steuerfluß sieht jetzt wie in der Abbildung 7.9 aus. Die Instanzen sind jetzt eine, von der Instanz *Khep_IR* getriggerte Kette von Instanzen mit einer Zykluszeit von 100ms.
Avoid ändert alle 1000 ms die Zykluszeit in *Khep_IR*:

```
CA_INST_control("Khep_IR,cyclus_trig,150,Breiten@");
```

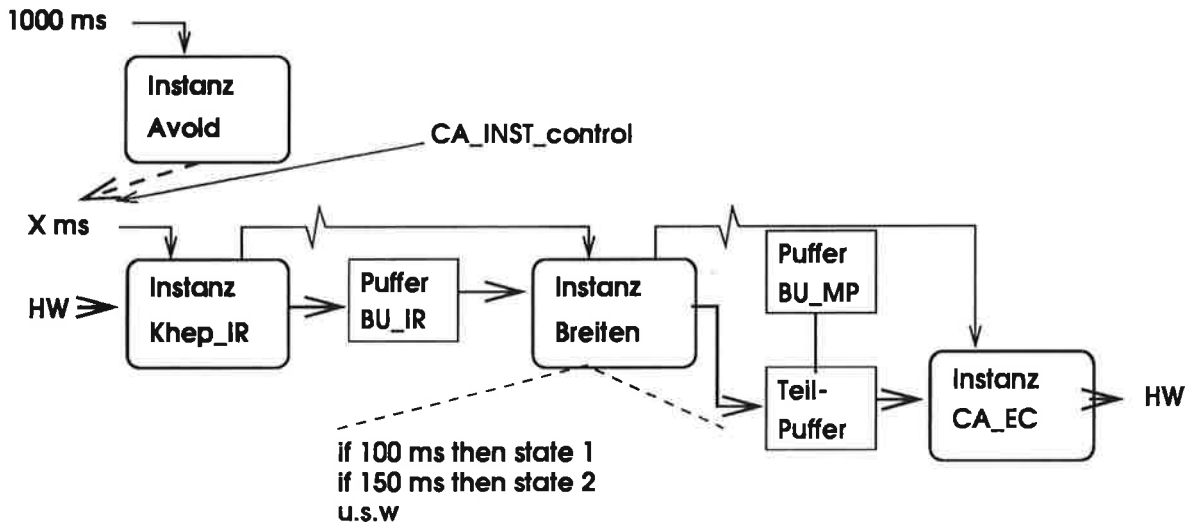

Abbildung 7.9: Darstellung der *Avoid*-Instanz

```
CA_INST_control("Khep_IR,cyclus_trig,200,Breiten@");
```

Um eine instabile Bewegung des Roboters zu vermeiden, kontrolliert die Instanz *Breiten* immer die Zykluszeit ihrer Eingaben. Abhängig von dieser Zykluszeit wechselt *Breiten* den Strukturzustand. Der Strukturzustand setzt eine stabile Geschwindigkeit, abhängig von der Zykluszeit der Eingaben.

7.2.1 Die Instanz *Avoid* in Quell-Code

```
void Avoid()
{ /*system variable */
  CSI    I;          /*pointer to Information block*/
  /* Task information*/
  /*instance variable*/
  /***************INIT begin **********************/
  /*CA_IDENT default begin*/
  strncpy(I.name,"Avoid",16);
  strncpy(I.to_trigg,"",16);
  I.cycletime = 10000;    I.state = 1;
  I.cycletoler = 20;    I.run_mode = cyclus;
  /*CA_IDENT default end*/
  CA_INST_init(&I);
  /*BUFFER make begin*/
  /*BUFFER make end*/
  I.meminit = READY;
  /* INST default use begin*/
  CA_INST_use(Breiten,"Breiten");
  /* INST default use end*/
  /*BUFFER connect begin*/
  CA_BUFF_register(&I,"Breiten");
```

```
/*BUFFER connect end*/
CA_INST_itime_set(&I);
/************** INIT end ************************/
for(;;)
  {
    /********** PRESTATE begin ******************/
    CA_PRE_STATE(&I);
    /* READ buffers begin*/
    /* READ buffers end*/
    /* UPDATE state begin*/
    /* UPDATE state end*/
    /********* PRESTATE end *********************/
    /********* STATE begin **********************/
    switch((int)(I.state))
      {
      case 1:
        /*STATE 1*/
        CA_INST_control("Khep_IR,cyclus_trig,400,Breiten@");
        CA_printf(CA_GL_SE,"Khep_IR,cyclus_trig,400,Breiten@,
                            OK!\n\r");
        CA_INST_control("Breiten,event_trig,Khep_IR,CA_EC");
        CA_printf(CA_GL_SE,",Breiten,event_trig,Khep_IR,CA_EC,
                             OK!\n\r");
        CA_INST_control("CA_EC,event,Breiten@");
        CA_printf(CA_GL_SE,"CA_EC,event,Breiten@, OK!\n\r");
        I.state = 2;
        break;
      case 2:
          CA_INST_control("Khep_IR,cyclus_trig,150,Breiten@");
          CA_printf(CA_GL_SE,"Khep_IR,cyclus_trig,150,Breiten@,
                              OK!\n\r");
          I.state = 3;
        break;
      case 3:
        CA_INST_control("Khep_IR,cyclus_trig,200,Breiten@");
        CA_printf(CA_GL_SE,"Khep_IR,cyclus_trig,200,Breiten@,
                            OK!\n\r");
          I.state = 4;
        break;
      case 4:
        CA_INST_control("Khep_IR,cyclus_trig,500,Breiten@");
        CA_printf(CA_GL_SE,"Khep_IR,cyclus_trig,500,Breiten@,
                            OK!\n\r");
          I.state = 5;
        break;
      case 5:
        CA_INST_control("Khep_IR,cyclus_trig,1000,Breiten@");
        CA_printf(CA_GL_SE,"Khep_IR,cyclus_trig,1000,Breiten@,
```

```
                                OK!\n\r");
            I.state = 6;
          break;
        case 6:
          CA_INST_nouse("Avoid");
          break;
        default:
          CA_error_abort("inst","no such state");
        }
      /******** STATE end ************************/
      /******** POSTSTATE begin ******************/
      /*write buffer begin*/
      /*write buffer end*/
      CA_POST_STATE(&I); /*CA_generate_event etc..*/
      /******** POSTSTATE end ********************/
    }
}
```

7.2.2 Die Instanz *Breiten* in Quell-Code

```
void Breiten()
{

  /*system variable */
  CSI    I;
   /* Task information*/
  /*instance variable*/
  int32 l=0,r=0;              /*speed_left and _right */
  BU_IR *inIR=NULL;                /*pointer to BU_IR buffer*/
  type_IR IR;                 /*local IR buffer*/
  BU_MP *utMP=NULL;                /*pointer to BU_MP buffer*/
  int32 w1=0;                   /*weight*/
  /*test variable*/
  /*  int k=0;
  /***************INIT begin **********************/
  /*default*/
  strncpy(I.name,"Breiten",16);
  strncpy(I.to_trigg,"",16);
   I.cycletime = 100;     I.state = 1;
  I.cycletoler = 20;     I.run_mode = cyclus;

  CA_INST_init(&I);
  /*BUFFER make begin end*/
  /*LOCAL variable init begin*/
  init_IR(&IR);
  /*LOCAL variable init end*/
  I.meminit = READY;
  /*INST use begin*/
```

```
 CA_INST_use(Khep_IR, "Khep_IR");
 CA_INST_use(CA_EC, "CA_EC");
 /*INST use end*/
 /*BUFFER connect begin*/
 inIR = (BU_IR *) CA_BUFF_connect(&I, Bin , SINGLE, "BU_IR");
 utMP = (BU_MP *) CA_BUFF_connect(&I, Bout , MULTI, "BU_MP");
 /*BUFFER connect end*/
 CA_INST_itime_set(&I);
/************** INIT end ************************/

 for(;;)
   {
     /********** PRESTATE begin ******************/
    CA_PRE_STATE(&I);

     /* READ buffers begin*/
     READ_BU_IR(/*in*/inIR,/*out*/ &IR);
     /* READ buffers end*/

     /* UPDATE state begin*/
     if( (I.run_mode==event)||(I.run_mode == event_trig) )
       {
         if( IR.cycletime < 101 )
           I.state = 4;
         else if( (IR.cycletime>=101)&&(IR.cycletime<=150) )
           I.state = 3;
         else if( (IR.cycletime>=151)&&(IR.cycletime<=200) )
           I.state = 2;
         else if( (IR.cycletime>=201) && (IR.cycletime<=500))
           I.state = 1;
         else
           I.state = 5;
       }
     else if((I.run_mode==cyclus)||(I.run_mode==cyclus_trig))
       {
         if( (IR.cycletime < 101))
           I.state = 2;
         else if( (IR.cycletime>=101)&&(IR.cycletime<=500))
           I.state = 1;
         else
           I.state = 5;
       }
     else
       CA_error_abort("Breiten","State failure");
     /* UPDATE state end*/

     /********* PRESTATE end *********************/
```

```
    /********* STATE begin **********************/

    switch((int)(I.state))
      {
      case 1:
        /*STATE 1 speed = 5*/
        breiten(/*par*/&I,5,/*in*/ &IR,/*out*/ &l,&r,&w1);
        break;

      case 2:
        /*STATE 2 speed = 10*/
        breiten(/*par*/&I,10,/*in*/ &IR,/*out*/ &l,&r, &w1);
        break;
      case 3:
        /*STATE 3 speed = 15*/
        breiten(/*par*/&I,15,/*in*/ &IR,/*out*/ &l,&r, &w1);
        break;
      case 4:
        /*STATE 4 speed = 20*/
         breiten(/*par*/&I,20,/*in*/ &IR,/*out*/ &l,&r, &w1);
        break;
      case 5:
        /*STATE 5 IR too slow*/
        CA_printf(CA_GL_SE,"Breiten: IR too slow,
                            terminating!!\n\r");
        l = 0;
        r = 0;
        I.use = 0;
        break;
      default:
        CA_error_abort("avoidobst","no such state");
      }
    /******** STATE end ************************/
    /******** POSTSTATE begin ******************/
    /*write buffer begin*/
    CA_tim_lock();
    utMP->speed[0] = l;
    utMP->speed[1] = r;
    CA_tim_unlock();
    /*write buffer end*/
    CA_POST_STATE(&I);
   /******** POSTSTATE end ********************/
  }
}
```

7.3 Abschlußexperiment: Kooperation mittels lokaler IR-Kommunikation

Das abschließende Experiment diente dazu, die zukünftigen Möglichkeiten der kooperativen Roboterarchitektur auszuloten und zu demonstrieren. Ein wichtiges Ziel von KACOR (Karlsure Cooperative Robots) ist die kooperative Durchführung nicht trivialer Montageaufgaben, wie sie beispielsweise der vorgestellte Cranfield Assembly Benchmark repräsentiert.
Beim ersten gemeinsamen Einsatz des lokalen Inter-Roboter-Kommunikationssystems und der kooperativen Steuerungsarchitektur CAIC, wurde eine Aufgabe gewählt, die entscheidend für das Gelingen der Benchmarkmontage ist: Das „Spacingpiece" des Cranfield Assembly Benchmark (vgl. Abb. 7.10) kann aufgrund seines Gewichtes nur von zwei Kheperas angehoben werden. Zur Montage muß es jedoch auf die Grundplatte mit den Abstandshaltern aufgesetzt werden.

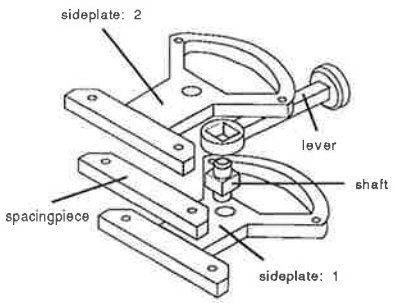

Abbildung 7.10: Cranfield Assembly Benchmark

Bereits in vorangegangenen Studienarbeiten hat sich gezeigt, daß mit den Kheperas die Benchmarkteile durch Umfahrung und Wegmessung zweifelsfrei identifiziert werden können [Heinzmann 95], [Hellqvist 95]. Auch die Plazierung und Orientierung der Miniaturroboter an bestimmten Ecken oder Seiten der Montageteile ist bereits erprobt worden.

An dieser Stelle setzt das Abschlußexperiment ein: Die Kheperas stehen sich gegenüber an den Kopfenden des Spacingpiece plaziert. Das Teil soll simultan gegriffen, angehoben und anschließend abtransportiert werden. Dabei wird ein dynamisches Master-Slave-Verhältnis angestrebt, bei dem immer der Roboter die Kontrolle innehat, in dessen Richtung der Abtransport stattfindet. Hintergrund ist das gleichzeitig aktive Verhalten zur Hindernisvermeidung, mit dem auf diese Weise bei Kollisionsgefahr ein Nothalt und eine Richtungsumkehr ausgelöst werden können. Bei der Richtungsumkehr übernimmt der andere Roboter die Führung.

In den folgenden Aufnahmen (Abb. 7.11-7.14) ist der zeitliche Ablauf eines Experiments festgehalten.

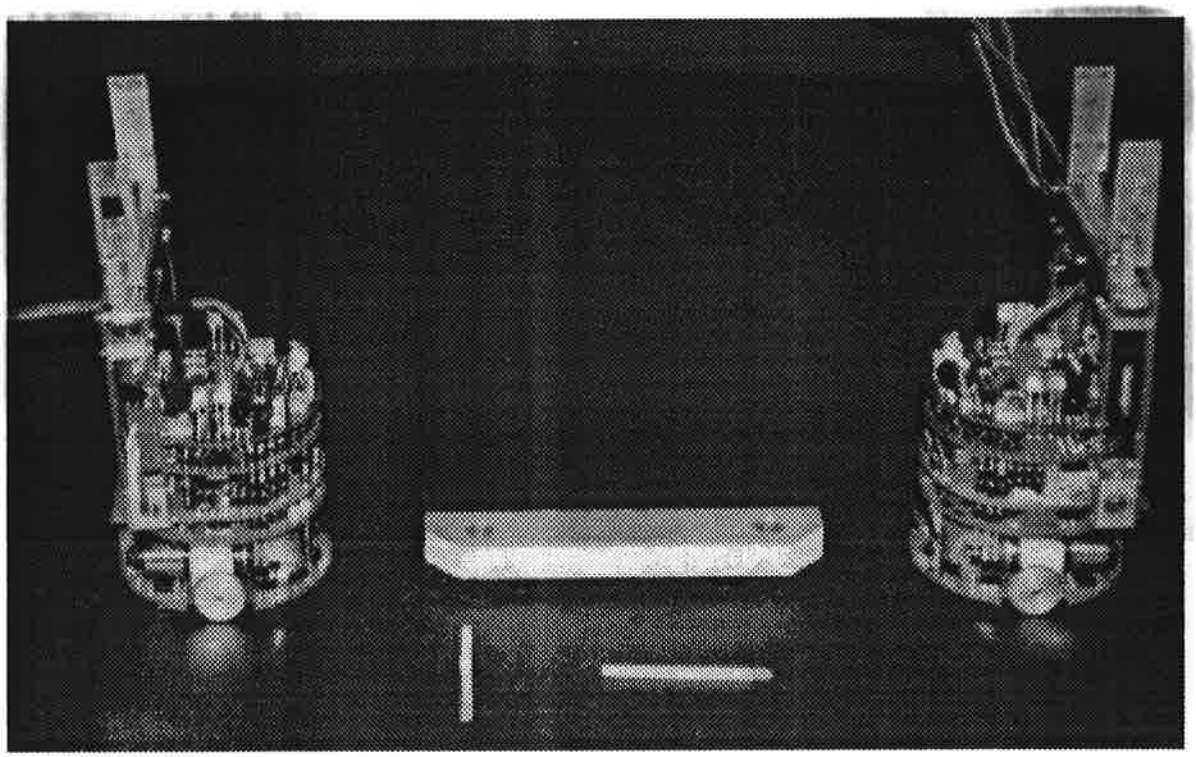

Abbildung 7.11: Abschlußexperiment: Startzustand

Abb. 7.11 zeigt den Zustand zu Beginn des Experiments: Die beiden Fahrzeuge haben das Spacingpiece identifiziert und sich in greifbarer Nähe an den Kopfenden des Teils plaziert. Der Roboter der zuerst die Führungsrolle übernimmt initiiert die Kontaktaufnahme über das IRC-Modul. Sobald auf seine Anforderung eine Verbindung zustande gekommen ist, sendet er das CAIC-Kommando zum Senken des Greifers per Infrarot an den Kommunikationspartner. Dieser wiederum bestätigt die Nachricht und beide Roboter senken die Greifer ab.

Die Situation mit abgesenkten Greifern ist in Abb. 7.12 dargestellt. Die Greiferbacken schließen sich, sobald das Spacingpiece die eingebaute Lichtschranke im Greifer verdeckt. Nach dem Greifen sind die Roboter bereit für das Anheben des Teils. Wiederum sendet der führende Roboter einen CAIC-Befehl aus, und wartet auf die Bestätigung durch den Gehilfen. Dann werden simultan die Greifer mit dem Montageteil gehoben.

In Abb. 7.13 sind die Kheperas mit dem angehobenen Spacingpiece zu sehen. Der Abtransport beginnt dann nach neuerlicher Absprache in Richtung des führenden Roboters.

Die Transportphase geht aus Abb. 7.14 hervor. Sobald der führende Roboter ein Hindernis detektiert und Kollisionsgefahr besteht, wird eine Nachricht verschickt, und damit die Bewegung beider Roboter gestoppt. In dem folgenden Nachrichtenaustausch wird das neue Master-Slave-Verhältnis ausgehandelt: Der bisher führende Roboter

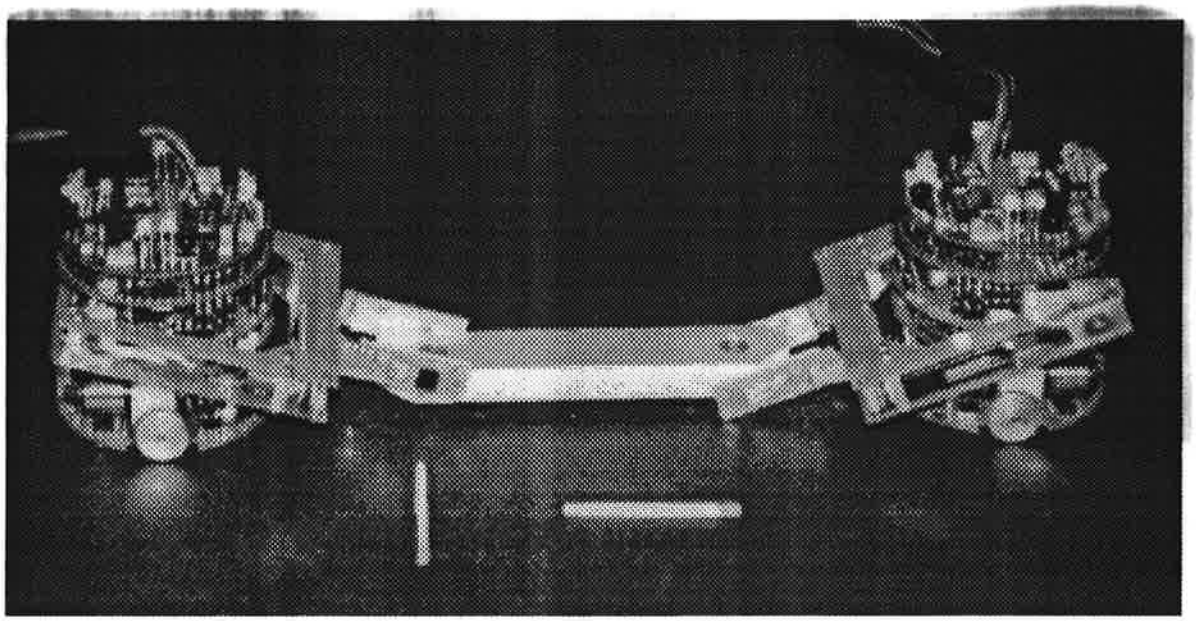

Abbildung 7.12: Abschlußexperiment: Greifphase

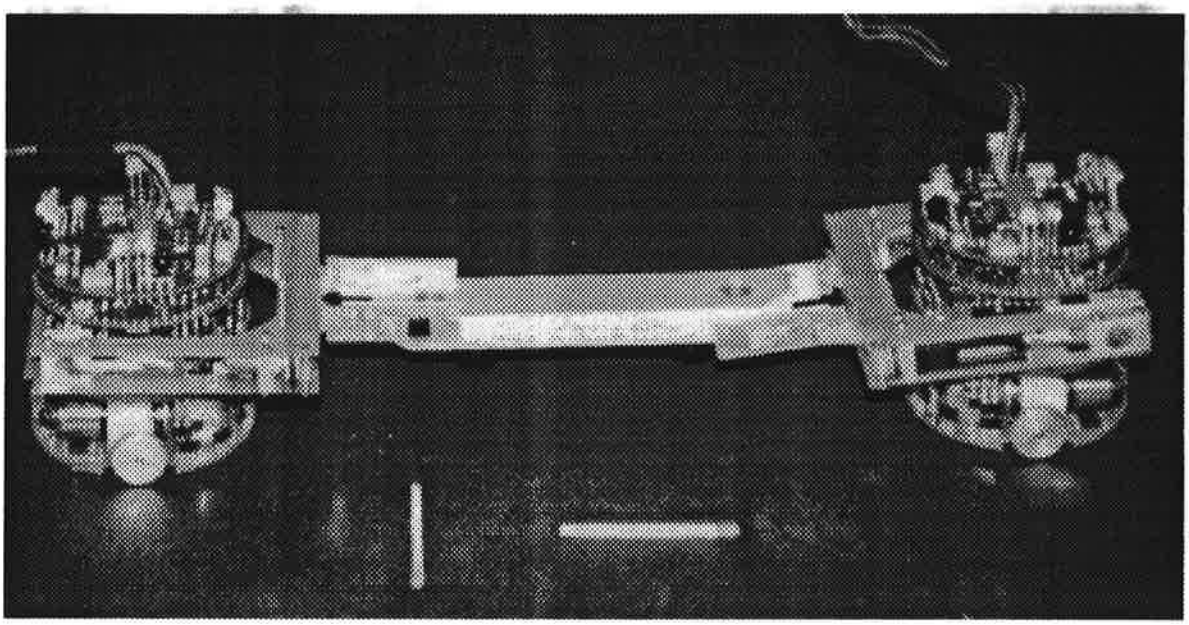

Abbildung 7.13: Abschlußexperiment: Hebephase

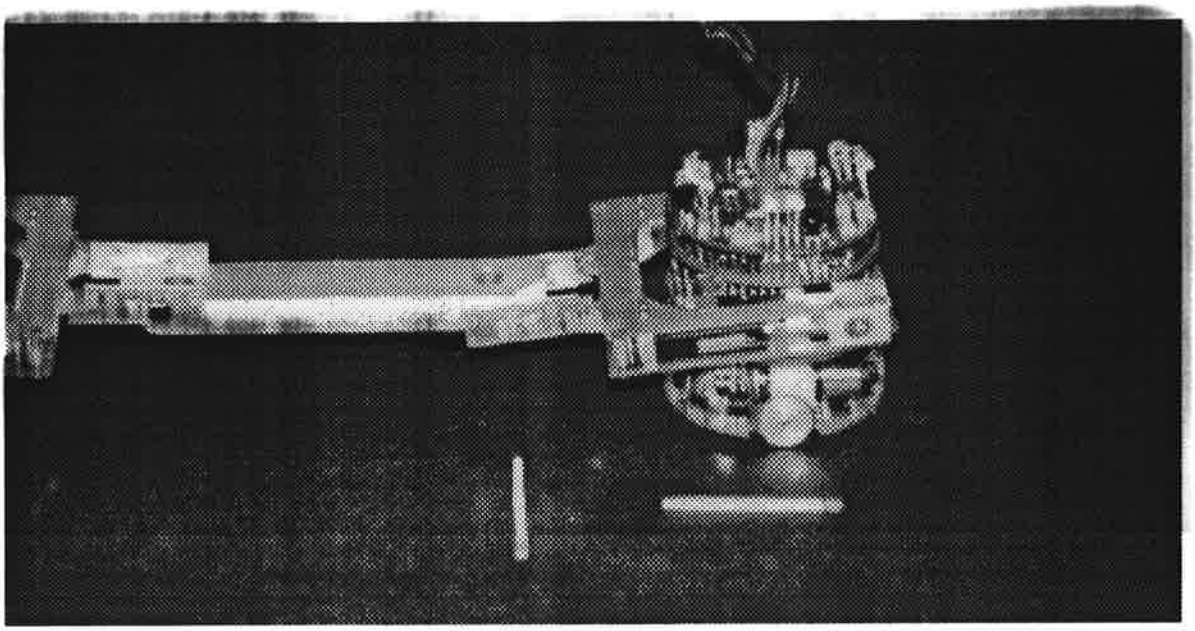

Abbildung 7.14: Abschlußexperiment: Transportphase

gehorcht nach der Umkehr der Transportrichtung nun dem bisherigen „Sklaven". Dieses erste Beispiel tatsächlicher Kooperation resultiert bei der Anwesenheit von Hindernissen in einer simplen Vorwärts-/Rückwärtsbewegung. Allerdings ist das KACOR-Projekt damit bereits ein wichtigen Schritt vorangekommen: Teile können identifiziert und -nach Absprache über das lokale Kommunikationssystem- simultan gegriffen und kooperativ manipuliert werden.

Kapitel 8

Zusammenfassung und Ausblick

8.1 Zusammenfassung

Steuerung autonomer Roboter erfordert komplexe und veränderliche Systeme, in denen mehere zeit- oder ereignisbasierte Regelkreise zusammenwirken. Beim gegenwärtigen Stand der Technik entscheidet bereits nicht mehr die Qualität des einzelnen Algorithmus über die Leistungsfähigkeit, sondern vielmehr die Steuerungsarchitektur. Daher muß sie systematisch betrachtet und an die spezielle Aufgabenstellung angepaßt werden.
Es wurde eine Steuerungsarchitektur vorgestellt, mit der dynamische Veränderungen des Informations- und des Steuerflusses ermöglicht werden, um situationsabhängige Steuerstrukturen, z.B hierarchische und/oder verhaltensorientierte Steuerungsarchitekturen zu erzeugen. Unterstützt wird das Konzept durch einen speziellen Echtzeitbetriebssystemskern, *CAIC*, *Cooperative Architecture for Intelligent Control.* Als Zielplattform wurde für die Implementierung von *CAIC* der Khephera-Roboter [K-Team 1994] ausgewählt. *CAIC* bietet dem Programmierer nur sechs Systemfunktionen an, die die Verwaltung der Instanzen ermöglicht: *use*, *nouse*, *connect*, *register*, *control* und *make.* Alle übrige Funktionen werden vom *CAIC*-Betriebssystem gehandhabt.
Eine grafische Schnittstelle bietet interaktiv die Änderung des Systems durch acht Systembefehle zur Laufzeit an.
Insgesamt wurden drei neue Begriffe und Prinzipien innerhalb dieser Arbeit vorgestellt:

- Strukturadaptive Steuerung.
- H-Scheduling
- MULTI-Puffer

Im Absclußexperiment wurde gezeigt wie zwei Khepera-Roboter mittels lokalen IR-Kommunikation kooperierend einen Cranfield Assembly Benchmark Teil greifen, heben und transportieren können.

8.2 Ausblick

In der Zukunft wird *CAIC* derart weiterentwickelt, daß zusammen mit der Fertigstellung eines lokalen Kommunikationssystems [Grasman 1996] komplexe Aufgabenstellungen durch mehrere Roboter kooperativ bearbeitet werden können. Zukünftige Vorschläge sind:

- Die Implementierung von mehreren Testinstanzen mit einer höheren Komplexität
- Testen anderer Zielplattformen.
- Implimentierung von *CAIC* mit einem plattformunabhängigen Echtzeitbetriebssystemskern.
- Verteilung von *CAIC* auf mehrere Prozessoren.
- Genauere Untersuchung von Deadlocks und der Konsistenz des *CAIC*.
- Untersuchung von echtzeitfähigen Defragmentierungsalgorithmen.

Abbildungsverzeichnis

Tabellenverzeichnis

Anhang A

A.1 Übersicht über Echtzeitbetriebssysteme

Die Tabelle besteht aus zwei Teilen. Der erste Teil dient als eine allgemeine Übersicht zum Betriebssystem mit den Themengebieten:

- Betriebssystemkategorie
- Zielsystem
- Hostsystem
- Programmiersprache
- Entwicklungssystem
- Debugger
- Dateisystem
- Grafiksystem
- Netzwerkeinbindungen
- Feldbusanbindung

Der zweite Teil der Tabelle stellt Informationen über Mengengerüst und Leistungsdaten des Betriebssystemkern zusammen, mit den Themen:

- Anzahl möglicher Tasks
- Anzahl der Prioritäten
- Schedulingalgorithmen
- Mechanismen zur Taskkommunikation

Erklärung zu einigen Abkürzungen

EZ Echtzeit

EZK Echtzeitkernel

EZBS Echtzeitbetriebssystem

Hersteller	**ARS Integrated System**	**ENEA DATA AB**	**ENEA DATA AB**	**ENEA DATA AB**
Produkt	**pSOS**	**OSE Basic**	**OSE Classic**	**OSE Delta**
Allgemeine Übersicht				
Kategorie	EZBS, EZK, embedded	EZK, embedded	EZK, embedded	EZK, embedded
Zielsystem	680x0, 80x86, 8016x PowerPC, CPU32, i960 Hitachi SH, MIPS	80x51, Z80, 68HCxx	680x0, 8016x, CPU32	680x0, PowerPC, AMD29k, CPU32
Hostsystem	UNIX, SUN, MS-Dos, Windows, NT, OS/2	SUN, MS-Dos, Windows	SUN,MS-Dos, CPU32	UNIX, SUN
Sprache	ASM, Ansi-C, C++ Pascal, ADA	ASM, Ansi-C	ASM, Ansi-C	ASM, C++
Entwicklungs-system	CAD-UL, Microtec, SDS, Greenhills, ALISYS	Borland C++, MS-Visual C, IAR	Microtec, X-Ray	Microtec, DLab
Debugger	XDB (CAD-UL), Microtec, XRAY+SDS, Greenhills	Borland TD, Microsoft Codeview	Borland TD, Microsoft Codeview, X-Ray	X-Ray
Dateisystem	UNIX, DOS, NFS, Real-Time-Filesystem	-	-	UNIX, DOS
Grafiksystem	X-Win, MetaWin für RT	MS-Windows	MS-Windows	-
Netzwerk	TCP/IP, Netware, OSI 1-7, SNMP CMIP, X.25	-	TCP/IP	TCP/IP
Feldbus	CAN-Bus	-	-	-
Sonstiges	32-Bit-Code, ROM-fähig, Multiprozessor fehlertolerant	ROM-fähig Multiprozessor	ROM-fähig Multiprozessor	ROM-fähig Multiprozessor
Leistungs-daten:				
Tasks	65 535	256	8000	unbegrenzt
Prioritäten	256	64	64	64
Scheduler	preemtiv, round robin, prioritätsgesteuert	preeemptive	preemtive	preemptive
Taskkommu-nikation	Semaphore, Messages, Mailbox, Signale	Messages	Messages	Messages
Sonstiges	Message-Queues, Prioritätenvererbung Speicherverwaltung Timer	Message-Queues	Message-Queues	Message-Queues

Tabelle A.1: Übersicht über Echtzeitbetriebssysteme, Teil a.

Hersteller	**HightTec EDV-Syst.**	**Kadak**	**Lynx RT Systems inc.**	**Microtec Resaerch**
Produkt	**PXROS**	**AMX**	**LynxOS**	**VRTXsa**
Allgemeine Übersicht				
Kategorie	EZK	EZK, embedded		EZBS, EZK, embedded
Zielsystem	80x86, 8016x, PowerPC	80x86, 680x0, i960 AMD29k, Z80, 64180 8085	80x86, 680x0, i860 PowerPC, 88000, SPARC, MIPS, RS6000	680x0, 80x86, CPU32, i960
Hostsystem	UNIX, SUN, MS-Dos Windows, OS/2	MS-Dos	UNIX	UNIX, SUN, MS-Dos Windows
Sprache	Ansi-C, C++	ASM, Ansi-C, C++	Ansi-C, C++, Pascal, Ada Modula, Fortran	ASM, Ansi-C, C++
Entwicklungs-system	GNU C++	C/C++: Borland, MS-Visual, Watcom	GNU	Microtec-C-Compiler
Debugger	GNU	Borland, MS-Visual Metaware, Watcom	GNU, Multitask/Multithread-Debugger	Microtec XRAY-Debugger SPECTRE
Dateisystem	UNIX, DOS	DOS	UNIX, DOS, NFS, Real-Time-Filesystem	UNIX, DOS
Grafiksystem	X-Win	-	X-Win	-
Netzwerk	TCP/IP, NFS	-	TCP/IP, NFS	TCP/IP, Netware Streamed-based-Library
Feldbus	PROFIBUS, CAN-Bus	-	-	PROFIBUS,CAN-Bus, LON
Sonstiges	ROM-fähig Multiprozessor, UNIX-Prozeßinterface	ROM-fähig	ROM-fähig Multiprozessor, Self-hosted	32-Bit-Code, ROM-fähig Multiprozessor
Leistungs-daten:				
Tasks	unbegrenzt	keine Angabe	unbegrenzt	unbegrenzt
Prioritäten	64	127	256	256
Scheduler	preemtive, prioritätgesteuert	preemtive, prioritätgesteuert	preemtive, round robin, prioritätgesteuert	preemtive, round robin, prioritätgesteuert
Taskkommu-nikation	Semaphore, Messages Mailbox, Signale, Kanäle	Semaphore, Messages Mailbox, Signale	Semaphore, Messages Mailbox, Signale, Pipes, shared Memory	Semaphore, Messages Mailbox, Signale, Pipes
Sonstiges	verschiedene Speicherklassen		Memory-Locking für Tasks, patentiertes Kernel-Thread-Konzept	Message-Queues, Prioritätenvererbung

Tabelle A.2: Übersicht über Echtzeitbetriebssysteme, Teil b.

Hersteller	**Microtec Research**	**Microtec Research**	**Microware**	**Modcomp**
Produkt	**VRTX32**	**VRTXmc**	**OS/9OS-9000**	**Realix**
Allgemeine Übersicht				
Kategorie	EZBS, EZK, embedded	EZBS, EZK, embedded	EZBS, EZK, embedded	EZ-Unix
Zielsystem	680x0, 80x86, SPARC, CPU32, AMD29k, i960	680x0, CPU32	690x0, CPU32, 80x86, PowerPC	keine Angabe
Hostsystem	UNIX, SUN, MS-Dos, Windows	UNIX, SUN, MS-Dos Windows	UNIX, SUN, MS-Dos Windows, Resident	REAL/iX
Sprache	ASM, Ansi-C, ADA	ASM, Ansi-C, ADA,	Ansi-C, C++	Ansi-C, C++ ADA Pascal
Entwicklungs-system	Microtec-C-Compiler	Microtec-C-Compiler	FasTrak, Ultra C	Standard UNIX, GNU
Debugger	Microtex XRAY-Debugger SPECTRA	Microtex XRAY-Debugger SPECTRA	FasTrak, Ultra C	GNU
Dateisystem	UNIX, DOS	-	DOS, privat	UNIX
Grafiksystem	-	-	X-Win	X-Win
Netzwerk	TCP/IP, Netware Streamed-based-Library	-	TCP/IP, OS/9-net, NeWLink	TCP/IP
Feldbus	PROFIBUS, CAN-Bus LON		PROFIBUS, CAN-Bus Interbus	PROFIBUS, CAN-Bus Interbus, Sinec H1, u.a.
Sonstiges	32-Bit, ROM-fähig Multiprozessor	32-Bit, ROM-fähig	32-Bit, ROM-fähig Multiprozessor	32-Bit, Multiprozessor
Leistungs-daten:				
Tasks	unbegrentzt	256	unbegrenzt	unbegrenzt
Prioritäten	256	256	unbegrenzt	256
Scheduler	preemtive, round robin, prioritätsgesteuert	preemtive, prioritätsgesteuert,	preemtive, round robin, prioritätsgesteuert kooperative	preemtive, round robin, prioritätsgesteuert, time-sharing
Taskkommu-nikation	Semephore, Messages, Mailbox, Signale, Pipes, Mutex	Mailbox, Signale	Semephore, Signale, Pipes, Datamodule	Semephore, Messages, Signal. Pipes
Sonstiges	Message-Queues	Message-Queues		Prioritätenvererbung Message-Queues Signalqueues

Tabelle A.3: Übersicht über Echtzeitbetriebssysteme, Teil c.

Hersteller	**OnTime Informatik**	**QNX Software Systems**	**Siemens**	**WindRiver**
Produkt	**RTKernel**	**QNX**	**RMOS**	**VxWorks**
Allgemeine Übersicht				
Kategorie	EZK, embedded	EZK, embedded, EZBS	EZK, embedded	EZK, embedded, EZBS
Zielsystem	80x86, NEC-Vxx	80x86	80x86, 680x0, 683x0	680x0, 80x86, 8016x PowerPC, CPU32, i960, SPARC, AMD29k, MIPS, 88100, Hitachi SH
Hostsystem	MS-Dos, Windows	QNX	MS-Dos, Windows UNIX, SunOS, VMS	MS-Dos, Windows UNIX, SUN
Sprache	ASM, Ansi-C, C++ Pascal	WATCOM C, C++, Inline-ASM	ASM, Ansi-C, C++ STEP (SoftPLC)	ASM, Ansi-C, C++
Entwicklungs-system	Borland C++, Microsoft C/C++, Visual C Borland Pascal	WATCOM	SICOMP RMOS Organon	VxGnu-Toolkit, WindRiver
Debugger	Borland TD, Microsoft CodeView	WATCOM	SICOMP RMOS Organon	WinView, VxGDB VxSim, StethoScope
Dateisystem	DOS	UNIX, DOS, ISO9660	DOS	UNIX (RT-11), DOS
Grafiksystem	MS-Windows	X-Win (Motif), QNX-Win, Photon	MS-Windows	X-Win
Netzwerk	Netware	TCP/IP, NFS, SNMP Streams	TCP/IP, Netware PC/NFS	TCP/IP, NFS, SNMP, Streams
Feldbus	-	-	SINEC H1 & L2, Bitbus, 3964R	-
Sonstiges	16/32-Bit-Code, ROM-fähig	32-Bit, ROM-fähig Multiprozessor, POSIX 1003	32-Bit, ROM-fähig Datenaustausch über DDE, Self-hosted	32-Bit, ROM-fähig Multiprozessor, POSIX 1003
Leistungs-daten:				
Tasks	unbegrenzt	300	keine Angabe	unbegrenzt
Prioritäten	64	32	keine Angabe	256
Scheduler	preemtive, round robin prioritätsges., kooperativ	preemtive, round robin prioritätsgesteuert	keine Angabe	preemtive, round robin prioritätsgesteuert
Taskkommu-nikation	Semaphore, Messages Mailbox	Semaphore, Messages Mailbox, Signale, shared-Memory	keine Angabe	Semaphore, Messages Mailbox, Signale, shared-Memory
Sonstiges	Prioritätenvererbung	Prioritätenvererbung, Message-Queues	keine Angabe	Prioritätenvererbung, Message-Queues

Tabelle A.4: Übersicht über Echtzeitbetriebssysteme, Teil d.

LITERATURVERZEICHNIS

Aström, K.-J., Wittenmark, B. (1990). Computer Controlled Systems. Prentice-Hall International Editions. Second edition.

Bergeon, B., Zolghadri, A., Benzian Z., Ermine, J.L., Monsion M. (1993). Specification of a real-time knowledge based supervision system. Proceedings of the 12th Triennial World Congress of IFAC, PERGAMON, pp. 571-6

Chen, I.-R., Tsao, T.-W., Bastani, F. (1993). Reliability of Uniprocessor and Multiprocessor Real-Time Artificial Intelligence Planning System. Proceedings. Fourth International Symposium on Software Relaiability Engineering, IEEE Computer Society Press, pp. 160-167

Franzi, E. (1994). The low level BIOS of Khepera. Rev. 3.2, K-Team, LAMI-EPFL, Loussane

Grasman, R. (1996). Lokale Kommunikation und Steuerungskopplung mit mobilen Miniaturrobotern. Universität Karlsruhe. Institut für Prozessrechentechnik und Robotik, Diplomarbeit

Heinzman, J. (1995). Konzeption eines verhaltensbasierten Steuerungsansatzes für mehrere Miniaturroboter, Universität Karlsruhe, Institut für Prozessrechentechnik und Robotik, Studienarbeit

Hellqvist, J. (1995). Konzeption und Realisierung eines verteilten Steuerungsansatzes für mehrere Miniaturroboter. Universität Karlsruhe, Institut für Prozessrechentechnik und Robotik, Studienarbeit

Herrtwich, R.G., Hommel, G. (1989). Kooperation und Konkurrenz. Springer

Johannesson, G. (1994). Objektorienterad processautomation med SattLine. Studentlitteratur, Lund

Kernighan, Ritchie. (1990).Programmieren in C. Carl Hanser Verlag München Wien

K-Team. (1995). General I/O Turret User Manual, Version 2.0, K-Team S.A., Lousanne

K-Team. (1994). Gripper User Manual, Version 1.0, EPFL, Lousanne

K-Team. (1994). User Manual, Version 3.0, EPFL, Lousanne

Lauber, R .(1989). Prozeßautomatisierung. Springer

Libes, D. (1995). Exploring Expect. O'Reilly Associates, Inc.

Lüth, T. (1996). Distributed Architectures for Multiple-Robot Control, submitted January 1996 for review to RSJ Journal "Advanced Robotics", Special Issue in Multi-Robot Cooperation, Vol. 10, No 6

Lüth, T., Längle, T., Heinzman, J. (1995). Dynamic task-mapping for real-time controller of distributed cooperative robot systems. 13 th IFAC Workshop on Distributed Control Systems DCCS'95, Toulouse, France

Lüth, T., Längle, T., Hellqvist, J .(1995). Verteilte Robotersteuerung mit strukturadaptiven Steuerungsarchitekturen.In Dillman, R., Remboldt, U., Lüth, T. Autonome Mobile Systeme 95, Springer-Verlag, pp. 250-258

Magnusson, B.,Henriksson R .(1995).Garbage Collection for Control Systems. Department of Computer Science, Lund University of Technology, Lund, Sweden

Nielsen, N., Arzen, K.-J .(1995). Computer Implementation of Control Systems. Department of Automatic Control, Lund Institute of Technology, Report

Nielsen, N., Andersson, L., Andersson, M., Arzen, K.-J .(1995). A realtime kernel with graphics support modules. Department of Automatic Control, Lund Institute of Technology, ISSN 0280-5316

Ousterhout, J.K .(1994). Tcl and the TK Toolkit. Addison-Wesley

Roux, O.H., Martineau, P .(1995). Deadlock prevention in a distributed real-time system. Preprints of the 13th IFAC Workshop, Toulouse-Blagnac, France, pp 125-130

Schrott, G .(1994). A distributed Task-Oriented Real-Time Programming System. Annual Rewiew in Automatic Programming, volume 18, Pergamon, pp 163-166

Smith, M.G .(1992). An Enviroment for More Easily Programming a Robot. Proceedings of the 1992 IEEE International Conference on Robotics and Automation, Nice, France

Scneider, S.A, Chen V.W., Pardo-Castellote, G .(1995) The ControlShell Component-Based Real-Time Programming System. Proceedings of the 1995 IEEE International Conference on Robotics and Automation, Nagoya, Aichi, Japan,
pp 2381-2388

Tindell, K .(1995). Real Time Systems and Fixed Priority Scheduling. Department of Computer Systems, Uppsala University, Report

Vega Saenz, L., Thomesse, J.-P .(1995). Temporal properties in distributed real-time applications cooperation models and communication
types, Preprints of the 13th IFAC Workshop, Toulouse-Blagnac, France, pp 91-96

Wettstein, H .(1984]. Architektur von Betriebssystemen. Carl Hanser Verlag, München

Wollert, J., Fiedler, J .(1996). Echtzeitbetriebssysteme, Automatisierungstechnische Praxis, 38, 1, pp 33-44